АЛГОРИТМЫ
ПОСТРОЕНИЕ И АНАЛИЗ
ВТОРОЕ ИЗДАНИЕ

Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein

INTRODUCTION TO ALGORITHMS

SECOND EDITION

The MIT Press
Cambridge, Massachusetts London, England

McGraw-Hill Book Company
Boston Burr Ridge, IL Dubuque, IA Madison, WI
New York San Francisco St. Louis Montrйal Toronto

Томас Кормен
Чарльз Лейзерсон
Рональд Ривест
Клиффорд Штайн

АЛГОРИТМЫ
ПОСТРОЕНИЕ И АНАЛИЗ

ВТОРОЕ ИЗДАНИЕ

Москва • Санкт-Петербург • Киев
2011

ББК 32.973.26-018.2.75
В24
УДК 681.3.07

Издательский дом “Вильямс”

Зав. редакцией *С.Н. Тригуб*

Перевод с английского канд. техн. наук *И.В. Красикова*, *Н.А. Ореховой*, *В.Н. Романова*

Под редакцией канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Кормен, Томас Х., **Лейзерсон**, Чарльз И., **Ривест**, Рональд Л., **Штайн**, Клиффорд.

В24 Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2011. — 1296 с. : ил. — Парал. тит. англ.

ISBN 978-5–8459–0857–5 (рус.)

Фундаментальный труд известных специалистов в области кибернетики достоин занять место на полке любого человека, чья деятельность так или иначе связана с информатикой и алгоритмами. Для профессионала эта книга может служить настольным справочником, для преподавателя — пособием для подготовки к лекциям и источником интересных нетривиальных задач, для студентов и аспирантов — отличным учебником. Каждый может найти в ней именно тот материал, который касается интересующей его темы, и изложенный именно с тем уровнем сложности и строгости, который требуется читателю.

Описание алгоритмов на естественном языке дополняется псевдокодом, который позволяет любому имеющему хотя бы начальные знания и опыт программирования, реализовать алгоритм на используемом им языке программирования. Строгий математический анализ и обилие теорем сопровождаются большим количеством иллюстраций, элементарными рассуждениями и простыми приближенными оценками. Широта охвата материала и степень строгости его изложения дают основания считать эту книгу одной из лучших книг, посвященных разработке и анализу алгоритмов.

ББК 32.973.26–018.2.75

ISBN 978-5–8459–0857–5 (рус.)
ISBN 0–07–013151–1 (англ.)

ОГЛАВЛЕНИЕ

Введение 30
Часть I. Основы 43
Глава 1. Роль алгоритмов в вычислениях 46
Глава 2. Приступаем к изучению 57
Глава 3. Рост функций 87
Глава 4. Рекуррентные соотношения 109
Глава 5. Вероятностный анализ и рандомизированные алгоритмы 140
Часть II. Сортировка и порядковая статистика 173
Глава 6. Пирамидальная сортировка 178
Глава 7. Быстрая сортировка 198
Глава 8. Сортировка за линейное время 220
Глава 9. Медианы и порядковые статистики 240
Часть III. Структуры данных 255
Глава 10. Элементарные структуры данных 260
Глава 11. Хеш-таблицы 282
Глава 12. Бинарные деревья поиска 316
Глава 13. Красно-черные деревья 336
Глава 14. Расширение структур данных 365
Часть IV. Усовершенствованные методы разработки и анализа 383
Глава 15. Динамическое программирование 386
Глава 16. Жадные алгоритмы 442
Глава 17. Амортизационный анализ 482
Часть V. Сложные структуры данных 511
Глава 18. B-деревья 515
Глава 19. Биномиальные пирамиды 537

Глава 20. Фибоначчиевы пирамиды 558
Глава 21. Структуры данных для непересекающихся множеств 581
Часть VI. Алгоритмы для работы с графами 607
Глава 22. Элементарные алгоритмы для работы с графами 609
Глава 23. Минимальные остовные деревья 644
Глава 24. Кратчайшие пути из одной вершины 663
Глава 25. Кратчайшие пути между всеми парами вершин 708
Глава 26. Задача о максимальном потоке 734
Часть VII. Избранные темы 795
Глава 27. Сортирующие сети 799
Глава 28. Работа с матрицами 823
Глава 29. Линейное программирование 869
Глава 30. Полиномы и быстрое преобразование Фурье 926
Глава 31. Теоретико-числовые алгоритмы 954
Глава 32. Поиск подстрок 1017
Глава 33. Вычислительная геометрия 1047
Глава 34. NP-полнота 1085
Глава 35. Приближенные алгоритмы 1151
Часть VIII. Приложения: математические основы 1189
Приложение А. Ряды 1191
Приложение Б. Множества и прочие художества 1202
Приложение В. Комбинаторика и теория вероятности 1226
Библиография 1257
Предметный указатель 1277

СОДЕРЖАНИЕ

Введение 30

Часть I. Основы 43
Введение 44

Глава 1. Роль алгоритмов в вычислениях 46
1.1 Алгоритмы 46
Какие задачи решаются с помощью алгоритмов? 47
Структуры данных 50
Методические указания 50
Сложные задачи 51
Упражнения 52
1.2 Алгоритмы как технология 52
Эффективность 52
Алгоритмы и другие технологии 54
Упражнения 55
Задачи 55
Заключительные замечания 56

Глава 2. Приступаем к изучению 57
2.1 Сортировка вставкой 57
Инварианты цикла и корректность сортировки вставкой 59
Соглашения, принятые при составлении псевдокода 61
Упражнения 63
2.2 Анализ алгоритмов 64
Анализ алгоритма, работающего по методу вставок 66

Наихудшее и среднее время работы 69
Порядок возрастания 70
Упражнения 71
2.3 Разработка алгоритмов 71
2.3.1 Метод декомпозиции 72
2.3.2 Анализ алгоритмов, основанных на принципе "разделяй и властвуй" 78
Анализ алгоритма сортировки слиянием 78
Упражнения 81
Задачи 83
Заключительные замечания 86

Глава 3. Рост функций 87
3.1 Асимптотические обозначения 88
Θ-обозначения 88
O-обозначения 91
Ω-обозначения 92
Асимптотические обозначения в уравнениях и неравенствах 93
o-обозначения 94
ω-обозначения 95
Сравнение функций 96
Упражнения 97
3.2 Стандартные обозначения и часто встречающиеся функции 98
Монотонность 98
Округление в большую и меньшую сторону 98
Модульная арифметика 98
Полиномы 99
Показательные функции 99
Логарифмы 100
Факториалы 102
Функциональная итерация 102
Итерированная логарифмическая функция 103
Числа Фибоначчи 103
Упражнения 104
Задачи 105
Заключительные замечания 108

Глава 4. Рекуррентные соотношения 109
Технические детали 110
4.1 Метод подстановки 111
Как угадать решение 112

Тонкие нюансы 113
Остерегайтесь ошибок 114
Замена переменных 114
Упражнения 115
4.2 Метод деревьев рекурсии 115
Упражнения 120
4.3 Основной метод 121
Основная теорема 121
Использование основного метода 122
Упражнения 123
⋆ 4.4 Доказательство основной теоремы 124
4.4.1 Доказательство теоремы для точных степеней 125
4.4.2 Учет округления чисел 130
Упражнения 133
Задачи 133
Заключительные замечания 138

Глава 5. Вероятностный анализ и рандомизированные алгоритмы 140
5.1 Задача о найме сотрудника 140
Анализ наихудшего случая 142
Вероятностный анализ 142
Рандомизированные алгоритмы 143
Упражнения 144
5.2 Индикаторная случайная величина 144
Анализ задачи о найме сотрудника с помощью индикаторных случайных величин 146
Упражнения 148
5.3 Рандомизированные алгоритмы 149
Массивы, полученные в результате случайной перестановки 151
Упражнения 155
⋆ 5.4 Вероятностный анализ и дальнейшее применение индикаторных случайных величин 156
5.4.1 Парадокс дней рождения 157
Анализ с помощью индикаторных случайных величин 158
5.4.2 Шары и урны 160
5.4.3 Последовательности выпадения орлов 161
5.4.4 Задача о найме сотрудника в оперативном режиме 165
Упражнения 167
Задачи 168
Заключительные замечания 171

Часть II. Сортировка и порядковая статистика 173
Введение 174

Глава 6. Пирамидальная сортировка 178
6.1 Пирамиды 179
Упражнения 181
6.2 Поддержка свойства пирамиды 182
Упражнения 184
6.3 Создание пирамиды 184
Упражнения 187
6.4 Алгоритм пирамидальной сортировки 187
Упражнения 188
6.5 Очереди с приоритетами 190
Упражнения 193
Задачи 194
Заключительные замечания 196

Глава 7. Быстрая сортировка 198
7.1 Описание быстрой сортировки 199
Разбиение массива 199
Упражнения 203
7.2 Производительность быстрой сортировки 203
Наихудшее разбиение 203
Наилучшее разбиение 204
Сбалансированное разбиение 204
Интуитивные рассуждения для среднего случая 205
Упражнения 207
7.3 Рандомизированная версия быстрой сортировки 208
Упражнения 209
7.4 Анализ быстрой сортировки 209
7.4.1 Анализ в наихудшем случае 209
7.4.2 Математическое ожидание времени работы 210
Время работы и сравнения 210
Упражнения 213
Задачи 214
Заключительные замечания 219

Глава 8. Сортировка за линейное время 220
8.1 Нижние оценки алгоритмов сортировки 221
Модель дерева решений 221
Нижняя оценка для наихудшего случая 222
Упражнения 223

8.2 Сортировка подсчетом 224
Упражнения 226
8.3 Поразрядная сортировка 226
Упражнения 230
8.4 Карманная сортировка 230
Упражнения 234
Задачи 234
Заключительные замечания 238

Глава 9. Медианы и порядковые статистики 240
9.1 Минимум и максимум 241
Одновременный поиск минимума и максимума 241
Упражнения 242
9.2 Выбор в течение линейного ожидаемого времени 243
Упражнения 247
9.3 Алгоритм выбора с линейным временем работы в наихудшем случае 247
Упражнения 250
Задачи 252
Заключительные замечания 254

Часть III. Структуры данных 255
Введение 256

Глава 10. Элементарные структуры данных 260
10.1 Стеки и очереди 260
Стеки 260
Очереди 262
Упражнения 263
10.2 Связанные списки 264
Поиск в связанном списке 265
Вставка в связанный список 265
Удаление из связанного списка 266
Ограничители 266
Упражнения 268
10.3 Реализация указателей и объектов 269
Представление объектов с помощью нескольких массивов 269
Представление объектов с помощью одного массива 270
Выделение и освобождение памяти 271
Упражнения 273
10.4 Представление корневых деревьев 274
Бинарные деревья 274

Корневые деревья с произвольным ветвлением 275
Другие представления деревьев 276
Упражнения 276
Задачи 277
Заключительные замечания 280

Глава 11. Хеш-таблицы 282
11.1 Таблицы с прямой адресацией 283
Упражнения 284
11.2 Хеш-таблицы 285
Разрешение коллизий при помощи цепочек 286
Анализ хеширования с цепочками 288
Упражнения 290
11.3 Хеш-функции 291
Чем определяется качество хеш-функции 291
Интерпретация ключей как целых неотрицательных чисел 292
11.3.1 Метод деления 292
11.3.2 Метод умножения 293
⋆ 11.3.3 Универсальное хеширование 294
Построение универсального множества хеш-функций 297
Упражнения 298
11.4 Открытая адресация 300
Линейное исследование 302
Квадратичное исследование 303
Двойное хеширование 303
Анализ хеширования с открытой адресацией 305
Упражнения 307
⋆ 11.5 Идеальное хеширование 308
Упражнения 312
Задачи 313
Заключительные замечания 315

Глава 12. Бинарные деревья поиска 316
12.1 Что такое бинарное дерево поиска 317
Упражнения 319
12.2 Работа с бинарным деревом поиска 319
Поиск 320
Поиск минимума и максимума 321
Предшествующий и последующий элементы 321
Упражнения 323
12.3 Вставка и удаление 324
Вставка 324

Удаление 325
Упражнения 327
⋆ 12.4 Случайное построение бинарных деревьев поиска 328
Упражнения 331
Задачи 332
Заключительные замечания 335

Глава 13. Красно-черные деревья 336
13.1 Свойства красно-черных деревьев 336
Упражнения 339
13.2 Повороты 340
Упражнения 341
13.3 Вставка 342
Анализ 350
Упражнения 350
13.4 Удаление 351
Анализ 356
Упражнения 356
Задачи 357
Заключительные замечания 364

Глава 14. Расширение структур данных 365
14.1 Динамические порядковые статистики 366
Выборка элемента с заданным рангом 367
Определение ранга элемента 368
Поддержка размера поддеревьев 369
Упражнения 371
14.2 Расширение структур данных 372
Расширение красно-черных деревьев 373
Упражнения 374
14.3 Деревья отрезков 375
Упражнения 380
Задачи 381
Заключительные замечания 382

Часть IV. Усовершенствованные методы разработки и анализа 383
Введение 384

Глава 15. Динамическое программирование 386
15.1 Расписание работы конвейера 387
Первый этап: структура самой быстрой сборки 389
Второй этап: рекурсивное решение 391

Третий этап: вычисление минимальных промежутков времени 393
Четвертый этап: построение самого быстрого пути 394
Упражнения 395
15.2 Перемножение цепочки матриц 395
Подсчет количества способов расстановки скобок 397
Первый этап: структура оптимальной расстановки скобок 398
Второй этап: рекурсивное решение 399
Третий этап: вычисление оптимальной стоимости 400
Четвертый этап: конструирование оптимального решения 403
Упражнения 404
15.3 Элементы динамического программирования 404
Оптимальная подструктура 405
Перекрытие вспомогательных задач 411
Построение оптимального решения 414
Запоминание 414
Упражнения 417
15.4 Самая длинная общая подпоследовательность 418
Этап 1: характеристика самой длинной общей подпоследовательности 419
Этап 2: рекурсивное решение 421
Этап 3: вычисление длины самой длинной общей подпоследовательности 422
Этап 4: построение самой длинной общей подпоследовательности 423
Улучшение кода 424
Упражнения 425
15.5 Оптимальные бинарные деревья поиска 425
Этап 1: структура оптимального бинарного дерева поиска 429
Этап 2: рекурсивное решение 430
Этап 3: вычисление математического ожидания стоимости поиска в оптимальном бинарном дереве поиска 431
Упражнения 433
Задачи 434
Заключительные замечания 440

Глава 16. Жадные алгоритмы 442
16.1 Задача о выборе процессов 443
Оптимальная подструктура задачи о выборе процессов 444
Рекурсивное решение 446

Преобразование решения динамического программирования в жадное решение 446
Рекурсивный жадный алгоритм 449
Итерационный жадный алгоритм 451
Упражнения 452
16.2 Элементы жадной стратегии 453
Свойство жадного выбора 454
Оптимальная подструктура 455
Сравнение жадных алгоритмов и динамического программирования 456
Упражнения 458
16.3 Коды Хаффмана 459
Префиксные коды 460
Построение кода Хаффмана 462
Корректность алгоритма Хаффмана 464
Упражнения 466
⋆ 16.4 Теоретические основы жадных методов 467
Матроиды 467
Жадные алгоритмы на взвешенном матроиде 470
Упражнения 474
⋆ 16.5 Планирование заданий 474
Упражнения 478
Задачи 478
Заключительные замечания 481

Глава 17. Амортизационный анализ 482
17.1 Групповой анализ 483
Стековые операции 483
Приращение показаний бинарного счетчика 485
Упражнения 487
17.2 Метод бухгалтерского учета 487
Стековые операции 489
Приращение показаний бинарного счетчика 490
Упражнения 490
17.3 Метод потенциалов 491
Стековые операции 492
Увеличение показаний бинарного счетчика 493
Упражнения 494
17.4 Динамические таблицы 495
17.4.1 Расширение таблицы 496
17.4.2 Расширение и сжатие таблицы 499

Упражнения 504
Задачи 505
Заключительные замечания 510

Часть V. Сложные структуры данных 511
Введение 512

Глава 18. B-деревья 515
Структуры данных во вторичной памяти 516
18.1 Определение B-деревьев 519
Высота B-дерева 521
Упражнения 522
18.2 Основные операции с B-деревьями 522
Поиск в B-дереве 522
Создание пустого B-дерева 523
Вставка ключа в B-дерево 524
Упражнения 528
18.3 Удаление ключа из B-дерева 530
Упражнения 533
Задачи 533
Заключительные замечания 536

Глава 19. Биномиальные пирамиды 537
19.1 Биномиальные деревья и биномиальные пирамиды 539
19.1.1 Биномиальные деревья 539
19.1.2 Биномиальные пирамиды 541
Упражнения 543
19.2 Операции над биномиальными пирамидами 544
Создание новой биномиальной пирамиды 544
Поиск минимального ключа 544
Слияние двух биномиальных пирамид 545
Вставка узла 550
Извлечение вершины с минимальным ключом 551
Уменьшение ключа 552
Удаление ключа 554
Упражнения 554
Задачи 555
Заключительные замечания 557

Глава 20. Фибоначчиевы пирамиды 558
20.1 Структура фибоначчиевых пирамид 559
Потенциальная функция 561
Максимальная степень 562
20.2 Операции над сливаемыми пирамидами 562
Создание новой фибоначчиевой пирамиды 563
Вставка узла 563
Поиск минимального узла 564
Объединение двух фибоначчиевых пирамид 564
Извлечение минимального узла 565
Упражнения 571
20.3 Уменьшение ключа и удаление узла 571
Уменьшение ключа 571
Удаление узла 575
Упражнения 575
20.4 Оценка максимальной степени 575
Упражнения 578
Задачи 578
Заключительные замечания 579

Глава 21. Структуры данных для непересекающихся множеств 581
21.1 Операции над непересекающимися множествами 582
Приложение структур данных для непересекающихся множеств 583
Упражнения 584
21.2 Представление непересекающихся множеств с помощью связанных списков 585
Простая реализация объединения 586
Весовая эвристика 587
Упражнения 588
21.3 Лес непересекающихся множеств 589
Эвристики для повышения эффективности 589
Псевдокоды 590
Влияние эвристик на время работы 592
Упражнения 592
⋆ 21.4 Анализ объединения по рангу со сжатием пути 592
Очень быстро и очень медленно растущая функция 593
Свойства рангов 594
Доказательство границы времени работы 595
Потенциальная функция 596

Изменения потенциала и амортизированная стоимость операций 598
Упражнения 601
Задачи 601
Заключительные замечания 605

Часть VI. Алгоритмы для работы с графами 607
Введение 608

Глава 22. Элементарные алгоритмы для работы с графами 609
22.1 Представление графов 609
Упражнения 612
22.2 Поиск в ширину 613
Анализ 616
Кратчайшие пути 617
Деревья поиска в ширину 620
Упражнения 621
22.3 Поиск в глубину 622
Свойства поиска в глубину 626
Классификация ребер 628
Упражнения 630
22.4 Топологическая сортировка 632
Упражнения 634
22.5 Сильно связные компоненты 635
Упражнения 640
Задачи 641
Заключительные замечания 643

Глава 23. Минимальные остовные деревья 644
23.1 Построение минимального остовного дерева 645
Упражнения 649
23.2 Алгоритмы Крускала и Прима 651
Алгоритм Крускала 651
Алгоритм Прима 653
Упражнения 656
Задачи 658
Заключительные замечания 661

Глава 24. Кратчайшие пути из одной вершины 663
Варианты 664
Оптимальная структура задачи о кратчайшем пути 665
Ребра с отрицательным весом 666

Циклы 667
Представление кратчайших путей 668
Ослабление 669
Свойства кратчайших путей и ослабления 671
Краткое содержание главы 672
24.1 Алгоритм Беллмана-Форда 672
Упражнения 676
24.2 Кратчайшие пути из одной вершины в ориентированных ациклических графах 677
Упражнения 679
24.3 Алгоритм Дейкстры 680
Анализ 684
Упражнения 686
24.4 Разностные ограничения и кратчайшие пути 687
Линейное программирование 687
Системы разностных ограничений 688
Графы ограничений 690
Решение систем разностных ограничений 692
Упражнения 692
24.5 Доказательства свойств кратчайших путей 694
Неравенство треугольника 694
Влияние ослабления на оценки кратчайшего пути 695
Ослабление и деревья кратчайших путей 697
Упражнения 700
Задачи 702
Заключительные замечания 706

Глава 25. Кратчайшие пути между всеми парами вершин 708
Краткое содержание главы 710
25.1 Задача о кратчайших путях и умножение матриц 711
Структура кратчайшего пути 711
Рекурсивное решение задачи о кратчайших путях между всеми парами вершин 712
Вычисление весов кратчайших путей в восходящем порядке 712
Улучшение времени работы 714
Упражнения 716
25.2 Алгоритм Флойда-Варшалла 718
Структура кратчайшего пути 718
Рекурсивное решение задачи о кратчайших путях между всеми парами вершин 719

Вычисление весов кратчайших путей в восходящем порядке 720
Построение кратчайшего пути 720
Транзитивное замыкание ориентированного графа 722
Упражнения 724
25.3 Алгоритм Джонсона для разреженных графов 726
Сохранение кратчайших путей 726
Генерация неотрицательных весов путем их изменения 728
Вычисление кратчайших путей между всеми парами вершин 728
Упражнения 730
Задачи 731
Заключительные замечания 732

Глава 26. Задача о максимальном потоке 734
26.1 Транспортные сети 735
Транспортные сети и потоки 735
Пример потока 737
Сети с несколькими источниками и стоками 739
Как работать с потоками 740
Упражнения 741
26.2 Метод Форда-Фалкерсона 742
Остаточные сети 743
Увеличивающие пути 745
Разрезы транспортных сетей 746
Базовый алгоритм Форда-Фалкерсона 749
Анализ метода Форда-Фалкерсона 750
Алгоритм Эдмондса-Карпа 752
Упражнения 755
26.3 Максимальное паросочетание 756
Задача поиска максимального паросочетания в двудольном графе 757
Поиск максимального паросочетания в двудольном графе 758
Упражнения 761
⋆ 26.4 Алгоритмы проталкивания предпотока 761
Интуитивные соображения 762
Основные операции 764
Операция проталкивания 764
Операция подъема 766
Универсальный алгоритм 766
Корректность метода проталкивания предпотока 768

Анализ метода проталкивания предпотока 770
Упражнения 773
⋆ 26.5 Алгоритм “поднять-в-начало” 774
Допустимые ребра и сети 775
Списки соседей 777
Разгрузка переполненной вершины 777
Алгоритм “поднять-в-начало” 780
Анализ 783
Упражнения 785
Задачи 786
Заключительные замечания 793

Часть VII. Избранные темы 795
Введение 796

Глава 27. Сортирующие сети 799
27.1 Сравнивающие сети 800
Упражнения 803
27.2 Нуль-единичный принцип 805
Упражнения 807
27.3 Битоническая сортирующая сеть 808
Полуфильтр 809
Битонический сортировщик 810
Упражнения 812
27.4 Объединяющая сеть 813
Упражнения 815
27.5 Сортирующая сеть 816
Упражнения 818
Задачи 819
Заключительные замечания 822

Глава 28. Работа с матрицами 823
28.1 Свойства матриц 824
Матрицы и векторы 824
Операции над матрицами 827
Обратные матрицы, ранги и детерминанты 828
Положительно определенные матрицы 831
Упражнения 831
28.2 Алгоритм умножения матриц Штрассена 833
Обзор алгоритма 833
Определение произведений подматриц 834
Обсуждение метода 838

Упражнения 839
28.3 Решение систем линейных уравнений 839
Обзор LUP-разложения 841
Прямая и обратная подстановки 842
Вычисление LU-разложения 845
Вычисление LUP-разложения 848
Упражнения 852
28.4 Обращение матриц 853
Вычисление обратной матрицы из LUP-разложения 853
Умножение матриц и обращение матрицы 854
Упражнения 857
28.5 Симметричные положительно определенные матрицы и метод наименьших квадратов 858
Метод наименьших квадратов 861
Упражнения 865
Задачи 865
Заключительные замечания 867

Глава 29. Линейное программирование 869
Политическая задача 869
Общий вид задач линейного программирования 872
Краткий обзор задач линейного программирования 872
Приложения линейного программирования 876
Алгоритмы решения задач линейного программирования 877
29.1 Стандартная и каноническая формы задач линейного программирования 877
Стандартная форма 878
Преобразование задач линейного программирования в стандартную форму 879
Преобразование задач линейного программирования в каноническую форму 882
Упражнения 885
29.2 Формулирование задач в виде задач линейного программирования 886
Кратчайшие пути 887
Максимальный поток 887
Поиск потока с минимальными затратами 888
Многопродуктовый поток 889
Упражнения 891
29.3 Симплекс-алгоритм 892
Пример симплекс-алгоритма 893

Замещение 897
Формальный симплекс-алгоритм 899
Завершение 905
Упражнения 907
29.4 Двойственность 908
Упражнения 914
29.5 Начальное базисное допустимое решение 914
Поиск начального решения 914
Основная теорема линейного программирования 920
Упражнения 921
Задачи 922
Заключительные замечания 924

Глава 30. Полиномы и быстрое преобразование Фурье 926
Полиномы 926
Краткое содержание главы 928
30.1 Представление полиномов 928
Представление, основанное на коэффициентах 929
Представление, основанное на значениях в точках 929
Быстрое умножение полиномов, заданных в коэффициентной форме 932
Упражнения 934
30.2 ДПФ и БПФ 935
Комплексные корни из единицы 935
Дискретное преобразование Фурье 938
Быстрое преобразование Фурье 938
Интерполяция в точках, являющихся комплексными корнями из единицы 941
Упражнения 942
30.3 Эффективные реализации БПФ 943
Итеративная реализация БПФ 944
Параллельная схема БПФ 947
Упражнения 949
Задачи 949
Заключительные замечания 953

Глава 31. Теоретико-числовые алгоритмы 954
Размер входных наборов данных и стоимость арифметических вычислений 955
31.1 Элементарные обозначения, принятые в теории чисел 956
Делимость и делители 956
Простые и составные числа 956

Теорема о делении, остатки и равенство по модулю 957
Общие делители и наибольшие общие делители 958
Взаимно простые целые числа 960
Единственность разложения на множители 960
Упражнения 961
31.2 Наибольший общий делитель 962
Алгоритм Евклида 963
Время работы алгоритма Евклида 964
Развернутая форма алгоритма Евклида 965
Упражнения 967
31.3 Модульная арифметика 968
Конечные группы 968
Группы, образованные сложением и умножением по модулю 969
Подгруппы 972
Подгруппы, сгенерированные элементом группы 973
Упражнения 975
31.4 Решение модульных линейных уравнений 975
Упражнения 979
31.5 Китайская теорема об остатках 979
Упражнения 982
31.6 Степени элемента 983
Возведение в степень путем последовательного возведения в квадрат 985
Упражнения 987
31.7 Криптосистема с открытым ключом RSA 987
Криптографические системы с открытым ключом 988
Криптографическая система RSA 991
Упражнения 995
⋆ 31.8 Проверка простоты 995
Плотность распределения простых чисел 996
Проверка псевдопростых чисел 997
Рандомизированный тест простоты Миллера-Рабина 999
Частота ошибок в тесте Миллера-Рабина 1002
Упражнения 1006
⋆ 31.9 Целочисленное разложение 1006
Эвристический ρ-метод Полларда 1007
Упражнения 1012
Задачи 1013
Заключительные замечания 1015

Глава 32. Поиск подстрок 1017
Обозначения и терминология 1019
32.1 Простейший алгоритм поиска подстрок 1020
Упражнения 1021
32.2 Алгоритм Рабина-Карпа 1022
Упражнения 1028
32.3 Поиск подстрок с помощью конечных автоматов 1028
Конечные автоматы 1029
Автоматы поиска подстрок 1030
Вычисление функции переходов 1035
Упражнения 1036
⋆ 32.4 Алгоритм Кнута-Морриса-Пратта 1036
Префиксная функция для образца 1037
Анализ времени работы 1040
Корректность вычисления префиксной функции 1041
Корректность алгоритма Кнута-Морриса-Пратта 1043
Упражнения 1044
Задачи 1045
Заключительные замечания 1046

Глава 33. Вычислительная геометрия 1047
33.1 Свойства отрезков 1048
Векторное произведение 1049
Поворот последовательных отрезков 1050
Определение того, пересекаются ли два отрезка 1051
Другие применения векторного произведения 1053
Упражнения 1053
33.2 Определение наличия пересекающихся отрезков 1055
Упорядочение отрезков 1056
Перемещение выметающей прямой 1057
Псевдокод, выявляющий пересечение отрезков 1058
Корректность 1060
Время работы 1061
Упражнения 1062
33.3 Построение выпуклой оболочки 1063
Сканирование по Грэхему 1065
Обход по Джарвису 1071
Упражнения 1073
33.4 Поиск пары ближайших точек 1074
Алгоритм декомпозиции 1075
Корректность 1077

Реализация и время работы алгоритма 1078
Упражнения 1079
Задачи 1080
Заключительные замечания 1083

Глава 34. NP-полнота 1085
NP-полнота и классы P и NP 1087
Как показать, что задача является NP-полной 1088
Краткое содержание главы 1091
34.1 Полиномиальное время 1091
Абстрактные задачи 1092
Кодирование 1093
Структура формальных языков 1096
Упражнения 1100
34.2 Проверка за полиномиальное время 1100
Гамильтоновы циклы 1101
Алгоритмы верификации 1102
Класс сложности NP 1103
Упражнения 1105
34.3 NP-полнота и приводимость 1106
Приводимость 1106
NP-полнота 1108
Выполнимость схем 1110
Упражнения 1117
34.4 Доказательство NP-полноты 1118
Выполнимость формулы 1119
3-CNF выполнимость 1122
Упражнения 1126
34.5 NP-полные задачи 1127
34.5.1 Задача о клике 1128
34.5.2 Задача о вершинном покрытии 1131
34.5.3 Задача о гамильтоновых циклах 1133
34.5.4 Задача о коммивояжере 1138
34.5.5 Задача о сумме подмножества 1140
Упражнения 1144
Задачи 1145
Заключительные замечания 1149

Глава 35. Приближенные алгоритмы 1151
Оценка качества приближенных алгоритмов 1151
Краткое содержание главы 1153
35.1 Задача о вершинном покрытии 1154

Упражнения 1157
35.2 Задача о коммивояжере 1157
35.2.1 Задача о коммивояжере с неравенством треугольника 1158
35.2.2 Общая задача о коммивояжере 1161
Упражнения 1163
35.3 Задача о покрытии множества 1164
Жадный приближенный алгоритм 1166
Анализ 1166
Упражнения 1169
35.4 Рандомизация и линейное программирование 1170
Рандомизированный приближенный алгоритм для задачи о MAX-3-CNF выполнимости 1170
Аппроксимация взвешенного вершинного покрытия с помощью линейного программирования 1172
Упражнения 1175
35.5 Задача о сумме подмножества 1176
Точный алгоритм с экспоненциальным временем работы 1176
Схема аппроксимации с полностью полиномиальным временем работы 1178
Упражнения 1182
Задачи 1182
Заключительные замечания 1186

Часть VIII. Приложения: математические основы 1189
Введение 1190

Приложение А. Ряды 1191
А.1 Суммы и их свойства 1192
Линейность 1192
Арифметическая прогрессия 1193
Суммы квадратов и кубов 1193
Геометрическая прогрессия 1193
Гармонический ряд 1194
Интегрирование и дифференцирование рядов 1194
Суммы разностей 1194
Произведения 1195
Упражнения 1195
А.2 Оценки сумм 1195
Математическая индукция 1196
Почленное сравнение 1196
Разбиение рядов 1198

Приближение интегралами 1199
Упражнения 1201
Задачи 1201
Заключительные замечания 1201

Приложение Б. Множества и прочие художества 1202
Б.1 Множества 1202
Упражнения 1207
Б.2 Отношения 1207
Упражнения 1209
Б.3 Функции 1210
Упражнения 1212
Б.4 Графы 1213
Упражнения 1217
Б.5 Деревья 1218
Б.5.1 Свободные деревья 1218
Б.5.2 Деревья с корнем и упорядоченные деревья 1220
Б.5.3 Бинарные и позиционные деревья 1221
Упражнения 1223
Задачи 1224
Заключительные замечания 1225

Приложение В. Комбинаторика и теория вероятности 1226
В.1 Основы комбинаторики 1226
Правила суммы и произведения 1227
Строки 1227
Перестановки 1227
Сочетания 1228
Биномиальные коэффициенты 1229
Оценки биномиальных коэффициентов 1229
Упражнения 1230
В.2 Вероятность 1232
Аксиомы вероятности 1232
Дискретные распределения вероятностей 1233
Непрерывное равномерное распределение вероятности 1234
Условная вероятность и независимость 1234
Теорема Байеса 1236
Упражнения 1237
В.3 Дискретные случайные величины 1238
Математическое ожидание случайной величины 1239
Дисперсия и стандартное отклонение 1242
Упражнения 1243

B.4 Геометрическое и биномиальное распределения 1243
Геометрическое распределение 1244
Биномиальное распределение 1245
Упражнения 1248
⋆ B.5 Хвосты биномиального распределения 1249
Упражнения 1254
Задачи 1255
Заключительные замечания 1256

Библиография 1257

Предметный указатель 1277

Введение

Эта книга служит исчерпывающим вводным курсом по современным компьютерным алгоритмам. В ней представлено большое количество конкретных алгоритмов, которые описываются достаточно глубоко, однако таким образом, чтобы их разработка и анализ были доступны читателям с любым уровнем подготовки. Авторы старались давать простые объяснения без ущерба для глубины изложения и математической строгости.

В каждой главе представлен определенный алгоритм, описывается метод его разработки, область применения или другие связанные с ним вопросы. Алгоритмы описываются как на обычном человеческом языке, так и в виде псевдокода, разработанного так, что он будет понятен для всех, у кого есть хотя бы минимальный опыт программирования. В книге представлено более 230 рисунков, иллюстрирующих работу алгоритмов. Поскольку один из критериев разработки алгоритмов — их *эффективность*, описание всех алгоритмов включает в себя тщательный анализ времени их работы.

Данный учебник предназначен в первую очередь для студентов и аспирантов, изучающих тот или иной курс по алгоритмам и структурам данных. Он также будет полезен для технических специалистов, желающих повысить свой уровень в этой области, поскольку описание процесса разработки алгоритмов сопровождается изложением технических вопросов.

В настоящем втором издании в книгу внесено множество изменений на всех уровнях, — от добавления целых новых глав до пересмотра отдельных предложений.

Преподавателю

Эта книга задумана так, что разнообразие описываемых в ней тем сочетается с полнотой изложения. Она может стать полезной при чтении разнообразных курсов, — от курса по структурам данных для студентов до курса по алгоритмам для аспирантов. Поскольку в книге намного больше материала, чем требуется

для обычного курса, рассчитанного на один семестр, можно выбрать только тот материал, который лучше всего соответствует курсу, который вы собираетесь преподавать.

Курсы удобно разрабатывать на основе отдельных глав. Книга написана так, что ее главы сравнительно независимы одна от другой. В каждой главе материал излагается по мере увеличения его сложности и разбит на разделы. В студенческом курсе можно использовать только более легкие разделы, а в аспирантском — всю главу в полном объеме.

В книгу вошли более 920 упражнений и свыше 140 задач. Упражнения даются в конце каждого раздела, а задачи — в конце каждой главы. Упражнения представлены в виде кратких вопросов для проверки степени освоения материала. Некоторые из них простые и предназначены для самоконтроля, в то время как другие — посложнее и могут быть рекомендованы в качестве домашних заданий. Решение задач требует больших усилий, и с их помощью часто вводится новый материал. Обычно задачи сформулированы так, что в них содержатся наводящие вопросы, помогающие найти верное решение.

Разделы и упражнения, которые больше подходят для аспирантов, чем для студентов, обозначены звездочкой ($\star$). Они не обязательно более сложные, чем те, возле которых звездочка отсутствует; просто для их понимания может понадобиться владение более сложным математическим аппаратом. Для того чтобы справиться с упражнениями со звездочкой, может понадобиться более основательная подготовка или неординарная сообразительность.

Студенту

Надеемся, что этот учебник станет хорошим введением в теорию алгоритмов. Авторы попытались изложить каждый алгоритм в доступной и увлекательной форме. Чтобы облегчить освоение незнакомых или сложных алгоритмов, каждый из них описывается поэтапно. В книге также приводится подробное объяснение математических вопросов, необходимых для того, чтобы понять проводимый анализ алгоритмов. Для тех читателей, которые уже в некоторой мере знакомы с какой-то темой, материал глав организован таким образом, чтобы эти читатели могли опустить вводные разделы и перейти непосредственно к более сложному материалу.

Книга получилась довольно большой, поэтому не исключено, что в курсе лекций будет представлена лишь часть изложенного в ней материала. Однако авторы попытались сделать ее такой, чтобы она стала полезной как сейчас в качестве учебника, способствующего усвоению курса лекций, так и позже, в профессиональной деятельности, в качестве настольного справочного пособия для математиков или инженеров.

Ниже перечислены необходимые предпосылки, позволяющие освоить материал этой книги.

- Читатель должен обладать некоторым опытом в программировании. В частности, он должен иметь представление о рекурсивных процедурах и простых структурах данных, таких как массивы и связанные списки.
- Читатель должен обладать определенными навыками доказательства теорем методом математической индукции. Для понимания некоторых вопросов, изложенных в этой книге, потребуется умение выполнять некоторые простые математические преобразования. Помимо этого, в частях I и VIII этой книги рассказывается обо всех используемых математических методах.

Профессионалу

Широкий круг вопросов, которые излагаются в этой книге, позволяет говорить о том, что она станет прекрасным учебником по теории алгоритмов. Поскольку каждая глава является относительно самостоятельной, читатель сможет сосредоточить внимание на вопросах, интересующих его больше других.

Основная часть обсуждаемых здесь алгоритмов обладает большой практической ценностью. Поэтому не обойдены вниманием особенности реализации алгоритмов и другие инженерные вопросы. Часто предлагаются реальные альтернативы алгоритмам, представляющим преимущественно теоретический интерес.

Если вам понадобится реализовать любой из приведенных алгоритмов, будет достаточно легко преобразовать приведенный псевдокод в код на вашем любимом языке программирования. Псевдокод разработан таким образом, чтобы каждый алгоритм был представлен ясно и лаконично. Вследствие этого не рассматриваются обработка ошибок и другие связанные с разработкой программного обеспечения вопросы, требующие определенных предположений, касающихся конкретной среды программирования. Авторы попытались представить каждый алгоритм просто и непосредственно, не используя индивидуальные особенности того или иного языка программирования, что могло бы усложнить понимание сути алгоритма.

Коллегам

В книге приведена обширная библиография и представлены указания на современную литературу. В конце каждой главы даются “заключительные замечания”, содержащие исторические подробности и ссылки. Однако эти замечания не могут служить исчерпывающим руководством в области алгоритмов. Возможно, в это будет сложно поверить, но даже в такую объемную книгу не удалось включить многие интересные алгоритмы из-за недостатка места.

Несмотря на огромное количество писем от студентов с просьбами предоставить решения задач и упражнений, политика авторов — не приводить ссылки на

источники, из которых эти задачи и упражнения были позаимствованы. Это сделано для того, чтобы студенты не искали готовые решения в литературе, а решали задачи самостоятельно.

Изменения во втором издании

Какие изменения произошли во втором издании книги? В зависимости от точки зрения, можно сказать, что их достаточно мало или довольно много.

Бóльшая часть глав первого издания содержится и во втором издании. Авторы убрали две главы и некоторые разделы, но добавили три новые главы и (кроме них) четыре новых раздела. Если судить об изменениях по оглавлению, то, скорее всего, можно прийти к выводу, что их объем достаточно скромный.

Однако эти изменения далеко выходят за рамки оглавления. Ниже без определенного порядка приводится обзор наиболее значительных изменений во втором издании.

- К коллективу соавторов присоединился Клифф Штайн (Cliff Stein).
- Были исправлены ошибки. Сколько их было допущено? Ну, скажем, несколько.
- Появились три новые главы:
  - в главе 1 обсуждается роль алгоритмов в вычислениях;
  - в главе 5 излагается вероятностный анализ и рандомизированные алгоритмы; в первом издании эта тема упоминается в разных местах книги;
  - глава 29 посвящена линейному программированию.
- Что касается глав, которые присутствовали в первом издании, в них добавлены новые разделы по таким вопросам:
  - идеальное хеширование (раздел 11.5);
  - два приложения динамического программирования (разделы 15.1 и 15.5);
  - алгоритмы аппроксимации, в которых применяется рандомизация и линейное программирование (раздел 35.4).
- Чтобы поместить в начальную часть книги больше алгоритмов, три главы по основным разделам математики перенесены из части I в приложение (часть VIII).
- Добавлено более 40 новых задач и 185 новых упражнений.
- В явном виде были использованы инварианты цикла для доказательства корректности алгоритмов. Первый инвариант цикла используется в главе 2, и в дальнейшем в книге они применяются еще несколько десятков раз.

- Переделаны многие места, где проводится вероятностный анализ. В частности, более десяти раз используется метод “индикаторных случайных величин”, упрощающих вероятностный анализ, особенно при наличии зависимости между случайными величинами.
- Дополнены и обновлены заключительные замечания к главам и библиография. Библиография увеличилась более чем на 50%. Кроме того, упоминаются многие новые результаты, которые были достигнуты в теории алгоритмов после выхода первого издания.

Также внесены перечисленные ниже изменения.

- В главе, посвященной решению рекуррентных соотношений, больше не содержится информация об итеративном методе. Вместо этого в разделе 4.2 на первый план выступают деревья рекурсии, сами по себе представляющие определенный метод. Авторы пришли к выводу, что использование деревьев рекурсии в меньшей мере связано с риском допустить ошибку. Однако следует заметить, что деревья лучше всего использовать для генерации предположений, которые затем проверяются методом подстановок.
- Несколько по-иному излагается метод разбиения, который применялся для быстрой сортировки (раздел 7.1), и алгоритм порядковой статистики, математическое ожидание времени работы которого выражается линейной функцией (раздел 9.2). В данном издании используется метод, разработанный Ломуто (Lomuto), который наряду с индикаторными случайными величинами позволяет несколько упростить анализ. Предложенный Хоаром (Hoare) метод, который содержался в первом издании, включен в качестве задачи в главе 7.
- Внесены модификации в обсуждение универсального хеширования в разделе 11.3.3, после чего оно интегрируется в представление идеального хеширования.
- Значительно упрощен анализ высоты бинарного дерева поиска, построенного в разделе 12.4 случайным образом.
- Существенно расширено обсуждение элементов динамического программирования (раздел 15.3) и жадных алгоритмов (раздел 16.2). Исследование задачи о выборе задания, которым начинается глава о жадных алгоритмах, помогает прояснить взаимосвязь между динамическим программированием и жадными алгоритмами.
- Доказательство времени работы для объединения непересекающихся множеств заменено в разделе 21.4 доказательством, в котором точные границы определяются с помощью метода потенциалов.
- Значительно упрощено доказательство корректности алгоритма из раздела 22.5, предназначенного для сильно связных компонентов.

- Реорганизована глава 24, посвященная задаче о кратчайших путях из одной вершины: доказательства существенных свойств в ней вынесены в отдельный раздел.
- В разделе 34.5 расширен обзор NP-полноты, приводятся новые доказательства NP-полноты задачи поиска гамильтонова цикла и задачи о сумме подмножества.

Наконец, почти каждый раздел подредактирован, в результате чего в них внесены исправления, упрощения, а также даны более четкие объяснения и доказательства.

Web-узел

Еще одно изменение по сравнению с первым изданием данной книги заключается в том, что теперь книга имеет свой Web-узел: `http://mitpress.mit.edu/algorithms/`. С помощью этого Web-узла можно сообщить об ошибках, получить список известных ошибок или внести предложения; авторы будут рады узнать мнение читателей. Особенно приветствуются идеи по поводу новых задач и упражнений, однако их следует предлагать вместе с решениями.

Авторы выражают сожаление, что не могут лично ответить на все замечания.

Благодарности к первому изданию

Большой вклад в эту книгу внесли многие друзья и коллеги, что способствовало повышению ее качества. Выражаем всем благодарность за помощь и конструктивные замечания.

Лаборатория вычислительной техники Массачуссетского технологического института предоставила идеальные рабочие условия. Особенно большую поддержку оказали наши коллеги из группы Теории вычислений, входящей в состав лаборатории. Особую благодарность мы выражаем Баруху Авербаху (Baruch Awerbuch), Шафи Гольдвассеру (Shafi Goldwasser), Лео Гибасу (Leo Guibas), Тому Лайтону (Tom Leighton), Альберту Мейеру (Albert Meyer), Дэвиду Шмойсу (David Shmoys) и Эве Тардош (Eva Tardos). Благодарим Вильяма Энга (William Ang), Салли Бемус (Sally Bemus), Рея Хиршфелда (Ray Hirschfeld) и Марка Рейнгольда (Mark Reinhold) за поддержку в рабочем состоянии наших машин (DEC Microvaxes, Apple Macintoshes и Sun Sparcstations), а также за перекомпилирование текста в формате TEX каждый раз, когда мы превышали наш лимит времени, выделенного на компиляцию. Компания Machines Corporation предоставила частичную поддержку Чарльзу Лейзерсону (Charles Leiserson), позволившую ему работать над этой книгой во время отпуска в Массачуссетском технологическом институте.

Многие наши коллеги опробовали черновые варианты этого учебника, читая по ним курсы лекций на других факультетах. Они предложили многочисленные

исправления и изменения. Особую благодарность хотелось бы выразить Ричарду Биглю (Richard Beigel), Эндрю Гольдбергу (Andrew Goldberg), Джоан Лукас (Joan Lucas), Марку Овермарсу (Mark Overmars), Алану Шерману (Alan Sherman) и Дайан Сувейн (Diane Souvaine).

Многие ассистенты преподавателей, читающих курсы лекций по учебнику, внесли значительный вклад в разработку материала. Особенно авторы благодарны Алану Баратцу (Alan Baratz), Бонни Бергер (Bonnie Berger), Эдити Дагат (Aditi Dhagat), Берту Калиски (Burt Kaliski), Артуру Ленту (Arthur Lent), Эндрю Моултону (Andrew Moulton), Мариосу Папаефсимиу (Marios Papaefthymiou), Синди Филлипс (Cindy Phillips), Марку Рейнгольду (Mark Reinhold), Филу Рогевей (Phil Rogaway), Флавио Роузу (Flavio Rose), Эйри Рудичу (Arie Rudich), Алану Шерману (Alan Sherman), Клиффу Штайну (Cliff Stein), Сасмите Сюр (Susmita Sur), Грегори Трокселу (Gregory Troxel) и Маргарет Таттл (Margaret Tuttle).

Ценную техническую помощь оказали многие частные лица. Дениз Серджент (Denise Sergent) провела много часов в лабораториях Массачуссетского технологического института, исследуя библиографические ссылки. Мария Синсейл (Maria Sensale), библиотекарь из нашего читального зала, была всегда приветливой и готовой оказать помощь. Много времени, отведенного для работы над литературой при подготовке заключительных замечаний к главам, удалось сэкономить благодаря предоставлению доступа к личной библиотеке Альберта Мейера (Albert Meyer). Шломо Кипнис (Shlomo Kipnis), Билл Нихаус (Bill Niehaus) и Дэвид Вилсон (David Wilson) откорректировали старые упражнения, разработали новые и написали примечания к их решениям. Мариос Папаефсимиу (Marios Papaefthymiou) и Грегори Троксель (Gregory Troxel) внесли вклад в составление предметного указателя. Неоценимую поддержку этого проекта в течение многих лет оказывали наши секретари Инна Радзиховски (Inna Radzihovsky), Дениз Серджент (Denise Sergent), Гейли Шерман (Gayle Sherman) и особенно Би Блекбурн (Be Blackburn), за что авторы им благодарны.

Многие ошибки в ранних черновых вариантах книги были обнаружены студентами. Особую благодарность за внимательное прочтение мы хотели бы выразить Бобби Блюмоуфу (Bobby Blumofe), Бонни Эйзенберг (Bonnie Eisenberg), Раймонду Джонсону (Raymond Johnson), Джону Кину (John Keen), Ричарду Летину (Richard Lethin), Марку Лиллибриджу (Mark Lillibridge), Джону Пезарису (John Pezaris), Стиву Понзио (Steve Ponzio) и Маргарет Таттл (Margaret Tuttle).

Авторы благодарны коллегам, прорецензировавшим отдельные главы или предоставившим информацию по отдельным алгоритмам. Особая благодарность выражается Биллу Айелло (Bill Aiello), Алоку Аггарвалю (Alok Aggarwal), Эрику Баху (Eric Bach), Вашеку Чваталу (Vasek Chvatal), Ричарду Коулу (Richard Cole), Джоан Хастад (Johan Hastad), Алексу Исии (Alex Ishii), Дэвиду Джонсону (David Johnson), Джо Килиану (Joe Kilian), Дайне Кравец (Dina Kravets), Брюсу Маггсу (Bruce Maggs), Джиму Орлину (Jim Orlin), Джеймсу Парку (James Park), Тейну

Пламбеку (Thane Plambeck), Гершелю Сейферу (Hershel Safer), Джеффу Шаллиту (Jeff Shallit), Клиффу Штайну (Cliff Stein), Гилу Стренгу (Gil Strang), Бобу Таржану (Bob Tarjan) и Паулю Вонгу (Paul Wang). Несколько наших коллег также любезно предоставили нам задачи; особенно мы благодарны Эндрю Гольдбергу (Andrew Goldberg), Дэнни Слитору (Danny Sleator) и Умешу Вазирани (Umesh Vazirani).

При написании учебника авторам было приятно работать с издательствами The MIT Press и McGraw-Hill. Особую благодарность за моральную поддержку, помощь и терпение они выражают сотрудникам издательства The MIT Press Фрэнку Сэтлоу (Frank Satlow), Терри Элингу (Terry Ehling), Лари Кохену (Larry Cohen) и Лори Лиджину (Lorrie Lejeune), а также Дэвиду Шапиро (David Shapiro) из издательства McGraw-Hil. Наша особая благодарность — Лари Кохену (Larry Cohen) за выдающуюся работу по корректированию.

Благодарности ко второму изданию

Когда мы предложили Джули Сассман (Julie Sussman, P.P.A.) быть техническим редактором второго издания, то даже не представляли себе, насколько удачный договор заключаем. В дополнение к техническому редактированию, Джули с энтузиазмом взялась за редактирование нашего текста. Стыдно вспомнить, как много ошибок Джули обнаружила в ранних черновиках, хотя это не удивительно с учетом того, сколько их она нашла в первом издании (к сожалению, после его публикации). Более того, Джули пожертвовала своим собственным рабочим графиком, чтобы согласовать его с нашим, — она даже взяла с собой главы из нашей книги в поездку на Виргинские острова! Джули, мы никогда не сможем отблагодарить вас в достаточной мере.

Во время работы над вторым изданием книги авторы работали в отделе вычислительной техники в Дартмутском колледже и лаборатории вычислительной техники в Массачуссетском технологическом институте. Оба учебных заведения послужили стимулирующей рабочей средой, и авторы выражают благодарность своим коллегам.

Друзья и коллеги со всего мира вносили предложения и высказывали свое мнение, что помогло направить усилия в нужное русло. Большое спасибо хотелось бы сказать Сандживу Апоре (Sanjeev Arora), Джавиду Асламу (Javed Aslam), Гаю Блеллоку (Guy Blelloch), Авриму Блюму (Avrim Blum), Скоту Дрисдейлу (Scot Drysdale), Хэйни Фариду (Hany Farid), Халу Габову (Hal Gabow), Эндрю Гольдбергу (Andrew Goldberg), Дэвиду Джонсону (David Johnson), Янлинь Лю (Yanlin Liu), Николасу Шабанелю (Nicolas Schabanel), Александеру Шрайверу (Alexander Schrijver), Саше Шену (Sasha Shen), Дэвиду Шмойсу (David Shmoys), Дену Шпильману (Dan Spielman), Джеральду Джею Сассману (Gerald Jay Sussman), Бо-

бу Таржану (Bob Tarjan), Миккелю Торупу (Mikkel Thorup) и Виджею Вазирани (Vijay Vazirani).

Многие преподаватели и коллеги охотно делились с нами своими знаниями по теории алгоритмов. Особую признательность авторы выражают Джону Л. Бентли (Jon L. Bentley), Бобу Флойду (Bob Floyd), Дону Кнуту (Don Knuth), Гарольду Куну (Harold Kuhn), Кунгу (H. T. Kung), Ричарду Липтону (Richard Lipton), Арнольду Россу (Arnold Ross), Ларри Снайдеру (Larry Snyder), Майклу Шамосу (Michael I. Shamos), Дэвиду Шмойсу (David Shmoys), Кену Штайглицу (Ken Steiglitz), Тому Шимански (Tom Szymanski), Эве Тардош (Eva Tardos), Бобу Таржану (Bob Tarjan) и Джеффри Ульману (Jeffrey Ullman).

Авторы признательны за работу многим ассистентам, читающим курсы по теории алгоритмов в Массачуссетском технологическом институте и колледже г. Дартмут, включая Джозефа Адлера (Joseph Adler), Крэйга Баррака (Craig Barrack), Бобби Блюмоуфа (Bobby Blumofe), Роберто де Приско (Roberto De Prisco), Маттео Фриго (Matteo Frigo), Игала Гальперина (Igal Galperin), Дэвида Гупту (David Gupta), Рея Д. Айера (Raj D. Iyer), Небила Каэйла (Nabil Kahale), Сарфраза Хуршида (Sarfraz Khurshid), Ставроса Коллиопулоса (Stavros Kolliopoulos), Алена Лебланка (Alain Leblanc), Юана Ма (Yuan Ma), Марию Минкофф (Maria Minkoff), Димитриса Митсураса (Dimitris Mitsouras), Алина Попеску (Alin Popescu), Геральда Прокопа (Harald Prokop), Судипту Сенгупту (Sudipta Sengupta), Донну Слоним (Donna Slonim), Джошуа А. Таубера (Joshua A. Tauber), Сивана Толедо (Sivan Toledo), Элишеву Вернер-Рейсс (Elisheva Werner-Reiss), Ли Витти (Lea Wittie), Кван Ву (Qiang Wu) и Майкла Женга (Michael Zhang).

Компьютерная поддержка была предоставлена Вильямом Энгом (William Ang), Скоттом Бломквистом (Scott Blomquist) и Грэгом Шомо (Greg Shomo) из Массачуссетского технологического института, а также Вейном Криппсом (Wayne Cripps), Джоном Конклом (John Konkle) и Тимом Трегубовым (Tim Tregubov) из Дартмута. Авторы также благодарят Би Блекбурна (Be Blackburn), Дона Дейли (Don Dailey), Ли Дикона (Leigh Deacon), Ирен Сибида (Irene Sebeda) и Шерил Паттон Ву (Cheryl Patton Wu) из Массачуссетского технологического института и Филлис Беллмор (Phyllis Bellmore), Келли Кларк (Kelly Clark), Делию Мауцели (Delia Mauceli), Семми Тревиса (Sammie Travis), Деба Уитинга (Deb Whiting) и Бет Юнг (Beth Young) из Дартмута за административную поддержку. Регулярную помощь также оказывали Майкл Фромбергер (Michael Fromberger), Брайан Кемпбелл (Brian Campbell), Аманда Эвбенкс (Amanda Eubanks), Санг Хун Ким (Sung Hoon Kim) и Неха Нарула (Neha Narula) из Дартмута.

Многие читатели любезно сообщили об ошибках в первом издании. Авторы благодарят перечисленных ниже людей, каждый из которых первым сообщил о той или иной ошибке: Лен Адлеман (Len Adleman), Селим Экл (Selim Akl), Ричард Андерсон (Richard Anderson), Жуан Андрейд-Цетто (Juan Andrade-Cetto), Грегори Бачелис (Gregory Bachelis), Дэвид Баррингтон (David Barrington), Пол Бим

(Paul Beame), Ричард Бигль (Richard Beigel), Маргрит Бетке (Margrit Betke), Алекс Блейкмор (Alex Blakemore), Бобби Блюмоуф (Bobby Blumofe), Александр Браун (Alexander Brown), Ксавье Кэйзин (Xavier Cazin), Джек Чен (Jack Chan), Ричард Ченг (Richard Chang), Чинхуа Чен (Chienhua Chen), Йен Ченг (Ien Cheng), Хун Чой (Hoon Choi), Дрю Коулс (Drue Coles), Кристиан Коллберг (Christian Collberg), Джордж Коллинз (George Collins), Эрик Конрад (Eric Conrad), Питер Цазар (Peter Csaszar), Пол Дитц (Paul Dietz), Мартин Дитцфельбингер (Martin Dietzfelbinger), Скот Дрисдейл (Scot Drysdale), Патриция Илай (Patricia Ealy), Яков Эйзенберг (Yaakov Eisenberg), Майкл Эрнст (Michael Ernst), Майкл Форменн (Michael Formann), Недим Фреско (Nedim Fresko), Хал Габов (Hal Gabow), Марек Галецки (Marek Galecki), Игал Гальперин (Igal Galperin), Луиза Гаргано (Luisa Gargano), Джон Гейтли (John Gately), Розарио Дженарио (Rosario Genario), Майэли Гереб (Mihaly Gereb), Рональд Гринберг (Ronald Greenberg), Джерри Гроссман (Jerry Grossman), Стефен Гуттери (Stephen Guattery), Александр Хартемик (Alexander Hartemik), Энтони Хилл (Anthony Hill), Томас Гофмейстер (Thomas Hofmeister), Мэтью Хостеттер (Mathew Hostetter), Юй-Чун Ху (Yih-Chun Hu), Дик Джонсонбаух (Dick Johnsonbaugh), Марцин Юрдзинки (Marcin Jurdzinki), Набил Каэйл (Nabil Kahale), Фумияки Камайя (Fumiaki Kamiya), Ананд Канагала (Anand Kanagala), Марк Кентровиц (Mark Kantrowitz), Скотт Карлин (Scott Karlin), Дин Келли (Dean Kelley), Сэнджей Канна (Sanjay Khanna), Халук Конук (Haluk Konuk), Дайна Кравец (Dina Kravets), Джон Кроугер (Jon Kroger), Бредли Казмаул (Bradley Kuszmaul), Тим Ламберт (Tim Lambert), Хенг Лау (Hang Lau), Томас Ленгауэр (Thomas Lengauer), Джордж Мадрид (George Madrid), Брюс Маггс (Bruce Maggs), Виктор Миллер (Victor Miller), Джозеф Мускат (Joseph Muskat), Танг Нгуйен (Tung Nguyen), Михаил Орлов (Michael Orlov), Джеймс Парк (James Park), Сионбин Парк (Seongbin Park), Иоаннис Паскалидис (Ioannis Paschalidis), Боз Патт-Шамир (Boaz Patt-Shamir), Леонид Пешкин (Leonid Peshkin), Патрицио Поблит (Patricio Poblete), Ира Пол (Ira Pohl), Стивен Понцио (Stephen Ponzio), Кьелл Пост (Kjell Post), Тодд Пойнор (Todd Poynor), Колин Препскиус (Colin Prepscius), Шолом Розен (Sholom Rosen), Дейл Рассел (Dale Russell), Гершель Сейфер (Hershel Safer), Карен Зайдель (Karen Seidel), Джоэль Сейфирес (Joel Seiferas), Эрик Селиджман (Erik Seligman), Стэнли Селков (Stanley Selkow), Джеффри Шаллит (Jeffrey Shallit), Грэг Шеннон (Greg Shannon), Мика Шарир (Micha Sharir), Саша Шен (Sasha Shen), Норман Шульман (Norman Shulman), Эндрю Зингер (Andrew Singer), Дэниел Слитор (Daniel Sleator), Боб Слоан (Bob Sloan), Майкл Софка (Michael Sofka), Фолькер Струмпен (Volker Strumpen), Лон Саншайн (Lon Sunshine), Джули Сассман (Julie Sussman), Астерио Танака (Asterio Tanaka), Кларк Томборсон (Clark Thomborson), Нильс Томмесен (Nils Thommesen), Гомер Тильтон (Homer Tilton), Мартин Томпа (Martin Tompa), Андрей Тум (Andrei Toom), Фельзер Торстен (Felzer Torsten), Хайенду Вэйшнав (Hirendu Vaishnav), М. Вельдхорст (M. Veldhorst), Люка Венути (Luca Venuti), Джайн Вонг (Jian Wang), Майкл Веллман (Michael Wellman),

Джерри Винер (Gerry Wiener), Рональд Вильямс (Ronald Williams), Дэвид Вольф (David Wolfe), Джефф Вонг (Jeff Wong), Ричард Ваунди (Richard Woundy), Нил Юнг (Neal Young), Гайан Ю (Huaiyuan Yu), Тайан Юксинг (Tian Yuxing), Джо Зачари (Joe Zachary), Стив Жанг (Steve Zhang), Флориан Щоук (Florian Zschoke) и Ури Цвик (Uri Zwick).

Многие наши коллеги написали подробные обзоры или заполнили длинные бланки опросов. Авторы благодарят Нэнси Амато (Nancy Amato), Джима Аспнеса (Jim Aspnes), Кевина Комптона (Kevin Compton), Вильяма Иванса (William Evans), Питера Гекса (Peter Gacs), Майкла Гольдвассера (Michael Goldwasser), Андржея Проскуровски (Andrzej Proskurowski), Виджея Рамачандрана (Vijaya Ramachandran) и Джона Райфа (John Reif). Мы также благодарим тех, кто участвовал в опросе и вернул его результаты. Это Джеймс Абелло (James Abello), Джош Бинело (Josh Benaloh), Брайан Бересфорд-Смит (Bryan Beresford-Smith), Кеннет Бла (Kenneth Blaha), Ганс Бодлаендер (Hans Bodlaender), Ричард Бори (Richard Borie), Тед Браун (Ted Brown), Доменико Кантоун (Domenico Cantone), М. Чен (M. Chen), Роберт Цимиковски (Robert Cimikowski), Вильям Клоксин (William Clocksin), Пауль Калл (Paul Cull), Рик Декер (Rick Decker), Мэтью Дикерсон (Matthew Dickerson), Роберт Дуглас (Robert Douglas), Маргарет Флек (Margaret Fleck), Майкл Гудрич (Michael Goodrich), Сюзанн Гамбруш (Susanne Hambrusch), Дин Гендрикс (Dean Hendrix), Ричард Джонсонбау (Richard Johnsonbaugh), Кирякос Калоркоти (Kyriakos Kalorkoti), Шринивас Канканахалли (Srinivas Kankanahalli), Хикио Кох (Hikyoo Koh), Стивен Линделль (Steven Lindell), Еррол Ллойд (Errol Lloyd), Энди Лопес (Andy Lopez), Дайан Рей Лопес (Dian Rae Lopez), Джордж Лакер (George Lucker), Дэвид Мейер (David Maier), Чарльз Мартель (Charles Martel), Ксайнонг Менг (Xiannong Meng), Дэвид Маунт (David Mount), Альберто Поликрити (Alberto Policriti), Анджей Проскуровски (Andrzej Proskurowski), Кирк Прус (Kirk Pruhs), Ив Робер (Yves Robert), Гуна Сизараман (Guna Seetharaman), Стэнли Селков (Stanley Selkow), Роберт Слоан (Robert Sloan), Чарльз Стил (Charles Steele), Джерад Тель (Gerard Tel), Мурали Варанаси (Murali Varanasi), Бернд Уолтер (Bernd Walter) и Альдин Райт (Alden Wright). Авторы хотели бы учесть все внесенные предложения. Единственная проблема заключается в том, что если бы это было сделано, объем второго издания мог бы превысить 3000 страниц!

Второе издание готовилось к публикации в LaTeX 2$_\varepsilon$. Майкл Даунс (Michael Downes) преобразовал LaTeX-сценарии из "классической" версии LaTeX в LaTeX 2$_\varepsilon$ и изменил текстовые файлы таким образом, чтобы в них использовались эти сценарии. Помощь в составлении текста в формате LaTeX также оказал Дэвид Джонс (David Jones). Рисунки ко второму изданию созданы авторами в программе MacDraw Pro. Как и в первом издании, предметный указатель составлен с помощью Windex, — программы на C, написанной авторами, а библиография подготовлена с помощью BibTeX. Айоркор Миллз-Титти (Ayorkor Mills-Tettey) и Роб Лизерн

(Rob Leathern) помогли преобразовать рисунки в MacDraw Pro, а Айоркор также проверил библиографию.

Как и во время работы над первым изданием, сотрудничество с издательствами MIT Press и McGraw-Hill принесло авторам большое удовольствие. Редакторы Боб Прайор (Bob Prior) из MIT Press и Бетси Джонс (Betsy Jones) были терпеливы к нашим причудам и помогли преодолеть все препятствия.

Наконец, мы выражаем благодарность нашим женам Николь Кормен (Nicole Cormen), Гейл Ривест (Gail Rivest) и Ребекке Иври (Rebecca Ivry), нашим детям Рикки (Ricky), Вильяму (William) и Дебби Лейзерсон (Debby Leiserson), Алексу (Alex) и Кристоферу Ривест (Christopher Rivest), а также Молли (Molly), Ноа (Noah) и Бенджамену Штайн (Benjamin Stein), нашим родителям Рени (Renee) и Перри (Perry) Корменам, Джин (Jean) и Марку (Mark) Лейзерсон, Ширли (Shirley) и Ллойду (Lloyd) Ривестам, а также Ирен (Irene) и Ире Штайн (Ira Stein) за любовь и поддержку во время написания этой книги. Этот проект стал возможным благодаря поддержке и поощрению членов наших семей. С любовью посвящаем им эту книгу.

Томас Кормен (Thomas H. Cormen), Ганновер, Нью-Гемпшир
Чарльз Лейзерсон (Charles E. Leiserson), Кембридж, Массачуссетс
Рональд Ривест (Ronald L. Rivest), Кембридж, Массачуссетс
Клиффорд Штайн (Clifford Stein), Ганновер, Нью-Гемпшир

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Информация для писем

из России: 127055, Москва, ул. Лесная, д. 43, стр.1

из Украины: 03150, Киев, а/я 152

ЧАСТЬ I

Основы

Введение

Эта часть книги заставит вас задуматься о вопросах, связанных с разработкой и анализом алгоритмов. Она была запланирована как вводный курс, в котором рассматриваются способы определения алгоритмов, некоторые стратегии их разработки, использующиеся в этой книге, а также применяемые при анализе алгоритмов различные основополагающие идеи. Кратко рассмотрим содержание глав первой части.

В главе 1 рассматривается понятие алгоритма и его роль в современных вычислительных системах. В ней приводится определение алгоритма и даются некоторые примеры. Здесь также обосновывается положение о том, что алгоритмы — это такой же технологический продукт, как, например, аппаратное обеспечение, графические интерфейсы пользователя, объектно-ориентированные системы или сети.

В главе 2 читатель получит возможность ознакомиться с алгоритмами, с помощью которых решается задача о сортировке последовательности из n чисел. Эти алгоритмы сформулированы в виде псевдокода. Несмотря на то, что используемый псевдокод напрямую не преобразуется ни в один из общепринятых языков программирования, он вполне адекватно передает структуру алгоритма, поэтому достаточно компетентный программист легко сможет реализовать его на любом языке программирования. Для изучения выбраны алгоритм сортировки вставкой, в котором используется инкрементный подход, и алгоритм сортировки слиянием, который характеризуется применением рекурсивного метода, известного также как метод "разделяй и властвуй" (метод разбиения). В обоих алгоритмах время выполнения возрастает с ростом количества сортируемых элементов, однако скорость этого роста зависит от выбранного алгоритма. В этой главе будет определено время работы изучаемых алгоритмов; кроме того, вы познакомитесь со специальными обозначениями для выражения времени работы алгоритмов.

В главе 3 дается точное определение обозначений, введенных в главе 2 (которые называются асимптотическими обозначениями). В начале главы 3 определяется несколько асимптотических обозначений для оценки времени работы алгоритма сверху и/или снизу. Остальные разделы главы в основном посвящены описанию математических обозначений. Их предназначение не столько в том, чтобы ознакомить читателя с новыми математическими концепциями, сколько в том, чтобы он смог убедиться, что используемые им обозначения совпадают с принятыми в данной книге.

В главе 4 представлено дальнейшее развитие метода "разделяй и властвуй", введенного в главе 2. В частности, в главе 4 представлены методы решения рекуррентных соотношений, с помощью которых описывается время работы рекурсивных алгоритмов. Бóльшая часть главы посвящена обоснованию "метода контроля" (master method), который используется для решения рекуррентных соотношений,

возникающих в алгоритмах разбиения. Заметим, что изучение доказательств можно опустить без ущерба для понимания материала книги.

Глава 5 служит введением в анализ вероятностей и рандомизированные алгоритмы (т.е. такие, которые основаны на использовании случайных чисел). Анализ вероятностей обычно применяется для определения времени работы алгоритма в тех случаях, когда оно может изменяться для различных наборов входных параметров, несмотря на то, что эти наборы содержат одно и то же количество параметров. В некоторых случаях можно предположить, что распределение входных величин описывается некоторым известным законом распределения вероятностей, а значит, время работы алгоритма можно усреднить по всем возможным наборам входных параметров. В других случаях распределение возникает не из-за входных значений, а в результате случайного выбора, который делается во время работы алгоритма. Алгоритм, поведение которого определяется не только входными значениями, но и величинами, полученными с помощью генератора случайных чисел, называется рандомизированным алгоритмом.

В приложениях A–B содержится дополнительный математический материал, который окажется полезным в процессе чтения книги. Скорее всего, вы уже знакомы с основной частью материала, содержащегося в приложениях (хотя некоторые из встречавшихся вам ранее обозначений иногда могут отличаться от тех, что приняты в данной книге). Поэтому к приложениям следует относиться как к справочному материалу. С другой стороны, не исключено, что вы еще незнакомы с большинством вопросов, рассматриваемых в части I.

ГЛАВА 1

Роль алгоритмов в вычислениях

Что такое алгоритмы? Стоит ли тратить время на их изучение? Какова роль алгоритмов и как они соотносятся с другими компьютерными технологиями? Эта глава дает ответы на поставленные вопросы.

1.1 Алгоритмы

Говоря неформально, ***алгоритм*** (algorithm) — это любая корректно определенная вычислительная процедура, на ***вход*** (input) которой подается некоторая величина или набор величин, и результатом выполнения которой является ***выходная*** (output) величина или набор значений. Таким образом, алгоритм представляет собой последовательность вычислительных шагов, преобразующих входные величины в выходные.

Алгоритм также можно рассматривать как инструмент, предназначенный для решения корректно поставленной ***вычислительной задачи*** (computational problem). В постановке задачи в общих чертах задаются отношения между входом и выходом. В алгоритме описывается конкретная вычислительная процедура, с помощью которой удается добиться выполнения указанных отношений.

Например, может понадобиться выполнить сортировку последовательности чисел в неубывающем порядке. Эта задача часто возникает на практике и служит благодатной почвой для ознакомления на ее примере со многими стандартными методами разработки и анализа алгоритмов. ***Задача сортировки*** (sorting problem) формально определяется следующим образом.

Вход: последовательность из n чисел $\langle a_1, a_2, \ldots, a_n \rangle$.

Выход: перестановка (изменение порядка) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ входной последовательности таким образом, что для ее членов выполняется соотношение $a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$.

Например, если на вход подается последовательность $\langle 31, 41, 59, 26, 41, 58 \rangle$, то вывод алгоритма сортировки должен быть таким: $\langle 26, 31, 41, 41, 58, 59 \rangle$. Подобная входная последовательность называется ***экземпляром*** (instance) задачи сортировки. Вообще говоря, ***экземпляр задачи*** состоит из ввода (удовлетворяющего всем ограничениям, наложенным при постановке задачи), необходимого для решения задачи.

В информатике сортировка является основополагающей операцией (во многих программах она используется в качестве промежуточного шага), в результате чего появилось большое количество хороших алгоритмов сортировки. Выбор наиболее подходящего алгоритма зависит от многих факторов, в том числе от количества сортируемых элементов, от их порядка во входной последовательности, от возможных ограничений, накладываемых на члены последовательности, а также от того, какое устройство используется для хранения последовательности: основная память, магнитные диски или накопители на магнитных лентах.

Говорят, что алгоритм ***корректен*** (correct), если для каждого ввода результатом его работы является корректный вывод. Мы говорим, что корректный алгоритм ***решает*** данную вычислительную задачу. Если алгоритм некорректный, то для некоторых вводов он может вообще не завершить свою работу или выдать ответ, отличный от ожидаемого. Правда, некорректные алгоритмы иногда могут оказаться полезными, если в них есть возможность контролировать частоту возникновения ошибок. Такой пример рассматривается в главе 31, в которой изучаются алгоритмы определения простых чисел, намного превышающих единицу. Тем не менее, обычно мы заинтересованы только в корректных алгоритмах.

Алгоритм может быть задан на естественном языке, в виде компьютерной программы или даже воплощен в аппаратном обеспечении. Единственное требование — его спецификация должна предоставлять точное описание вычислительной процедуры, которую требуется выполнить.

Какие задачи решаются с помощью алгоритмов?

Вычислительные задачи, для которых разработаны алгоритмы, отнюдь не ограничиваются сортировкой. (Возможно, об их разнообразии можно судить по объему данной книги.) Практическое применение алгоритмов чрезвычайно широко, о чем свидетельствуют приведенные ниже примеры.

- Целью проекта по расшифровке генома человека является идентификация всех 100 000 генов, входящих в состав ДНК человека, определение последо-

вательностей, образуемых 3 миллиардами базовых пар, из которых состоит ДНК, сортировка этой информации в базах данных и разработка инструментов для ее анализа. Для реализации всех перечисленных этапов нужны сложные алгоритмы. Решение разнообразных задач, являющихся составными частями данного проекта, выходит за рамки настоящей книги, однако идеи, описанные во многих ее главах, используются для решения упомянутых биологических проблем. Это позволяет ученым достигать поставленных целей, эффективно используя вычислительные ресурсы. При этом экономится время (как машинное, так и затрачиваемое сотрудниками) и деньги, а также повышается эффективность использования лабораторного оборудования.

- Internet позволяет пользователям в любой точке мира быстро получать доступ к информации и извлекать ее в больших объемах. Управление и манипуляция этими данными осуществляется с помощью хитроумных алгоритмов. В число задач, которые необходимо решить, входит определение оптимальных маршрутов, по которым перемещаются данные (методы для решения этой задачи описываются в главе 24), и быстрый поиск страниц, на которых находится та или иная информация, с помощью специализированных поисковых машин (соответствующие методы приводятся в главах 11 и 32).
- Электронная коммерция позволяет заключать сделки и предоставлять товары и услуги с помощью электронных технических средств. Для того чтобы она получила широкое распространение, важно иметь возможность защищать такую информацию, как номера кредитных карт, пароли и банковские счета. В число базовых технологий в этой области входят криптография с открытым ключом и цифровые подписи (они описываются в главе 31), основанные на численных алгоритмах и теории чисел.
- В производстве и коммерции очень важно распорядиться ограниченными ресурсами так, чтобы получить максимальную выгоду. Нефтяной компании может понадобиться информация о том, где пробурить скважины, чтобы получить от них как можно более высокую прибыль. Кандидат в президенты может задаться вопросом, как потратить деньги, чтобы максимально повысить свои шансы победить на выборах. Авиакомпаниям важно знать, какую минимальную цену можно назначить за билеты на тот или иной рейс, чтобы уменьшить количество свободных мест и не нарушить при этом законы, регулирующие авиаперевозку пассажиров. Поставщик услуг Internet должен уметь так размещать дополнительные ресурсы, чтобы повышался уровень обслуживания клиентов. Все эти задачи можно решить с помощью линейного программирования, к изучению которого мы приступим в главе 29.

Несмотря на то, что некоторые детали представленных примеров выходят за рамки настоящей книги, в ней приводятся основные методы, применяющиеся для

их решения. В книге также показано, как решить многие конкретные задачи, в том числе те, что перечислены ниже.

- Пусть имеется карта дорог, на которой обозначены расстояния между каждой парой соседних перекрестков. Наша цель — определить кратчайший путь от одного перекрестка к другому. Количество возможных маршрутов может быть огромным, даже если исключить те из них, которые содержат самопересечения. Как найти наиболее короткий из всех возможных маршрутов? При решении этой задачи карта дорог (которая сама по себе является моделью настоящих дорог) моделируется в виде графа (мы подробнее познакомимся с графами в главе 10 и приложении Б). Задача будет заключаться в определении кратчайшего пути от одной вершины графа к другой. Эффективное решение этой задачи представлено в главе 24.
- Пусть дана последовательность $\langle A_1, A_2, \ldots, A_n \rangle$, образованная n матрицами, и нам нужно найти произведение этих матриц. Поскольку матричное произведение обладает свойством ассоциативности, существует несколько корректных порядков умножения. Например, если $n = 4$, перемножение матриц можно выполнять любым из следующих способов (определяемых скобками): $(A_1 (A_2 (A_3 A_4)))$, $(A_1 ((A_2 A_3) A_4))$, $((A_1 A_2) (A_3 A_4))$, $((A_1 (A_2 A_3)) A_4)$ или $(((A_1 A_2) A_3) A_4)$. Если все эти матрицы квадратные (т.е. одинакового размера), порядок перемножения не влияет на продолжительность процедуры. Если же матрицы различаются по размеру (но при этом их размеры соответствуют правилам матричного умножения), то порядок их перемножения может очень сильно влиять на время выполнения процедуры. Количество всевозможных вариантов, различающихся порядком перемножения матриц, растет с увеличением количества матриц по экспоненциальному закону, поэтому для перебора всех этих вариантов может потребоваться довольно длительное время. Из главы 15 мы узнаем, как эта задача намного эффективнее решается с помощью общего метода, известного как динамическое программирование.
- Пусть имеется уравнение $ax \equiv b \pmod{n}$, где a, b и n — целые числа, и нужно найти все целые x по модулю n, удовлетворяющие этому уравнению. Оно может не иметь решения, может иметь одно или несколько решений. Можно попытаться применить метод, заключающийся в последовательной подстановке в уравнение чисел $x = 0, 1, \ldots, n-1$, но в главе 31 представлен более эффективный метод.
- Пусть имеется n принадлежащих плоскости точек, и нужно найти выпуклую оболочку этих точек. Выпуклой оболочкой точек называется минимальный выпуклый многоугольник, содержащий эти точки. Для решения этой задачи удобно воспользоваться такой наглядной картиной: если представить точки в виде вбитых в доску и торчащих из нее гвоздей, то выпуклую оболочку

можно получить, намотав на них резинку таким образом, чтобы все гвозди вошли внутрь замкнутой линии, образуемой резинкой. Каждый гвоздь, вокруг которого обвивается резинка, становится вершиной выпуклой оболочки (рис. 33.6). В качестве набора вершин может выступать любое из 2^n подмножеств множества точек. Однако недостаточно знать, какие точки являются вершинами выпуклой оболочки, нужно еще указать порядок их обхода. Таким образом, чтобы найти выпуклую оболочку, приходится перебрать множество вариантов. В главе 33 описаны два хороших метода определения выпуклой оболочки.

Приведенные выше случаи применения алгоритмов отнюдь не являются исчерпывающими, однако на их примере выявляются две общие характеристики, присущие многим интересным алгоритмам.

1. Есть множество вариантов-кандидатов, большинство из которых решениями не являются. Поиск среди них нужной комбинации является довольно трудоемким.
2. Задачи имеют практическое применение. Простейший пример среди перечисленных задач — определение кратчайшего пути, соединяющего два перекрестка. Любая компания, занимающаяся авто- или железнодорожными перевозками, финансово заинтересована в том, чтобы определить кратчайший маршрут. Это способствовало бы снижению затрат труда и расходов горючего. Маршрутизатору Internet также нужно иметь возможность находить кратчайшие пути в сети, чтобы как можно быстрее доставлять сообщения.

Структуры данных

В книге представлен ряд структур данных. ***Структура данных*** (data structure) — это способ хранения и организации данных, облегчающий доступ к этим данным и их модификацию. Ни одна структура данных не является универсальной и не может подходить для всех целей, поэтому важно знать преимущества и ограничения, присущие некоторым из них.

Методические указания

Данную книгу можно рассматривать как “сборник рецептов” для алгоритмов. Правда, однажды вам встретится задача, для которой вы не сможете найти опубликованный алгоритм (например, таковы многие из приведенных в книге упражнений и задач). Данная книга научит вас методам разработки алгоритмов и их анализа. Это позволит вам разрабатывать корректные алгоритмы и оценивать их эффективность самостоятельно.

Сложные задачи

Большая часть этой книги посвящена эффективным алгоритмам. Обычной мерой эффективности для нас является скорость и время, в течение которого алгоритм выдает результат. Однако существуют задачи, для которых неизвестны эффективные методы их решения. В главе 34 рассматривается интересный вопрос, имеющий отношение к подобным задачам, известным как NP-полные.

Почему же NP-полные задачи представляют интерес? Во-первых, несмотря на то, что до сих пор не найден эффективный алгоритм их решения, также не доказано, что такого алгоритма не существует. Другими словами, неизвестно, существует ли эффективный алгоритм для NP-полных задач. Во-вторых, набор NP-полных задач обладает замечательным свойством. Оно заключается в том, что если эффективный алгоритм существует хотя бы для одной из этих задач, то его можно сформулировать и для всех остальных. Эта взаимосвязь между NP-полными задачами и отсутствие методов их эффективного решения вызывает еще бóльший интерес к ним. В третьих, некоторые NP-полные задачи похожи (но не идентичны) на задачи, для которых известны эффективные алгоритмы. Небольшое изменение формулировки задачи может значительно ухудшить эффективность самого лучшего из всех известных алгоритмов.

Полезно располагать знаниями о NP-полных задачах, поскольку в реальных приложениях некоторые из них возникают неожиданно часто. Если перед вами встанет задача найти эффективный алгоритм для NP-полной задачи, скорее всего, вы потратите много времени на безрезультатные поиски. Если же вы покажете, что данная задача принадлежит к разряду NP-полных, то можно будет вместо самого лучшего из всех возможных решений попробовать найти достаточно эффективное.

Рассмотрим конкретный пример. Компания грузового автотранспорта имеет один центральный склад. Каждый день грузовики загружаются на этом складе и отправляются по определенному маршруту, доставляя груз в несколько мест. В конце рабочего дня грузовик должен вернуться на склад, чтобы на следующее утро его снова можно было загрузить. Чтобы сократить расходы, компании нужно выбрать оптимальный порядок доставки груза в различные точки. При этом расстояние, пройденное грузовиком, должно быть минимальным. Эта задача хорошо известна как “задача о коммивояжере”, и она является NP-полной. Эффективный алгоритм решения для нее неизвестен, однако при некоторых предположениях можно указать такие алгоритмы, в результате выполнения которых полученное расстояние будет ненамного превышать минимально возможное. Подобные “приближенные алгоритмы” рассматриваются в главе 35.

Упражнения

1.1-1. Приведите реальные примеры задач, в которых возникает одна из таких вычислительных задач: сортировка, определение оптимального порядка перемножения матриц, поиск выпуклой оболочки.

1.1-2. Какими еще параметрами, кроме скорости, можно характеризовать алгоритм на практике?

1.1-3. Выберите одну из встречавшихся вам ранее структур данных и опишите ее преимущества и ограничения.

1.1-4. Что общего между задачей об определении кратчайшего пути и задачей о коммивояжере? Чем они различаются?

1.1-5. Сформулируйте задачу, в которой необходимо только наилучшее решение. Сформулируйте также задачу, в которой может быть приемлемым решение, достаточно близкое к наилучшему.

1.2 Алгоритмы как технология

Предположим, быстродействие компьютера и объем его памяти можно увеличивать до бесконечности. Была бы в таком случае необходимость в изучении алгоритмов? Была бы, но только для того, чтобы продемонстрировать, что метод решения имеет конечное время работы и что он дает правильный ответ.

Если бы компьютеры были неограниченно быстрыми, подошел бы любой корректный метод решения задачи. Возможно, вы бы предпочли, чтобы реализация решения была выдержана в хороших традициях программирования (т.е. качественно разработана и аккуратно занесена в документацию), но чаще всего выбирался бы метод, который легче всего реализовать.

Конечно же, сегодня есть весьма производительные компьютеры, но их быстродействие не может быть бесконечно большим. Память тоже дешевеет, но она не может быть бесплатной. Таким образом, время вычисления — это такой же ограниченный ресурс, как и объем необходимой памяти. Этими ресурсами следует распоряжаться разумно, чему и способствует применение алгоритмов, эффективных в плане расходов времени и памяти.

Эффективность

Алгоритмы, разработанные для решения одной и то же задачи, часто очень сильно различаются по эффективности. Эти различия могут быть намного значительнее, чем те, что вызваны применением неодинакового аппаратного и программного обеспечения.

В качестве примера можно привести два алгоритма сортировки, которые рассматриваются в главе 2. Для выполнения первого из них, известного как ***сортировка вставкой***, требуется время, которое оценивается как $c_1 n^2$, где n — количество сортируемых элементов, а c_1 — константа, не зависящая от n. Таким образом, время работы этого алгоритма приблизительно пропорционально n^2. Для выполнения второго алгоритма, ***сортировки слиянием***, требуется время, приблизительно равное $c_2 n \lg n$, где $\lg n$ — краткая запись $\log_2 n$, а c_2 — некоторая другая константа, не зависящая от n. Обычно константа метода вставок меньше константы метода слияния, т.е. $c_1 < c_2$. Мы убедимся, что постоянные множители намного меньше влияют на время работы алгоритма, чем множители, зависящие от n. Для двух приведенных методов последние относятся как $\lg n$ к n. Для небольшого количества сортируемых элементов сортировка включением обычно работает быстрее, однако когда n становится достаточно большим, все заметнее проявляется преимущество сортировки слиянием, возникающее благодаря тому, что для больших n незначительная величина $\lg n$ по сравнению с n полностью компенсирует разницу величин постоянных множителей. Не имеет значения, во сколько раз константа c_1 меньше, чем c_2. С ростом количества сортируемых элементов обязательно будет достигнут переломный момент, когда сортировка слиянием окажется более производительной.

В качестве примера рассмотрим два компьютера — А и Б. Компьютер А более быстрый, и на нем работает алгоритм сортировки вставкой, а компьютер Б более медленный, и на нем работает алгоритм сортировки методом слияния. Оба компьютера должны выполнить сортировку множества, состоящего из миллиона чисел. Предположим, что компьютер А выполняет миллиард инструкций в секунду, а компьютер Б — лишь десять миллионов. Таким образом, компьютер А работает в 100 раз быстрее, чем компьютер Б. Чтобы различие стало еще бóльшим, предположим, что код для метода вставок (т.е. для компьютера А) написан самым лучшим в мире программистом с использованием команд процессора, и для сортировки n чисел надо выполнить $2n^2$ команд (т.е. $c_1 = 2$). Сортировка же методом слияния (на компьютере Б) реализована программистом-середнячком с помощью языка высокого уровня. При этом компилятор оказался не слишком эффективным, и в результате получился код, требующий выполнения $50 n \lg n$ команд (т.е. $c_2 = 50$). Для сортировки миллиона чисел компьютеру А понадобится

$$\frac{2 \cdot \left(10^6\right)^2 \text{команд}}{10^9 \text{команд / с}} = 2000 \text{ с},$$

а компьютеру Б —

$$\frac{50 \cdot 10^6 \cdot \lg 10^6 \text{команд}}{10^7 \text{команд / с}} \approx 100 \text{ с}.$$

Как видите, использование кода, время работы которого возрастает медленнее, даже при плохом компиляторе на более медленном компьютере требует на по-

рядок меньше процессорного времени! Если же нужно выполнить сортировку 10 миллионов чисел, то преимущество метода слияния становится еще более очевидным: если для сортировки вставкой потребуется приблизительно 2.3 дня, то для сортировки слиянием — меньше 20 минут. Общее правило таково: чем больше количество сортируемых элементов, тем заметнее преимущество сортировки слиянием.

Алгоритмы и другие технологии

Приведенный выше пример демонстрирует, что алгоритмы, как и аппаратное обеспечение компьютера, представляют собой ***технологию***. Общая производительность системы настолько же зависит от эффективности алгоритма, как и от мощности применяющегося аппаратного обеспечения. В области разработки алгоритмов происходит такое же быстрое развитие, как и в других компьютерных технологиях.

Возникает вопрос, действительно ли так важны алгоритмы, работающие на современных компьютерах, если и так достигнуты выдающиеся успехи в других областях высоких технологий, таких как:

- аппаратное обеспечение с высокими тактовыми частотами, конвейерной обработкой и суперскалярной архитектурой;
- легкодоступные, интуитивно понятные графические интерфейсы (GUI);
- объектно-ориентированные системы;
- локальные и глобальные сети.

Ответ — да, безусловно. Несмотря на то, что иногда встречаются приложения, которые не требуют алгоритмического наполнения (например, некоторые простые Web-приложения), для большинства приложений определенное алгоритмическое наполнение необходимо. Например, рассмотрим Web-службу, определяющую, как добраться из одного места в другое (во время написания книги существовало несколько таких служб). В основе ее реализации лежит высокопроизводительное аппаратное обеспечение, графический интерфейс пользователя, глобальная сеть и, возможно, объектно-ориентированный подход. Кроме того, для определенных операций, выполняемых данной Web-службой, необходимо использование алгоритмов — например, таких как вычисление квадратных корней (что может потребоваться для определения кратчайшего пути), визуализации карт и интерполяции адресов.

Более того, даже приложение, не требующее алгоритмического наполнения на высоком уровне, сильно зависит от алгоритмов. Ведь известно, что работа приложения зависит от производительности аппаратного обеспечения, а при его разработке применяются разнообразные алгоритмы. Все мы также знаем, что

приложение тесно связано с графическим интерфейсом пользователя, а для разработки любого графического интерфейса пользователя требуются алгоритмы. Вспомним приложения, работающие в сети. Чтобы они могли функционировать, необходимо осуществлять маршрутизацию, которая, как уже говорилось, основана на ряде алгоритмов. Чаще всего приложения составляются на языке, отличном от машинного. Их код обрабатывается компилятором или интерпретатором, которые интенсивно используют различные алгоритмы. И таким примерам нет числа.

Кроме того, ввиду постоянного роста вычислительных возможностей компьютеров, они применяются для решения все более сложных задач. Как мы уже убедились на примере сравнительного анализа двух методов сортировки, с ростом сложности решаемой задачи различия в эффективности алгоритмов проявляются все значительнее.

Знание основных алгоритмов и методов их разработки — одна из характеристик, отличающих умелого программиста от новичка. Располагая современными компьютерными технологиями, некоторые задачи можно решить и без основательного знания алгоритмов, однако знания в этой области позволяют достичь намного большего.

Упражнения

1.2-1. Приведите пример приложения, для которого необходимо алгоритмическое наполнение на уровне приложений, и дайте характеристику функций, входящих в состав этих алгоритмов.

1.2-2. Предположим, на одной и той же машине проводится сравнительный анализ реализаций двух алгоритмов сортировки, работающих по методу вставок и по методу слияния. Для сортировки n элементов методом вставок необходимо $8n^2$ шагов, а для сортировки методом слияния — $64n \lg n$ шагов. При каком значении n время сортировки методом вставок превысит время сортировки методом слияния?

1.2-3. При каком минимальном значении n алгоритм, время работы которого определяется формулой $100n^2$, работает быстрее, чем алгоритм, время работы которого выражается как 2^n, если оба алгоритма выполняются на одной и той же машине?

Задачи

1-1. **Сравнение времени работы алгоритмов**

Ниже приведена таблица, строки которой соответствуют различным функциям $f(n)$, а столбцы — значениям времени t. Определите максимальные значения n, для которых задача может быть решена за время t, если

предполагается, что время работы алгоритма, необходимое для решения задачи, равно $f(n)$ микросекунд.

	1 секунда	1 минута	1 час	1 день	1 месяц	1 год	1 век
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

Заключительные замечания

Имеется множество учебников, посвященных общим вопросам, которые имеют отношение к алгоритмам. К ним относятся книги Ахо (Aho), Хопкрофта (Hopcroft) и Ульмана (Ullman) [5,6], Бейза (Baase) и Ван Гельдера (Van Gelder) [26], Брассарда (Brassard) и Бретли (Bratley) [46,47], Гудрича (Goodrich) и Тамазии (Tamassia) [128], Горовица (Horowitz), Сани (Sahni) и Раджисекарана (Rajasekaran) [158], Кингстона (Kingston) [179], Кнута (Knuth) [182,183,185], Козена (Kozen) [193], Манбера (Manber) [210], Мельхорна (Mehlhorn) [217,218,219], Пурдома (Purdom) и Брауна (Brown) [252], Рейнгольда (Reingold), Ньевергельта (Nievergelt) и Део (Deo) [257], Седжевика (Sedgewick) [269], Скьены (Skiena) [280] и Вильфа (Wilf) [315]. Некоторые аспекты разработки алгоритмов, имеющие большую практическую направленность, обсуждаются в книгах Бентли (Bentley) [39,40] и Гоннета (Gonnet) [126]. Обзоры по алгоритмам можно также найти в книгах *Handbook of Theoretical Computer Science, Volume A* [302] и *CRC Handbook on Algorithms and Theory of Computation* [24]. Обзоры алгоритмов, применяющихся в вычислительной биологии, можно найти в учебниках Гасфилда (Gusfield) [136], Певзнера (Pevzner) [240], Сетубала (Setubal) и Мейдениса (Meidanis) [272], а также Вотермена (Waterman) [309].

ГЛАВА 2

Приступаем к изучению

В этой главе мы ознакомимся с основными понятиями, с помощью которых на протяжении всей книги будет проводиться разработка и анализ алгоритмов.

В начале главы исследуется алгоритм сортировки вставками. Он предназначен для решения задачи сортировки, поставленной в главе 1. Для формулирования алгоритмов мы используем псевдокод (который должен быть понятен читателям, имеющим опыт программирования). Для каждого алгоритма обосновывается его корректность и анализируется время работы. В ходе анализа указанного алгоритма вводятся обозначения, используемые для указания зависимости времени работы алгоритма от количества сортируемых элементов. После обсуждения сортировки вставками описывается метод декомпозиции, основанный на принципе "разделяй и властвуй". Этот подход используется для разработки различных алгоритмов; в данном случае с его помощью будет сформулирован алгоритм, получивший название сортировки слиянием. В конце главы анализируется время работы этого алгоритма сортировки.

2.1 Сортировка вставкой

Наш первый алгоритм, алгоритм сортировки методом вставок, предназначен для решения ***задачи сортировки*** (sorting problem), сформулированной в главе 1. Приведем еще раз эту формулировку.

Вход: последовательность из n чисел $\langle a_1, a_2, \ldots, a_n \rangle$.

Выход: перестановка (изменение порядка) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ входной последовательности таким образом, что для ее членов выполняется соотношение $a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$.

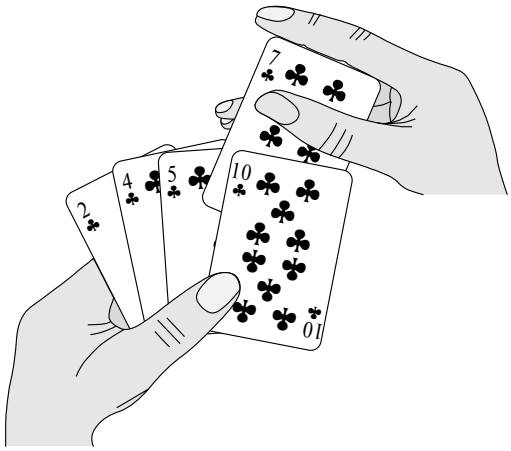


Рис. 2.1. Сортировка карт методом вставок

Сортируемые числа также известны под названием ***ключи*** (keys).

В этой книге алгоритмы обычно описываются в виде ***псевдокода***, который во многих отношениях похож на языки программирования C, Pascal и Java. Тем, кто знаком хотя бы с одним из этих языков, не потребуется много усилий на чтение алгоритмов. От обычного кода псевдокод отличается тем, что он выразителен, а также лаконично, точно и понятно описывает алгоритм. Иногда в качестве такового выступает литературный язык, поэтому не удивляйтесь, если в инструкции "настоящего" кода встретите обычную фразу или предложение. Другое различие между псевдокодом и обычным кодом заключается в том, что в псевдокоде, как правило, не рассматриваются некоторые вопросы, которые приходится решать разработчикам программного обеспечения. Такие вопросы, как абстракция данных, модульность и обработка ошибок часто игнорируются, чтобы более выразительно передать суть, заложенную в алгоритме.

Наше изучение алгоритмов начинается с рассмотрения ***сортировки вставкой*** (insertion sort). Этот алгоритм эффективно работает при сортировке небольшого количества элементов. Сортировка вставкой напоминает способ, к которому прибегают игроки для сортировки имеющихся на руках карт. Пусть вначале в левой руке нет ни одной карты, и все они лежат на столе рубашкой вверх. Далее со стола берется по одной карте, каждая из которых помещается в нужное место среди карт, которые находятся в левой руке. Чтобы определить, куда нужно поместить очередную карту, ее масть и достоинство сравнивается с мастью и достоинством карт в руке. Допустим, сравнение проводится в направлении слева направо (рис. 2.1). В любой момент времени карты в левой руке будут рассортированы, и это будут те карты, которые первоначально лежали в стопке на столе.

Псевдокод сортировки методом вставок представлен ниже под названием INSERTION_SORT. На его вход подается массив $A[1..n]$, содержащий последовательность из n сортируемых чисел (количество элементов массива A обозначено в этом коде как $length[A]$.) Входные числа ***сортируются без использования дополнительной памяти***: их перестановка производится в пределах массива, и объем используемой при этом дополнительной памяти не превышает некоторую постоянную величину. По окончании работы алгоритма INSERTION_SORT входной массив содержит отсортированную последовательность:

```
INSERTION_SORT(A)
1  for j ← 2 to length[A]
2      do key ← A[j]
3         ▷ Вставка элемента A[j] в отсортированную
          ▷ последовательность A[1..j − 1]
4         i ← j − 1
5         while i > 0 и A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8         A[i + 1] ← key
```

Инварианты цикла и корректность сортировки вставкой

На рис. 2.2 продемонстрировано, как этот алгоритм работает для массива $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Элементы массива обозначены квадратиками, над которыми находятся индексы, а внутри — значения соответствующих элементов. Части *а–д* этого рисунка соответствуют итерациям цикла **for** в строках 1–8 псевдокода. В каждой итерации черный квадратик содержит значение ключа, которое сравнивается со значениями серых квадратиков, расположенных слева от него (строка псевдокода 5). Серыми стрелками указаны те значения массива, которые сдвигаются на одну позицию вправо (строка 6), а черной стрелкой — перемещение ключа (строка 8). В части *е* показано конечное состояние сортируемого массива.

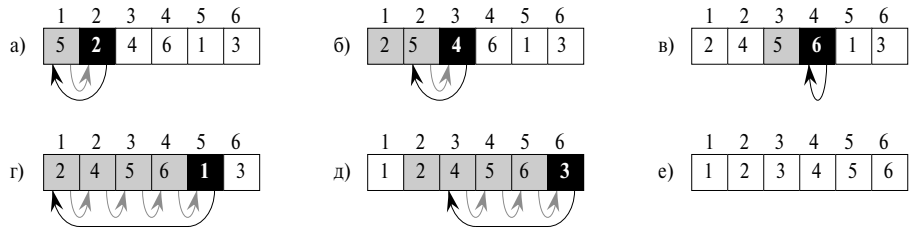


Рис. 2.2. Выполнение алгоритма INSERTION_SORT над массивом $A = \langle 5, 2, 4, 6, 1, 3 \rangle$

Индекс j указывает "текущую карту", которая помещается в руку. В начале каждой итерации внешнего цикла **for** с индексом j массив A состоит из двух частей. Элементы $A[1..j-1]$ соответствуют отсортированным картам в руке, а элементы $A[j+1..n]$ — стопке карт, которые пока что остались на столе. Заметим, что элементы $A[1..j-1]$ *изначально* тоже находились на позициях от 1 до $j-1$, но в другом порядке, а теперь они отсортированы. Назовем это свойство элементов $A[1..j-1]$ ***инвариантом цикла*** (loop invariant) и сформулируем его еще раз.

> В начале каждой итерации цикла **for** из строк 1–8 подмассив $A[1..j-1]$ содержит те же элементы, которые были в нем с самого начала, но расположенные в отсортированном порядке.

Инварианты цикла позволяют понять, корректно ли работает алгоритм. Необходимо показать, что инварианты циклов обладают следующими тремя свойствами.

Инициализация. Они справедливы перед первой инициализацией цикла.

Сохранение. Если они истинны перед очередной итерацией цикла, то остаются истинны и после нее.

Завершение. По завершении цикла инварианты позволяют убедиться в правильности алгоритма.

Если выполняются первые два свойства, инварианты цикла остаются истинными перед каждой очередной итерацией цикла. Обратите внимание на сходство с математической индукцией, когда для доказательства определенного свойства для всех элементов упорядоченной последовательности нужно доказать его справедливость для начального элемента этой последовательности, а затем обосновать шаг индукции. В данном случае первой части доказательства соответствует обоснование того, что инвариант цикла выполняется перед первой итерацией, а второй части — доказательство того, что инвариант цикла выполняется после очередной итерации (шаг индукции).

Для наших целей третье свойство самое важное, так как нам нужно с помощью инварианта цикла продемонстрировать корректность алгоритма. Оно также отличает рассматриваемый нами метод от обычной математической индукции, в которой шаг индукции используется в бесконечных последовательностях. В данном случае по окончании цикла "индукция" завершается.

Рассмотрим, соблюдаются ли эти свойства для сортировки методом включений.

Инициализация. Начнем с того, что покажем справедливость инварианта цикла перед первой итерацией, т.е. при $j = 2$[1]. Таким образом, подмножество элементов $A[1..j-1]$ состоит только из одного элемента $A[1]$, сохраняющего

[1] Если рассматривается цикл **for**, момент времени, когда проверяется справедливость инварианта цикла перед первой итерацией, наступает сразу после начального присваивания значения индексу

исходное значение. Более того, в этом подмножестве элементы рассортированы (тривиальное утверждение). Все вышесказанное подтверждает, что инвариант цикла соблюдается перед первой итерацией цикла.

Сохранение. Далее, обоснуем второе свойство: покажем, что инвариант цикла сохраняется после каждой итерации. Выражаясь неформально, можно сказать, что в теле внешнего цикла **for** происходит сдвиг элементов $A[j-1]$, $A[j-2]$, $A[j-3]$, ... на одну позицию вправо до тех пор, пока не освободится подходящее место для элемента $A[j]$ (строки 4–7), куда он и помещается (строка 8). При более формальном подходе к рассмотрению второго свойства потребовалось бы сформулировать и обосновать инвариант для внутреннего цикла **while**. Однако на данном этапе мы предпочитаем не вдаваться в такие формальные подробности, поэтому будем довольствоваться неформальным анализом, чтобы показать, что во внешнем цикле соблюдается второе свойство.

Завершение. Наконец, посмотрим, что происходит по завершении работы цикла. При сортировке методом включений внешний цикл **for** завершается, когда j превышает n, т.е. когда $j = n + 1$. Подставив в формулировку инварианта цикла значение $n + 1$, получим такое утверждение: в подмножестве элементов $A[1..n]$ находятся те же элементы, которые были в нем до начала работы алгоритма, но расположенные в отсортированном порядке. Однако подмножество $A[1..n]$ и есть сам массив A! Таким образом, весь массив отсортирован, что и подтверждает корректность алгоритма.

Метод инвариантов циклов будет применяться далее в данной главе, а также в последующих главах книги.

Соглашения, принятые при составлении псевдокода

При составлении псевдокода использовались следующие соглашения.

1. Структура блоков указывается с помощью отступов. Например, тело цикла **for**, начало которого — в строке 1, образовано строками 2–8, а тело цикла **while**, который начинается в строке 5, состоит из строк 6 и 7, а строка 8 в него не входит. Стиль отступов имеет силу также в инструкциях **if-then-else**. Использование отступов вместо общепринятых указателей блочной структуры, таких как инструкции **begin** и **end**, значительно уменьшает громоздкость. При этом псевдокод не становится менее понятным, а даже наоборот[2].

цикла, но перед первой проверкой в заголовочной инструкции цикла. В листинге INSERTION_SORT это момент, когда переменной j присвоено значение 2, но еще не выполнена проверка неравенства $j \leqslant length[A]$.

[2] В реальных языках программирования, вообще говоря, не рекомендуется применять отступы для того, чтобы подчеркнуть структуру блоков, поскольку, если код разбит на несколько страниц, то сложно определить уровень вложенности блоков.

2. Инструкции циклов **while**, **for** и **repeat**, а также общепринятая конструкция **if-then-else** интерпретируются аналогично одноименным инструкциям языка Pascal[3]. Что касается цикла **for**, то здесь есть одно тонкое различие: в языке Pascal после выхода из цикла значение переменной, выступающей в роли счетчика цикла, становится неопределенным, а в данной книге счетчик сохраняет свое значение после выхода из цикла. Таким образом, сразу после завершения цикла **for** значение счетчика равно значению, на единицу превышающему верхнюю границу цикла. Это свойство уже использовалось для доказательства корректности алгоритма сортировки, работающего по методу вставок. Заголовок цикла **for**, заданный в строке 1, имеет вид **for** $j \leftarrow 2$ **to** $length\,[A]$, поэтому по завершении цикла $j = length\,[A] + 1$ (или, что то же самое, $j = n + 1$).
3. Символ "▷" указывают, что оставшаяся часть строки — это комментарий.
4. Множественное присваивание, имеющее вид $i \leftarrow j \leftarrow e$, обозначает, что значение выражения e присваивается обеим переменным i и j; оно аналогично двум последовательным инструкциям присваивания: $j \leftarrow e$ и $i \leftarrow j$.
5. Переменные (такие как i, j и key) являются локальными по отношению к данной процедуре. Глобальные переменные не применяются, если это не указано явным образом.
6. Доступ к элементам массива осуществляется путем указания имени массива, после которого в квадратных скобках следует индекс. Например, $A\,[i]$ — это обозначение i-го элемента массива A. С помощью обозначения ".." указывается интервал значений, которые принимает индекс массива. Таким образом, обозначение $A\,[1..j]$ свидетельствует о том, что данное подмножество массива A состоит из j элементов $A\,[1]$, $A\,[2]$, ..., $A\,[j]$.
7. Сложные данные обычно представляются в виде ***объектов***, состоящих из ***атрибутов*** и ***полей***. Доступ к определенному полю осуществляется с помощью имени поля, после которого в квадратных скобках указывается имя объекта. Например, массив трактуется как объект с атрибутом $length$, указывающим количество его элементов. Чтобы указать количество элементов массива A, нужно написать $length\,[A]$. Несмотря на то, что квадратные скобки используются и для индексирования элементов массива, и в атрибутах объекта, их интерпретация будет понятна из контекста.

   Переменная, которая используется в качестве имени массива или объекта, трактуется как указатель на данные, представляющие этот массив или объект. Для всех полей f объекта x присваивание $y \leftarrow x$ приводит к тому, что $f\,[y] = f\,[x]$. Более того, если теперь выполнить присваивание $f\,[x] \leftarrow 3$,

[3]Инструкции, эквивалентные перечисленным, есть в большинстве языков программирования с блочной структурой, однако их синтаксис может немного отличаться от синтаксиса языка Pascal.

то впоследствии будет выполняться не только соотношение $f[x] = 3$, но и соотношение $f[y] = 3$. Другими словами, после присваивания $y \leftarrow x$ переменные x и y указывают на один и тот же объект.

Иногда указатель вообще не ссылается ни на какой объект. В таком случае он имеет специальное значение NIL.

8. Параметры передаются в процедуру ***по значению***: в вызывающей процедуре создается своя собственная копия параметров, и если в вызванной процедуре какому-то параметру присваивается значение, то в вызывающей процедуре *не* происходит никаких изменений. Если передаются объекты, то происходит копирование указателя на данные, представляющие этот объект, но поля объекта не копируются. Например, если x — параметр вызванной процедуры, то присваивание $x \leftarrow y$, выполненное в этой процедуре, невидимо для вызывающей процедуры. Однако присваивание $f[x] \leftarrow 3$ будет видимым.
9. Логические операторы "и" и "или" вычисляются ***сокращенно*** (short circuiting). Это означает, что при вычислении выражения "x и y" сначала вычисляется значение выражения x. Если оно ложно, то все выражение не может быть истинным, и значение выражения y не вычисляется. Если же выражение x истинно, то для определения значения всего выражения необходимо вычислить выражение y. Аналогично, в выражении "x или y" величина y вычисляется только в том случае, если выражение x ложно. Сокращенные операторы позволяют составлять такие логические выражения, как "$x \neq$ NIL и $f[x] = y$", не беспокоясь о том, что произойдет при попытке вычислить выражение $f[x]$, если $x =$ NIL.

Упражнения

2.1-1. Используя в качестве модели рис. 2.2, проиллюстрируйте работу алгоритма INSERTION_SORT по упорядочению массива $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2. Перепишите процедуру INSERTION_SORT так, чтобы она выполняла сортировку не в невозрастающем, а в неубывающем порядке.

2.1-3. Рассмотрим ***задачу поиска***.

Вход: последовательность n чисел $A = \langle a_1, a_2, \ldots, a_n \rangle$ и величина v.

Выход: индекс i, обладающий свойством $v = A[i]$, или значение NIL, если в массиве A отсутствует значение v.

Составьте псевдокод ***линейного поиска***, при работе которого выполняется сканирование последовательности в поиске значения v. Докажите корректность алгоритма с помощью инварианта цикла. Убедитесь, что

выбранные инварианты цикла удовлетворяют трем необходимым условиям.

2.1-4. Рассмотрим задачу сложения двух двоичных целых чисел длиной n битов каждое, хранящихся в массивах A и B, которые состоят из n элементов. Сумму этих двух чисел необходимо занести в двоичной форме в массив C, состоящий из $n + 1$ элемента. Приведите строгую формулировку задачи и составьте псевдокод для сложения этих двух чисел.

2.2 Анализ алгоритмов

Анализ алгоритма заключается в том, чтобы предсказать требуемые для его выполнения ресурсы. Иногда оценивается потребность в таких ресурсах, как память, пропускная способность сети или необходимое аппаратное обеспечение, но чаще всего определяется время вычисления. Путем анализа нескольких алгоритмов, предназначенных для решения одной и той же задачи, можно без труда выбрать наиболее эффективный. В процессе такого анализа может также оказаться, что несколько алгоритмов примерно равноценны, а все остальные нужно отбросить.

Прежде чем мы научимся анализировать алгоритмы, необходимо разработать технологию, которая будет для этого использоваться. В эту технологию нужно будет включить модель ресурсов и величины их измерения. С учетом того, что алгоритмы реализуются в виде компьютерных программ, в этой книге в большинстве случаев в качестве технологии реализации принята модель обобщенной однопроцессорной машины с ***памятью с произвольным доступом*** (Random-Access Machine — RAM). В этой модели команды процессора выполняются последовательно; одновременно выполняемые операции отсутствуют.

Строго говоря, в модели RAM следует точно определить набор инструкций и время их выполнения, однако это утомительно и мало способствует пониманию принципов разработки и анализа алгоритмов. С другой стороны, нужно соблюдать осторожность, чтобы не исказить модель RAM. Например, что будет, если в RAM встроена команда сортировки? В этом случае сортировку можно выполнять с помощью всего одной команды процессора. Такая модель нереалистична, поскольку настоящие компьютеры не имеют подобных встроенных команд, а мы ориентируемся именно на их устройство. В рассматриваемую модель входят те команды, которые обычно можно найти в реальных компьютерах: арифметические (сложение, вычитание, умножение, деление, вычисление остатка деления, приближение действительного числа ближайшим меньшим или ближайшим большим целым), операции перемещения данных (загрузка, занесение в память, копирование) и управляющие (условное и безусловное ветвление, вызов подпрограммы и воз-

врат из нее). Для выполнения каждой такой инструкции требуется определенный фиксированный промежуток времени.

В модели RAM есть целочисленный тип данных и тип чисел с плавающей точкой. Несмотря на то, что обычно в этой книге точность не рассматривается, в некоторых приложениях она играет важную роль. Также предполагается, что существует верхний предел размера слова данных. Например, если обрабатываются входные данные с максимальным значением n, обычно предполагается, что целые числа представлены $c \lg n$ битами, где c — некоторая константа, не меньшая единицы. Требование $c \geqslant 1$ обусловлено тем, что в каждом слове должно храниться одно из n значений, что позволит индексировать входные элементы. Кроме того, предполагается, что c — это конечная константа, поэтому объем слова не может увеличиваться до бесконечности. (Если бы это было возможно, в одном слове можно было бы хранить данные огромных объемов и осуществлять над ними операции в рамках одной элементарной команды, что нереально.)

В реальных компьютерах содержатся команды, не упомянутые выше, которые представляют "серую область" модели RAM. Например, является ли возведение в степень элементарной командой? В общем случае — нет; чтобы вычислить выражение x^y, в котором x и y — действительные числа, потребуется несколько команд. Однако в некоторых случаях эту операцию можно представить в виде элементарной команды. Во многих компьютерах имеется команда побитового сдвига влево, которая в течение времени, требуемого для выполнения элементарной команды, сдвигает биты целого числа на k позиций влево. В большинстве случаев такой сдвиг целого числа на одну позицию эквивалентен его умножению на 2. Сдвиг битов на k позиций влево эквивалентен его умножению на $2k$. Таким образом, на этих компьютерах $2k$ можно вычислить с помощью одной элементарной инструкции, сдвинув целое число 1 на k позиций влево; при этом k не должно превышать количество битов компьютерного слова. Мы попытаемся избегать таких "серых областей" модели RAM, однако вычисление $2k$ будет рассматриваться как элементарная операция, если k — достаточно малое целое положительное число.

В исследуемой модели RAM не предпринимаются попытки смоделировать иерархию запоминающих устройств, общепринятую на современных компьютерах. Таким образом, кэш и виртуальная память (которая чаще всего реализуется в соответствии с требованиями страничной организации памяти) не моделируется. В некоторых вычислительных моделях предпринимается попытка смоделировать эффекты, вызванные иерархией запоминающих устройств, которые иногда важны в реальных программах, работающих на реальных машинах. В ряде рассмотренных в данной книге задач принимаются во внимание эти эффекты, но в большинстве случаев они не учитываются. Модели, включающие в себя иерархию запоминающих устройств, заметно сложнее модели RAM, поэтому они могут затруднить работу. Кроме того, анализ, основанный на модели RAM, обычно

замечательно предсказывает производительность алгоритмов, выполняющихся на реальных машинах.

Анализ даже простого алгоритма в модели RAM может потребовать значительных усилий. В число необходимых математических инструментов может войти комбинаторика, теория вероятностей, навыки алгебраических преобразований и способность идентифицировать наиболее важные слагаемые в формуле. Поскольку поведение алгоритма может различаться для разных наборов входных значений, потребуется методика учета, описывающая поведение алгоритма с помощью простых и понятных формул.

Даже когда для анализа данного алгоритма выбирается всего одна модель машины, нам все еще предстоит выбрать средства для выражения анализа. Хотелось бы выбрать простые обозначения, которые позволят легко с ними работать и выявлять важные характеристики требований, предъявляемых алгоритмом к ресурсам, а также избегать сложных деталей.

Анализ алгоритма, работающего по методу вставок

Время работы процедуры INSERTION_SORT зависит от набора входных значений: для сортировки тысячи чисел требуется больше времени, чем для сортировки трех чисел. Кроме того, время сортировки с помощью этой процедуры может быть разным для последовательностей, состоящих из одного и того же количества элементов, в зависимости от степени упорядоченности этих последовательностей до начала сортировки. В общем случае время работы алгоритма увеличивается с увеличением количества входных данных, поэтому общепринятая практика — представлять время работы программы как функцию, зависящую от количества входных элементов. Для этого понятия “время работы алгоритма” и “размер входных данных” нужно определить точнее.

Наиболее адекватное понятие ***размера входных данных*** (input size) зависит от рассматриваемой задачи. Во многих задачах, таких как сортировка или дискретные преобразования Фурье, это *количество входных элементов*, например, размер n сортируемого массива. Для многих других задач, таких как перемножение двух целых чисел, наиболее подходящая мера для измерения размера ввода — *общее количество битов*, необходимых для представления входных данных в обычных двоичных обозначениях. Иногда размер ввода удобнее описывать с помощью не одного, а двух чисел. Например, если на вход алгоритма подается граф, размер ввода можно описывать, указывая количество вершин и ребер графа. Для каждой рассматриваемой далее задачи будет указываться способ измерения размера входных данных.

Время работы алгоритма для того или иного ввода измеряется в количестве элементарных операций, или “шагов”, которые необходимо выполнить. Здесь удобно ввести понятие шага, чтобы рассуждения были как можно более машинно-

независимыми. На данном этапе мы будем исходить из точки зрения, согласно которой для выполнения каждой строки псеводкода требуется фиксированное время. Время выполнения различных строк может отличаться, но мы предположим, что одна и та же i-я строка выполняется за время c_i, где c_i — константа. Эта точка зрения согласуется с моделью RAM и отражает особенности практической реализации псевдокода на реальных компьютерах[4].

В последующих рассуждениях формула для выражения времени работы алгоритма INSERTION_SORT, которая сначала будет сложным образом зависеть от всех величин c_i, значительно упростится благодаря более лаконичным обозначениям, с которыми проще работать. Эти более простые обозначения позволят легче определить, какой из двух алгоритмов эффективнее.

Начнем с того, что введем для процедуры INSERTION_SORT время выполнения каждой инструкции и количество их повторений. Для каждого $j = 2, 3, \ldots, n$, где $n = length\,[A]$, обозначим через t_j количество проверок условия в цикле **while** (строка 5). При нормальном завершении циклов **for** или **while** (т.е. когда перестает выполняться условие, заданное в заголовке цикла) условие проверяется на один раз больше, чем выполняется тело цикла. Само собой разумеется, мы считаем, что комментарии не являются исполняемыми инструкциями, поэтому они не увеличивают время работы алгоритма:

INSERTION_SORT(A)		время	количество раз
1	**for** $j \leftarrow 2$ **to** $length[A]$	c_1	n
2	**do** $key \leftarrow A[j]$	c_2	$n-1$
3	$\triangleright$ Вставка элемента $A[j]$ в отсортированную последовательность $A[1..j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

Время работы алгоритма — это сумма промежутков времени, необходимых для выполнения каждой входящей в его состав исполняемой инструкции. Если вы-

[4]Здесь есть некоторые тонкие нюансы. Шаги вычислений, описанные на обычном языке, часто представляют собой разновидности процедур, состоящих из нескольких элементарных инструкций. Например, далее в этой книге может встретиться строка "сортировка точек по координате x". Как вы увидите, эта команда требует больше, чем постоянного количества времени работы. Заметим также, что команда вызова подпрограммы выполняется в течение фиксированного времени, однако сколько длится выполнение вызванной подпрограммы — это зависит от ее сложности. Таким образом, процесс **вызова** подпрограммы (передача в нее параметров и т.п. действия) следует отличать от **выполнения** этой подпрограммы.

полнение инструкции длится в течение времени c_i и она повторяется в алгоритме n раз, то ее вклад в полное время работы алгоритма равно $c_i n$.[5] Чтобы вычислить время работы алгоритма INSERTION_SORT (обозначим его через $T(n)$), нужно просуммировать произведения значений, стоящих в столбцах *время* и *количество раз*, в результате чего получим:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + \\ + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1).$$

Даже если размер входных данных является фиксированной величиной, время работы алгоритма может зависеть от степени упорядоченности сортируемых величин, которой они обладали до ввода. Например, самый благоприятный случай для алгоритма INSERTION_SORT — это когда все элементы массива уже отсортированы. Тогда для каждого $j = 2, 3, \ldots, n$ получается, что $A[i] \leqslant key$ в строке 5, еще когда i равно своему начальному значению $j - 1$. Таким образом, при $j = 2, 3, \ldots, n$ $t_j = 1$, и время работы алгоритма в самом благоприятном случае вычисляется так:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = \\ = (c_1 + c_2 + c_4 + c_5 + c_8)\, n - (c_2 + c_4 + c_5 + c_8).$$

Это время работы можно записать как $an + b$, где a и b — *константы*, зависящие от величин c_i; т.е. это время является ***линейной функцией*** от n.

Если элементы массива отсортированы в порядке, обратном требуемому (в данном случае в порядке убывания), то это наихудший случай. Каждый элемент $A[j]$ необходимо сравнивать со всеми элементами уже отсортированного подмножества $A[1..j-1]$, так что для $j = 2, 3, \ldots, n$ значения $t_j = j$. С учетом того, что

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

и

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

[5] Это правило не всегда применимо к такому ресурсу, как память. Если инструкция оперирует с m словами памяти и выполняется n раз, то это еще не означает, что всего при этом потребляется mn слов памяти.

(как выполняется это суммирование, показано в приложении А), получаем, что время работы алгоритма INSERTION_SORT в худшем случае определяется соотношением

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) = \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - \\ &\quad - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Это время работы можно записать как $an^2 + bn + c$, где константы a, b и c зависят от c_i. Таким образом, это ***квадратичная функция*** от n.

Обычно время работы алгоритма фиксировано для определенных входных данных, как в случае сортировки вставкой, однако в последующих главах мы ознакомимся с некоторыми интересными алгоритмами, поведение которых носит вероятностный характер. Время их работы может меняться даже для одних и тех же входных данных.

Наихудшее и среднее время работы

Анализируя алгоритм, работающий по методу вставок, мы рассматривали как наилучший, так и наихудший случай, когда элементы массива были рассортированы в порядке, обратном требуемому. Далее в этой книге мы будем уделять основное внимание определению только времени работы в ***наихудшем случае***, т.е. максимальном времени работы из *всех* входных данных размера n. Тому есть три причины.

- Время работы алгоритма в наихудшем случае — это верхний предел этой величины для любых входных данных. Располагая этим значением, мы точно знаем, что для выполнения алгоритма не потребуется большее количество времени. Не нужно будет делать каких-то сложных предположений о времени работы и надеяться, что на самом деле эта величина не будет превышена.
- В некоторых алгоритмах наихудший случай встречается достаточно часто. Например, если в базе данных происходит поиск информации, то наихудшему случаю соответствует ситуация, когда нужная информация в базе данных отсутствует. В некоторых поисковых приложениях поиск отсутствующей информации может происходить довольно часто.
- Характер поведения “усредненного” времени работы часто ничем не лучше поведения времени работы для наихудшего случая. Предположим, что последовательность, к которой применяется сортировка методом вставок,

сформирована случайным образом. Сколько времени понадобится, чтобы определить, в какое место подмассива $A[1..j-1]$ следует поместить элемент $A[j]$? В среднем половина элементов подмассива $A[1..j-1]$ меньше, чем $A[j]$, а половина — больше его. Таким образом, в среднем нужно проверить половину элементов подмассива $A[1..j-1]$, поэтому t_j приблизительно равно $j/2$. В результате получается, что среднее время работы алгоритма является квадратичной функцией от количества входных элементов, т.е. характер этой зависимости такой же, как и для времени работы в наихудшем случае.

В некоторых частных случаях нас будет интересовать ***среднее*** время работы алгоритма, или его ***математическое ожидание***[6]. В главе 5 будет продемонстрирован метод ***вероятностного анализа***, позволяющий определить ожидаемое время работы. Однако при анализе усредненного времени работы возникает одна проблема, которая заключается в том, что не всегда очевидно, какие входные данные для данной задачи будут "усредненными". Часто делается предположение, что все наборы входных параметров одного и того же объема встречаются с одинаковой вероятностью. На практике это предположение может не соблюдаться, однако иногда можно применять ***рандомизированные алгоритмы***, в которых используется случайный выбор, и это позволяет провести вероятностный анализ.

Порядок возрастания

Для облегчения анализа процедуры INSERTION_SORT были сделаны некоторые упрощающие предположения. Во-первых, мы проигнорировали фактическое время выполнения каждой инструкции, представив эту величину в виде некоторой константы c_i. Далее мы увидели, что учет всех этих констант дает излишнюю информацию: время работы алгоритма в наихудшем случае выражается формулой $an^2 + bn + c$, где a, b, и c — некоторые константы, зависящие от стоимостей c_i. Таким образом, мы игнорируем не только фактические стоимости команд, но и их абстрактные стоимости c_i.

Теперь введем еще одно абстрактное понятие, упрощающее анализ. Это ***скорость роста*** (rate of growth), или ***порядок роста*** (order of growth), времени работы, который и интересует нас на самом деле. Таким образом, во внимание будет приниматься только главный член формулы (т.е. в нашем случае an^2), поскольку при больших n членами меньшего порядка можно пренебречь. Кроме того, постоянные множители при главном члене также будут игнорироваться, так как для оценки вычислительной эффективности алгоритма с входными данными большого объема они менее важны, чем порядок роста. Таким образом, время работы

[6] Далее в книге строгий термин "математическое ожидание" некоторой величины зачастую для простоты изложения заменяется термином "ожидаемое значение", например, "ожидаемое время работы алгоритма" означает "математическое ожидание времени работы алгоритма". — *Прим. ред.*

алгоритма, работающего по методу вставок, в наихудшем случае равно $\Theta\left(n^2\right)$ (произносится "тета от n в квадрате"). В этой главе Θ-обозначения используются неформально; их строгое определение приводится в главе 3.

Обычно один алгоритм считается эффективнее другого, если время его работы в наихудшем случае имеет более низкий порядок роста. Из-за наличия постоянных множителей и второстепенных членов эта оценка может быть ошибочной, если входные данные невелики. Однако если объем входных данных значительный, то, например, алгоритм $\Theta\left(n^2\right)$ в наихудшем случае работает быстрее, чем алгоритм $\Theta\left(n^3\right)$.

Упражнения

2.2-1. Выразите функцию $n^3/1000 - 100n^2 - 100n + 3$ в Θ-обозначениях.

2.2-2. Рассмотрим сортировку элементов массива A, которая производится так. Сначала определяется наименьший элемент массива A, который ставится на место элемента $A[1]$, затем производится поиск второго наименьшего элемента массива A, который ставится на место элемента $A[2]$. Этот процесс продолжается для первых $n-1$ элементов массива A. Запишите псевдокод этого алгоритма, известного как ***сортировка выбором*** (selection sort). Какой инвариант цикла сохраняется для этого алгоритма? Почему его достаточно выполнить для первых $n-1$ элементов, а не для всех n элементов? Определите время работы алгоритма в наилучшем и наихудшем случаях и запишите его в Θ-обозначениях.

2.2-3. Рассмотрим алгоритм линейного поиска (см. упражнение 2.1-3). Для скольких элементов входной последовательности в среднем нужно произвести проверку, если предполагается, что все элементы массива с равной вероятностью могут иметь искомое значение? Что происходит в наихудшем случае? Чему равно время работы алгоритма линейного поиска в среднем и в наихудшем случае (в Θ-обозначениях)? Обоснуйте ваш ответ.

2.2-4. Каким образом можно модифицировать почти каждый алгоритм, чтобы получить оптимальное время работы в наилучшем случае?

2.3 Разработка алгоритмов

Есть много методов разработки алгоритмов. В алгоритме, работающем по методу вставок, применяется ***инкрементный*** подход: располагая отсортированным подмассивом $A[1..j-1]$, мы помещаем очередной элемент $A[j]$ туда, где он должен находиться, в результате чего получаем отсортированный подмассив $A[1..j]$.

В данном разделе рассматривается альтернативный подход к разработке алгоритмов, известный как метод разбиения (“разделяй и властвуй”). Разработаем с помощью этого подхода алгоритм сортировки, время работы которого в наихудшем случае намного меньше времени работы алгоритма, работающего по методу включений. Одно из преимуществ алгоритмов, разработанных методом разбиения, заключается в том, что время их работы часто легко определяется с помощью технологий, описанных в главе 4.

2.3.1 Метод декомпозиции

Многие полезные алгоритмы имеют ***рекурсивную*** структуру: для решения данной задачи они рекурсивно вызывают сами себя один или несколько раз, чтобы решить вспомогательную задачу, имеющую непосредственное отношение к поставленной задаче. Такие алгоритмы зачастую разрабатываются с помощью метода ***декомпозиции***, или ***разбиения***: сложная задача разбивается на несколько более простых, которые подобны исходной задаче, но имеют меньший объем; далее эти вспомогательные задачи решаются рекурсивным методом, после чего полученные решения комбинируются с целью получить решение исходной задачи.

Парадигма, лежащая в основе метода декомпозиции “разделяй и властвуй”, на каждом уровне рекурсии включает в себя три этапа.

Разделение задачи на несколько подзадач.

Покорение — рекурсивное решение этих подзадач. Когда объем подзадачи достаточно мал, выделенные подзадачи решаются непосредственно.

Комбинирование решения исходной задачи из решений вспомогательных задач.

Алгоритм ***сортировки слиянием*** (merge sort) в большой степени соответствует парадигме метода разбиения. На интуитивном уровне его работу можно описать таким образом.

Разделение: сортируемая последовательность, состоящая из n элементов, разбивается на две меньшие последовательности, каждая из которых содержит $n/2$ элементов.

Покорение: сортировка обеих вспомогательных последовательностей методом слияния.

Комбинирование: слияние двух отсортированных последовательностей для получения окончательного результата.

Рекурсия достигает своего нижнего предела, когда длина сортируемой последовательности становится равной 1. В этом случае вся работа уже сделана, поскольку любую такую последовательность можно считать упорядоченной.

Основная операция, которая производится в процессе сортировки по методу слияний, — это объединение двух отсортированных последовательностей в ходе комбинирования (последний этап). Это делается с помощью вспомогательной процедуры MERGE(A, p, q, r), где A — массив, а p, q и r — индексы, нумерующие элементы массива, такие, что $p \leqslant q < r$. В этой процедуре предполагается, что элементы подмассивов $A[p..q]$ и $A[q+1..r]$ упорядочены. Она ***сливает*** эти два подмассива в один отсортированный, элементы которого заменяют текущие элементы подмассива $A[p..r]$.

Для выполнения процедуры MERGE требуется время $\Theta(n)$, где $n = r - p + 1$ — количество подлежащих слиянию элементов. Процедура работает следующим образом. Возвращаясь к наглядному примеру сортировки карт, предположим, что на столе лежат две стопки карт, обращенных лицевой стороной вниз. Карты в каждой стопке отсортированы, причем наверху находится карта наименьшего достоинства. Эти две стопки нужно объединить в одну выходную, в которой карты будут рассортированы и также будут обращены рубашкой вверх. Основной шаг состоит в том, чтобы из двух младших карт выбрать самую младшую, извлечь ее из соответствующей стопки (при этом в данной стопке верхней откажется новая карта) и поместить в выходную стопку. Этот шаг повторяется до тех пор, пока в одной из входных стопок не кончатся карты, после чего оставшиеся в другой стопке карты нужно поместить в выходную стопку. С вычислительной точки зрения выполнение каждого основного шага занимает одинаковые промежутки времени, так как все сводится к сравнению достоинства двух верхних карт. Поскольку необходимо выполнить, по крайней мере, n основных шагов, время работы процедуры слияния равно $\Theta(n)$.

Описанная идея реализована в представленном ниже псевдокоде, однако в нем также есть дополнительное ухищрение, благодаря которому в ходе каждого основного шага не приходится проверять, является ли каждая из двух стопок пустой. Идея заключается в том, чтобы поместить в самый низ обеих объединяемых колод так называемую ***сигнальную*** карту особого достоинства, что позволяет упростить код. Для обозначения сигнальной карты используется символ ∞. Не существует карт, достоинство которых больше достоинства сигнальной карты. Процесс продолжается до тех пор, пока проверяемые карты в обеих стопках не окажутся сигнальными. Как только это произойдет, это будет означать, что все несигнальные карты уже помещены в выходную стопку. Поскольку заранее известно, что в выходной стопке должно содержаться ровно $r - p + 1$ карта, выполнив такое количество основных шагов, можно остановиться:

MERGE(A, p, q, r)
1 $n_1 \leftarrow q - p + 1$
2 $n_2 \leftarrow r - q$
3 Создаем массивы $L[1..n_1 + 1]$ и $R[1..n_2 + 1]$

```
 4  for i ← 1 to n1
 5      do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7      do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13      do if L[i] ⩽ R[j]
14            then A[k] ← L[i]
15                 i ← i + 1
16            else A[k] ← R[j]
17                 j ← j + 1
```

Подробно опишем работу процедуры MERGE. В строке 1 вычисляется длина n_1 подмассива $A[p..q]$, а в строке 2 — длина n_2 подмассива $A[q+1..r]$. Далее в строке 3 создаются массивы L ("левый" — "left") и R ("правый" — "right"), длины которых равны $n_1 + 1$ и $n_2 + 1$ соответственно. В цикле **for** в строках 4 и 5 подмассив $A[p..q]$ копируется в массив $L[1..n_1]$, а в цикле **for** в строках 6 и 7 подмассив $A[q+1..r]$ копируется в массив $R[1..n_2]$. В строках 8 и 9 последним элементам массивов L и R присваиваются сигнальные значения.

Как показано на рис. 2.3, в результате копирования и добавления сигнальных карт получаем массив L с последовательностью чисел $\langle 2, 4, 5, 7, \infty\rangle$ и массив R с последовательностью чисел $\langle 1, 2, 3, 6, \infty\rangle$. Светло-серые ячейки массива A содержат конечные значения, а светло-серые ячейки массивов L и R — значения, которые еще только должны быть скопированы обратно в массив A. В светло-серых ячейках находятся исходные значения из подмассива $A[9..16]$ вместе с двумя сигнальными картами. В темно-серых ячейках массива A содержатся значения, которые будут заменены другими, а в темно-серых ячейках массивов L и R — значения, уже скопированные обратно в массив A. В частях рисунка *а–з* показано состояние массивов A, L и R, а также соответствующие индексы k, i и j перед каждой итерацией цикла в строках 12–17. В части *и* показано состояние массивов и индексов по завершении работы алгоритма. На данном этапе подмассив $A[9..16]$ отсортирован, а два сигнальных значения в массивах L и R — единственные элементы, оставшиеся в этих массивах и не скопированные в массив A. В строках 10–17, проиллюстрированных на рис. 2.3, выполняется $r - p + 1$ основных шагов, в ходе каждого из которых производятся манипуляции с инвариантом цикла, описанным ниже.

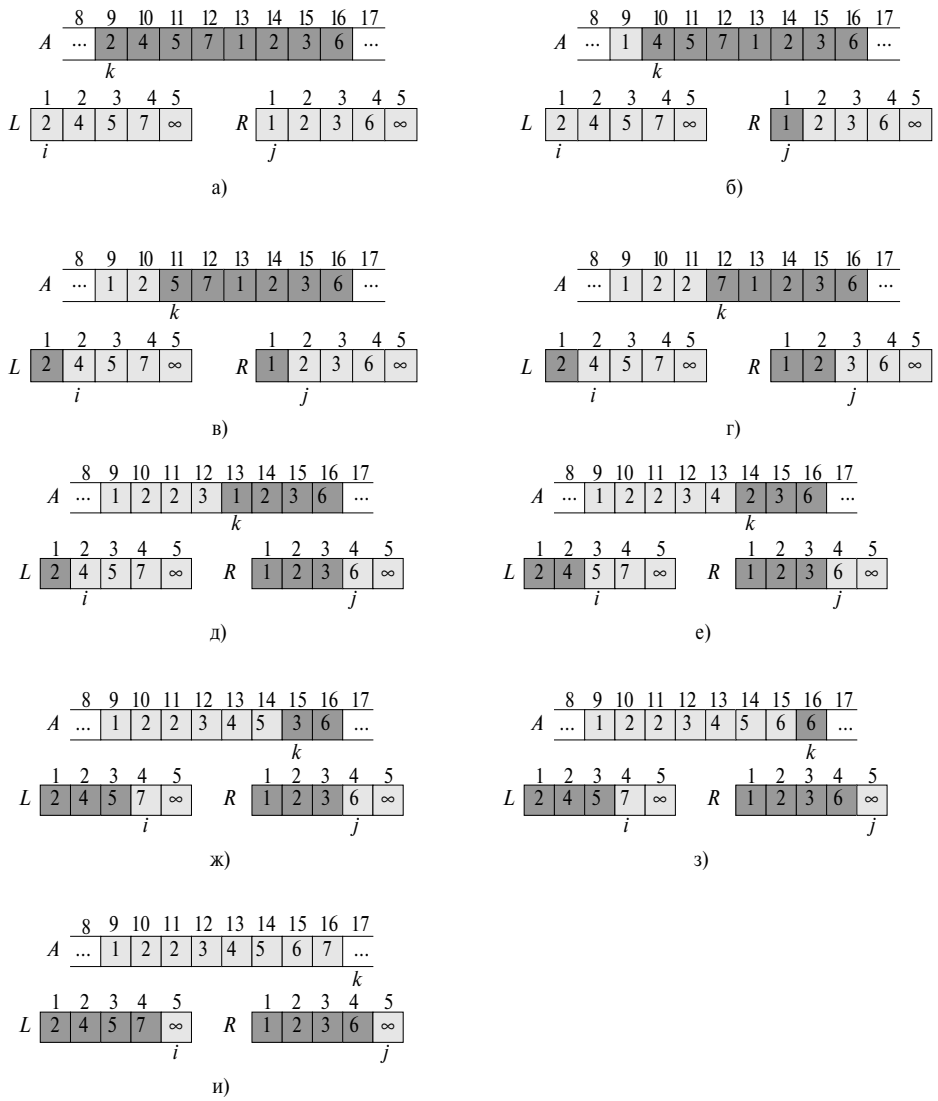


Рис. 2.3. Операции, выполняемые в строках 10–17 процедуры MERGE$(A, 9, 12, 16)$, когда в подмассиве $A\,[9..16]$ содержится последовательность $\langle 2, 4, 5, 7, 1, 2, 3, 6\rangle$

Перед каждой итерацией цикла **for** в строках 12–17, подмассив $A\,[p..k-1]$ содержит $k-p$ наименьших элементов массивов $L\,[1..n_1+1]$ и $R\,[1..n_2+1]$ в отсортированном порядке. Кроме того, элементы $L\,[i]$ и $R\,[i]$ являются наименьшими элементами массивов L и R, которые еще не скопированы в массив A.

Необходимо показать, что этот инвариант цикла соблюдается перед первой итерацией рассматриваемого цикла **for**, что каждая итерация цикла не нарушает его, и что с его помощью удается продемонстрировать корректность алгоритма, когда цикл заканчивает свою работу.

Инициализация. Перед первой итерацией цикла $k = p$, поэтому подмассив $A[p..k-1]$ пуст. Он содержит $k - p = 0$ наименьших элементов массивов L и R. Поскольку $i = j = 1$, элементы $L[i]$ и $R[j]$ — наименьшие элементы массивов L и R, не скопированные обратно в массив A.

Сохранение. Чтобы убедиться, что инвариант цикла сохраняется после каждой итерации, сначала предположим, что $L[i] \leqslant R[j]$. Тогда $L[i]$ — наименьший элемент, не скопированный в массив A. Поскольку в подмассиве $A[p..k-1]$ содержится $k-p$ наименьших элементов, после выполнения строки 14, в которой значение элемента $L[i]$ присваивается элементу $A[k]$, в подмассиве $A[p..k]$ будет содержаться $k - p + 1$ наименьший элемент. В результате увеличения параметра k цикла **for** и значения переменной i (строка 15), инвариант цикла восстанавливается перед следующей итерацией. Если же выполняется неравенство $L[i] > R[j]$, то в строках 16 и 17 выполняются соответствующие действия, в ходе которых также сохраняется инвариант цикла.

Завершение. Алгоритм завершается, когда $k = r + 1$. В соответствии с инвариантом цикла, подмассив $A[p..k-1]$ (т.е. подмассив $A[p..r]$) содержит $k - p = r - p + 1$ наименьших элементов массивов $L[1..n_1 + 1]$ и $R[1..n_2 + 1]$ в отсортированном порядке. Суммарное количество элементов в массивах L и R равно $n_1 + n_2 + 2 = r - p + 3$. Все они, кроме двух самых больших, скопированы обратно в массив A, а два оставшихся элемента являются сигнальными.

Чтобы показать, что время работы процедуры MERGE равно $\Theta(n)$, где $n = r - p + 1$, заметим, что каждая из строк 1–3 и 8–11 выполняется в течение фиксированного времени; длительность циклов **for** в строках 4–7 равна $\Theta(n_1 + n_2) = \Theta(n)$,[7] а в цикле **for** в строках 12–17 выполняется n итераций, на каждую из которых затрачивается фиксированное время.

Теперь процедуру MERGE можно использовать в качестве подпрограммы в алгоритме сортировки слиянием. Процедура MERGE_SORT(A, p, r) выполняет сортировку элементов в подмассиве $A[p..r]$. Если справедливо неравенство $p \geqslant r$, то в этом подмассиве содержится не более одного элемента, и, таким образом, он отсортирован. В противном случае производится разбиение, в ходе которого

[7] В главе 3 будет показано, как формально интерпретируются уравнения с Θ-обозначениями.

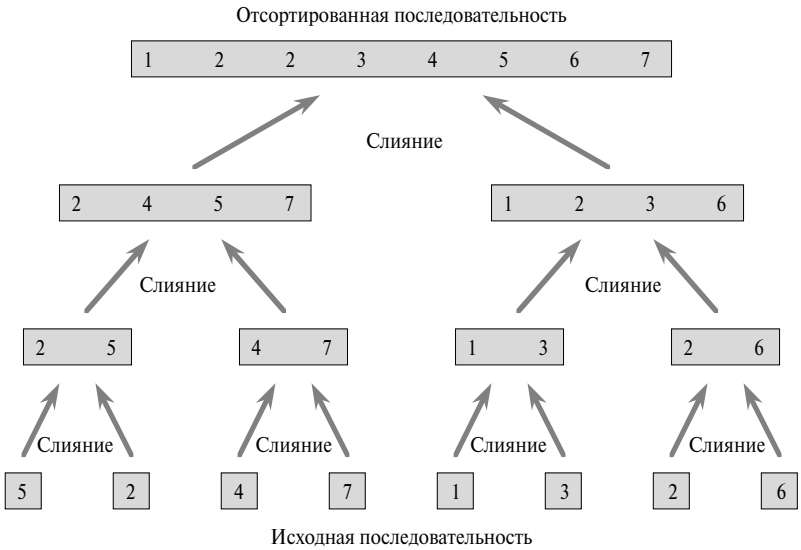


Рис. 2.4. Процесс сортировки массива $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$ методом слияния. Длины подлежащих объединению отсортированных подпоследовательностей возрастают по мере работы алгоритма

вычисляется индекс q, разделяющий массив $A[p..r]$ на два подмассива: $A[p..q]$ с $\lceil n/2 \rceil$ элементами и $A[q..r]$ с $\lfloor n/2 \rfloor$ элементами[8].

```
MERGE_SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE_SORT(A, p, q)
4          MERGE_SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

Чтобы отсортировать последовательность $A = \langle A[1], A[2], \ldots, A[n] \rangle$, вызывается процедура MERGE_SORT$(A, 1, length[A])$, где $length[A] = n$. На рис. 2.4 проиллюстрирована работа этой процедуры в восходящем направлении, если n — это степень двойки. В ходе работы алгоритма происходит попарное объединение одноэлементных последовательностей в отсортированные последовательности длины 2, затем — попарное объединение двухэлементных последовательностей в отсортированные последовательности длины 4 и т.д., пока не будут

[8]Выражение $\lceil x \rceil$ обозначает наименьшее целое число, которое больше или равно x, а выражение $\lfloor x \rfloor$ — наибольшее целое число, которое меньше или равно x. Эти обозначения вводятся в главе 3. Чтобы убедиться в том, что в результате присваивания переменной q значения $\lfloor (p + r)/2 \rfloor$ длины подмассивов $A[p..q]$ и $A[q + 1..r]$ получаются равными $\lceil n/2 \rceil$ и $\lfloor n/2 \rfloor$, достаточно проверить четыре возможных случая, в которых каждое из чисел p и r либо четное, либо нечетное.

получены две последовательности, состоящие из $n/2$ элементов, которые объединяются в конечную отсортированную последовательность длины n.

2.3.2 Анализ алгоритмов, основанных на принципе "разделяй и властвуй"

Если алгоритм рекурсивно обращается к самому себе, время его работы часто описывается с помощью ***рекуррентного уравнения***, или ***рекуррентного соотношения***, в котором полное время, требуемое для решения всей задачи с объемом ввода n, выражается через время решения вспомогательных подзадач. Затем данное рекуррентное уравнение решается с помощью определенных математических методов, и устанавливаются границы производительности алгоритма.

Получение рекуррентного соотношения для времени работы алгоритма, основанного на принципе "разделяй и властвуй", базируется на трех этапах, соответствующих парадигме этого принципа. Обозначим через $T(n)$ время решения задачи, размер которой равен n. Если размер задачи достаточно мал, скажем, $n \leqslant c$, где c — некоторая заранее известная константа, то задача решается непосредственно в течение определенного фиксированного времени, которое мы обозначим через $\Theta(1)$. Предположим, что наша задача делится на a подзадач, объем каждой из которых равен $1/b$ от объема исходной задачи. (В алгоритме сортировки методом слияния числа a и b были равны 2, однако нам предстоит ознакомиться со многими алгоритмами разбиения, в которых $a \neq b$.) Если разбиение задачи на вспомогательные подзадачи происходит в течение времени $D(n)$, а объединение решений подзадач в решение исходной задачи — в течение времени $C(n)$, то мы получим такое рекуррентное соотношение:

$$T(n) = \begin{cases} \Theta(1) & \text{при } n \leqslant c, \\ aT(n/b) + D(n) + C(n) & \text{в противном случае.} \end{cases}$$

В главе 4 будет показано, как решаются рекуррентные соотношения такого вида.

Анализ алгоритма сортировки слиянием

Псевдокод MERGE_SORT корректно работает для произвольного (в том числе и нечетного) количества сортируемых элементов. Однако если количество элементов в исходной задаче равно степени двойки, то анализ рекуррентного уравнения упрощается. В этом случае на каждом шаге деления будут получены две подпоследовательности, размер которых точно равен $n/2$. В главе 4 будет показано, что это предположение не влияет на порядок роста, полученный в результате решения рекуррентного уравнения.

Чтобы получить рекуррентное уравнение для верхней оценки времени работы $T(n)$ алгоритма, выполняющего сортировку n чисел методом слияния, будем

рассуждать следующим образом. Сортировка одного элемента методом слияния длится в течение фиксированного времени. Если $n > 1$, время работы распределяется таким образом.

Разбиение. В ходе разбиения определяется, где находится средина подмассива. Эта операция длится фиксированное время, поэтому $D(n) = \Theta(1)$.

Покорение. Рекурсивно решаются две подзадачи, объем каждой из которых составляет $n/2$. Время решения этих подзадач равно $2T(n/2)$.

Комбинирование. Как уже упоминалось, процедура MERGE в n-элементном подмассиве выполняется в течение времени $\Theta(n)$, поэтому $C(n) = \Theta(n)$.

Сложив функции $D(n)$ и $C(n)$, получим сумму величин $\Theta(n)$ и $\Theta(1)$, которая является линейной функцией от n, т.е. $\Theta(n)$. Прибавляя к этой величине слагаемое $2T(n/2)$, соответствующее этапу "покорения", получим рекуррентное соотношение для времени работы $T(n)$ алгоритма сортировки по методу слияния в наихудшем случае:

$$T(n) = \begin{cases} \Theta(1) & \text{при } n = 1, \\ 2T(n/2) + \Theta(n) & \text{при } n > 1. \end{cases} \tag{2.1}$$

В главе 4 мы ознакомимся с теоремой, с помощью которой можно показать, что величина $T(n)$ представляет собой $\Theta(n \lg n)$, где $\lg n$ обозначает $\lg_2 n$. Поскольку логарифмическая функция растет медленнее, чем линейная, то для достаточно большого количества входных элементов производительность алгоритма сортировки методом слияния, время работы которого равно $\Theta(n \lg n)$, превзойдет производительность алгоритма сортировки методом вставок, время работы которого в наихудшем случае равно $\Theta(n^2)$.

Правда, можно и без упомянутой теоремы интуитивно понять, что решением рекуррентного соотношения (2.1) является выражение $T(n) = \Theta(n \lg n)$. Перепишем уравнение (2.1) в таком виде:

$$T(n) = \begin{cases} c & \text{при } n = 1, \\ 2T(n/2) + cn & \text{при } n > 1, \end{cases} \tag{2.2}$$

где константа c обозначает время, которое требуется для решения задачи? размер который равен 1, а также удельное (приходящееся на один элемент) время, требуемое для разделения и сочетания[9].

[9]Маловероятно, чтобы одна и та же константа представляла и время, необходимое для решения задачи, размер который равен 1, и приходящееся на один элемент время, в течение которого выполняются этапы разбиения и объединения. Чтобы обойти эту проблему, достаточно предположить, что c — максимальный из перечисленных промежутков времени. В таком случае мы получим верхнюю границу времени работы алгоритма. Если же в качестве c выбрать наименьший из всех перечисленных промежутков времени, то в результате решения рекуррентного соотношения получим нижнюю границу времени работы алгоритма. Принимая во внимание, что обе границы имеют порядок $n \lg n$, делаем вывод, что время работы алгоритма ведет себя, как $\Theta(n \lg n)$.

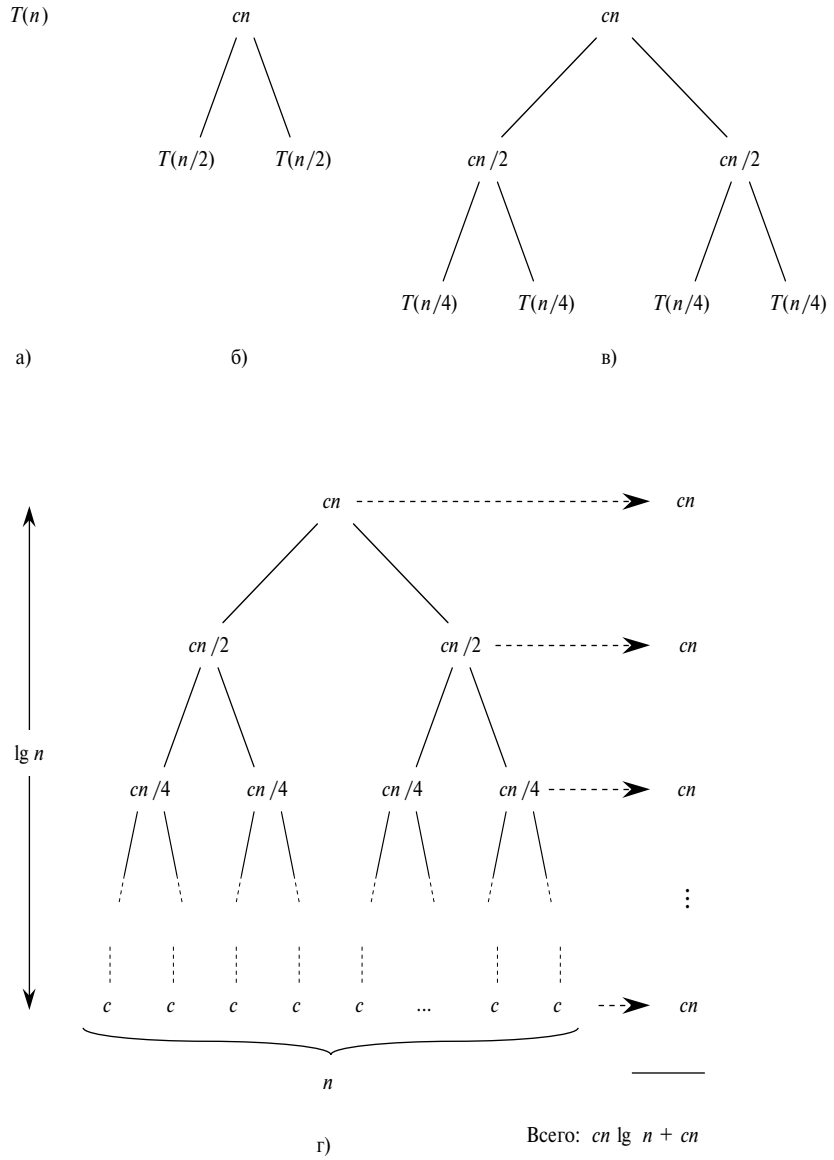


Рис. 2.5. Построение дерева рекурсии для уравнения $T(n) = 2T(n/2) + cn$

Процесс решения рекуррентного соотношения (2.2) проиллюстрирован на рис. 2.5. Для удобства предположим, что n равно степени двойки. В части *а* упомянутого рисунка показано время $T(n)$, представленное в части *б* в виде эквивалентного дерева, которое представляет рекуррентное уравнение. Корнем этого дерева является слагаемое cn (стоимость верхнего уровня рекурсии), а два

поддерева, берущих начало от корня, представляют две меньшие рекуррентные последовательности $T(n/2)$. В части *в* показан очередной шаг рекурсии. Время выполнения каждого из двух подузлов, находящихся на втором уровне рекурсии, равно $cn/2$. Далее продолжается разложение каждого узла, входящего в состав дерева, путем разбиения их на составные части, определенные в рекуррентной последовательности. Так происходит до тех пор, пока размер задачи не становится равным 1, а время ее выполнения — константе c. Получившееся в результате дерево показано в части *г*. Дерево состоит из $\lg n + 1$ уровней (т.е. его высота равна $\lg n$), а каждый уровень дает вклад в полное время работы, равный cn. Таким образом, полное время работы алгоритма равно $cn \lg n + cn$, что соответствует $\Theta(n \lg n)$.

После того как дерево построено, длительности выполнения всех его узлов суммируются по всем уровням. Полное время выполнения верхнего уровня равно cn, следующий уровень дает вклад, равный $c(n/2) + c(n/2) = cn$. Ту же величину вклада дают и все последующие уровни. В общем случае уровень i (если вести отсчет сверху) имеет 2^i узлов, каждый из которых дает вклад в общее время работы алгоритма, равный $c\left(n/2^i\right)$, поэтому полное время выполнения всех принадлежащих уровню узлов равно $2^i c\left(n/2^i\right) = cn$. На нижнем уровне имеется n узлов, каждый из которых дает вклад c, что в сумме дает время, равное cn.

Полное количество уровней дерева на рис. 2.5 равно $\lg n + 1$. Это легко понять из неформальных индуктивных рассуждений. В простейшем случае, когда $n = 1$, имеется всего один уровень. Поскольку $\lg 1 = 0$, выражение $\lg n + 1$ дает правильное количество уровней. Теперь в качестве индуктивного допущения примем, что количество уровней рекурсивного дерева с 2^i узлами равно $\lg 2^i + 1 = i + 1$ (так как для любого i выполняется соотношение $\lg 2^i = i$). Поскольку мы предположили, что количество входных элементов равняется степени двойки, то теперь нужно рассмотреть случай для 2^{i+1} элементов. Дерево с 2^{i+1} узлами имеет на один уровень больше, чем дерево с 2^i узлами, поэтому полное количество уровней равно $(i + 1) + 1 = \lg 2^{i+1} + 1$.

Чтобы найти полное время, являющееся решением рекуррентного соотношения (2.2), нужно просто сложить вклады от всех уровней. Всего имеется $\lg n + 1$ уровней, каждый из которых выполняется в течение времени cn, так что полное время равно $cn(\lg n + 1) = cn \lg n + cn$. Пренебрегая членами более низких порядков и константой c, в результате получаем $\Theta(n \lg n)$.

Упражнения

2.3-1. Используя в качестве образца рис. 2.4, проиллюстрируйте работу алгоритма сортировки методом слияний для массива $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

2.3-2. Перепишите процедуру MERGE так, чтобы в ней не использовались сигнальные значения. Сигналом к остановке должен служить тот факт, что все элементы массива L или массива R скопированы обратно в массив A, после чего в этот массив копируются элементы, оставшиеся в непустом массиве.

2.3-3. Методом математической индукции докажите, что если n равно степени двойки, то решением рекуррентного уравнения

$$T(n) = \begin{cases} 2 & \text{при } n = 2, \\ 2T(n/2) + n & \text{при } n = 2^k,\ k > 1 \end{cases}$$

является $T(n) = n \lg n$.

2.3-4. Сортировку вставкой можно представить в виде рекурсивной последовательности следующим образом. Чтобы отсортировать массив $A[1..n]$, сначала нужно выполнить сортировку массива $A[1..n-1]$, после чего в этот отсортированный массив помещается элемент $A[n]$. Запишите рекуррентное уравнение для времени работы этой рекурсивной версии алгоритма сортировки вставкой.

2.3-5. Возвращаясь к задаче поиска (см. упражнение 2.1-3), нетрудно заметить, что если последовательность A отсортирована, то значение среднего элемента этой последовательности можно сравнить с искомым значением v и сразу исключить половину последовательности из дальнейшего рассмотрения. ***Двоичный (бинарный) поиск*** (binary search) — это алгоритм, в котором такая процедура повторяется неоднократно, что всякий раз приводит к уменьшению оставшейся части последовательности в два раза. Запишите псевдокод алгоритма бинарного поиска (в итерационном или рекурсивном виде). Докажите, что время работы этого алгоритма в наихудшем случае возрастает как $\Theta(\lg n)$.

2.3-6. Обратите внимание, что в цикле **while** в строках 5–7 процедуры INSERTION_SORT в разделе 2.1, используется линейный поиск для просмотра (в обратном порядке) отсортированного подмассива $A[1..j-1]$. Можно ли использовать бинарный поиск (см. упражнение 2.3-5) вместо линейного, чтобы время работы этого алгоритма в наихудшем случае улучшилось и стало равным $\Theta(n \lg n)$?

⋆ 2.3-7. Пусть имеется множество S, состоящее из n целых чисел, и отдельное целое число x; необходимо определить, существуют ли во множестве S два элемента, сумма которых равна x. Разработайте алгоритм решения этой задачи, время работы которого возрастало бы с увеличением n как $\Theta(n \lg n)$.

Задачи

2-1. **Сортировка вставкой для небольших подмассивов, возникающих при сортировке слиянием**

Несмотря на то, что с увеличением количества сортируемых элементов время сортировки методом слияний в наихудшем случае растет как $\Theta(n \lg n)$, а время сортировки методом вставок — как $\Theta(n^2)$, благодаря постоянным множителям для малых n сортировка методом вставок выполняется быстрее. Таким образом, есть смысл использовать сортировку методом вставок в процессе сортировки методом слияний, когда подзадачи становятся достаточно маленькими. Рассмотрите модификацию алгоритма сортировки, работающего по методу слияний, когда n/k подмассивов длины k сортируются методом вставок, после чего они объединяются с помощью обычного механизма слияния. Величина k должна быть найдена в процессе решения задачи.

а) Покажите, что n/k подмассивов, в каждом из которых находится k элементов, в наихудшем случае можно отсортировать по методу вставок за время $\Theta(nk)$.

б) Покажите, что в наихудшем случае время, требуемое для объединения этих подмассивов, равно $\Theta(n \lg(n/k))$.

в) Допустим, что время работы модифицированного алгоритма в наихудшем случае равно $\Theta(nk + n \lg(n/k))$. Определите максимальное асимптотическое (в Θ-обозначениях) значение k как функцию от n, для которого асимптотическое время работы модифицированного алгоритма равно времени работы стандартного алгоритма сортировки методом слияния.

г) Как следует выбирать параметр k на практике?

2-2. **Корректность пузырьковой сортировки**

Пузырьковый метод — популярный алгоритм сортировки. В его основе лежит многократная перестановка соседних элементов, нарушающих порядок сортировки:

```
BUBBLESORT(A)
1  for i ← 1 to length[A]
2      do for j ← length[A] downto i + 1
3          do if A[j] < A[j − 1]
4              then Поменять местами A[j] ↔ A[j − 1]
```

а) Пусть A' — выходной массив, полученный в результате работы процедуры BUBBLESORT(A). Чтобы доказать, что этот алгоритм работает

корректно, нужно доказать, что он выполняется в течение конечного времени и что выполняются неравенства

$$A'[1] \leqslant A'[2] \leqslant \cdots \leqslant A'[n], \tag{2.3}$$

где $n = length[A]$. Что еще необходимо доказать, чтобы стало очевидно, что в ходе работы алгоритма BUBBLESORT действительно выполняется сортировка входного массива?

В следующих двух пунктах доказываются неравенства (2.3).

б) Дайте точную формулировку инварианта цикла **for** в строках 2–4, и докажите, что он сохраняется. Доказательство должно иметь такую же структуру доказательства инварианта цикла, которая использовалась в аналогичных доказательствах в данной главе.

в) С помощью условия завершения инварианта цикла, доказанного в пункте *б*, сформулируйте инвариант цикла **for** в строках 1–4, который бы позволил доказать неравенства (2.3). Доказательство должно иметь такую же структуру доказательства инварианта цикла, которая использовалась в аналогичных доказательствах в данной главе.

г) Определите время пузырьковой сортировки в наихудшем случае и сравните его со временем сортировки методом вставок.

2-3. **Корректность правила Горнера**

В приведенном ниже фрагменте кода реализовано правило Горнера (Horner's rule), позволяющее вычислить значение полинома

$$P(x) = \sum_{k=0}^{n} a_k x^k = \\ = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n)\ldots))$$

по заданным коэффициентам $a_0, a_1, \ldots, a_n$ и величине x:

```
1  y ← 0
2  i ← n
3  while i ≥ 0
4        do y ← a_i + x · y
5           i ← i − 1
```

а) Определите асимптотическое время работы приведенного выше фрагмента кода.

б) Напишите псевдокод, реализующий алгоритм обычного вычисления полинома, когда каждое слагаемое полинома вычисляется отдельно.

Определите асимптотическое время работы этого алгоритма и сравните его со временем работы алгоритма, основанного на правиле Горнера.

в) Докажите, что ниже сформулирован инвариант цикла **while** в строках 3–5.

> В начале каждой итерации цикла **while** в строках 3–5 выполняется такое соотношение:
>
> $$y = \sum_{k=0}^{n-(j+1)} a_{k+i+1} x^k.$$
>
> Суммирование по нулевому количеству слагаемых интерпретируется как 0. Доказательство должно иметь такую же структуру доказательства инварианта цикла, которая использовалась в аналогичных доказательствах в данной главе. В ходе доказательства необходимо продемонстрировать, что по завершении алгоритма выполняется соотношение $y = \sum_{k=0}^{n} a_k x^k$.

г) В заключение продемонстрируйте, что в приведенном фрагменте кода правильно вычисляется значение полинома, который задается коэффициентами $a_0, a_1, \ldots, a_n$.

2-4. **Инверсии**

Предположим, что $A[1..n]$ — это массив, состоящий из n различных чисел. Если $i < j$ и $A[i] > A[j]$, то пара (i, j) называется ***инверсией*** (inversion) в массиве A.

а) Перечислите пять инверсий, содержащихся в массиве $\langle 2, 3, 8, 6, 1 \rangle$.

б) Сконструируйте массив из элементов множества $\{1, 2, \ldots, n\}$, содержащий максимальное количество инверсий. Сколько инверсий в этом массиве?

в) Какая взаимосвязь между временем сортировки методом вставок и количеством инверсий во входном массиве? Обоснуйте ваш ответ.

г) Сформулируйте алгоритм, определяющий количество инверсий, содержащихся в произвольной перестановке n элементов, время работы которого в наихудшем случае равно $\Theta(n \lg n)$. (*Указание*: модифицируйте алгоритм сортировки слиянием.)

Заключительные замечания

В 1968 году Кнут (Knuth) опубликовал первый из трех томов, объединенных названием *The Art of Computer Programming* (Искусство программирования) [182,183,185]. Этот том стал введением в современные компьютерные алгоритмы с акцентом на анализе времени их работы, а весь трехтомник до сих пор остается интереснейшим и ценным пособием по многим темам, представленным в данной книге. Согласно Кнуту, слово “алгоритм” происходит от имени персидского математика девятнадцатого века аль-Хорезми (al-Khowârizmî).

Ахо (Aho), Хопкрофт (Hopcroft) и Ульман (Ullman) [5] являются сторонниками асимптотического анализа алгоритмов, являющегося средством сравнения их относительной производительности. Они также популяризируют применение рекуррентных соотношений для описания времени работы рекурсивных алгоритмов.

Кнут [185] с энциклопедической полнотой рассмотрел многие алгоритмы сортировки. Его сравнение алгоритмов сортировки включает в себя подробный анализ с точным подсчетом шагов, подобный тому, что был проведен в этой книге для сортировки методом вставок. В ходе обсуждения Кнутом алгоритма сортировки вставкой приводится несколько вариаций этого алгоритма. Важнейшей из них является сортировка по Шеллу (D.L. Shell), который использовал сортировку методом вставок для упорядочения периодических последовательностей, чтобы получить более производительный алгоритм сортировки.

В книге Кнута также описана сортировка слиянием. В книге упоминается, что в 1938 году была создана механическая машина, способная за один проход объединять две стопки перфокарт. По всей видимости, первым программу для сортировки методом слияния (предназначенную для компьютера EDVAC) в 1945 году разработал Джон фон Нейман (J. von Neumann), который был одним из первых создателей теории вычислительных машин.

Ранние этапы развития в области доказательства корректности программ описаны Гризом (Gries) [133], который считает, что первая статья по этой теме была написана Науром (P. Naur). Авторство понятия инварианта цикла Гриз приписывает Флойду (R.W. Floyd). В учебнике Митчелла (Mitchell) [222] описывается прогресс, достигнутый в области доказательства корректности программ в настоящее время.

ГЛАВА 3

Рост функций

Определенный в главе 2 порядок роста, характеризующий время работы алгоритма, является наглядной характеристикой эффективности алгоритма, а также позволяет сравнивать производительность различных алгоритмов. Если количество сортируемых элементов n становится достаточно большим, производительность алгоритма сортировки по методу слияний, время работы которого в наихудшем случае возрастает как $\Theta(n \lg n)$, становится выше производительности алгоритма сортировки вставкой, время работы которого в наихудшем случае возрастает как $\Theta\left(n^2\right)$. Несмотря на то, что в некоторых случаях можно определить точное время работы алгоритма, как это было сделано для алгоритма сортировки методом вставок в главе 2, обычно не стоит выполнять оценку с большой точностью. Для достаточно больших входных данных постоянные множители и слагаемые низшего порядка, фигурирующие в выражении для точного времени работы алгоритма, подавляются эффектами, вызванными увеличением размера ввода.

Рассматривая входные данные достаточно больших размеров для оценки только такой величины, как порядок роста времени работы алгоритма, мы тем самым изучаем ***асимптотическую*** эффективность алгоритмов. Это означает, что нас интересует только то, как время работы алгоритма растет с увеличением размера входных данных *в пределе*, когда этот размер увеличивается до бесконечности. Обычно алгоритм, более эффективный в асимптотическом смысле, будет более производительным для всех входных данных, за исключением очень маленьких.

В этой главе приведено несколько стандартных методов, позволяющих упростить асимптотический анализ алгоритмов. В начале следующего раздела вводится несколько видов “асимптотических обозначений”, одним из которых являются уже известные нам Θ-обозначения. Далее представлены некоторые соглашения

по поводу обозначений, принятых в данной книге. Наконец, в завершающей части главы рассматривается поведение функций, часто встречающихся при анализе алгоритмов.

3.1 Асимптотические обозначения

Обозначения, используемые нами для описания асимптотического поведения времени работы алгоритма, используют функции, область определения которых — множество неотрицательных целых чисел $\mathbf{N} = \{0, 1, 2, \ldots\}$. Подобные обозначения удобны для описания времени работы $T(n)$ в наихудшем случае, как функции, определенной только для целых чисел, представляющих собой размер входных данных. Однако иногда удобно изменить толкование асимптотических обозначений тем или иным образом. Например, эти обозначения легко обобщаются на область действительных чисел или, наоборот, ограничиваются до области, являющейся подмножеством натуральных чисел. При этом важно понимать точный смысл обозначений, чтобы изменение толкования не привело к неверному их использованию. В данном разделе вводятся основные асимптотические обозначения, а также описывается, как чаще всего изменяется их толкование.

Θ-обозначения

В главе 2 было показано, что время работы алгоритма сортировки методом вставок в наихудшем случае выражается функцией $T(n) = \Theta(n^2)$. Давайте разберемся в смысле данного обозначения. Для некоторой функции $g(n)$ запись $\Theta(g(n))$ обозначает *множество функций*

$$\Theta(g(n)) = \left\{ \begin{array}{l} f(n): \text{ существуют положительные константы } c_1, c_2 \text{ и } n_0 \\ \text{такие что } 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n) \text{ для всех } n \geqslant n_0 \end{array} \right\}^{1}.$$

Функция $f(n)$ принадлежит множеству $\Theta(g(n))$, если существуют положительные константы c_1 и c_2, позволяющие заключить эту функцию в рамки между функциями $c_1 g(n)$ и $c_2 g(n)$ для достаточно больших n. Поскольку $\Theta(g(n))$ — это множество, то можно написать "$f(n) \in \Theta(g(n))$". Это означает, что функция $f(n)$ принадлежит множеству $\Theta(g(n))$ (другими словами, является его элементом). Мы обычно будем использовать эквивалентную запись "$f(n) = \Theta(g(n))$". Такое толкование знака равенства для обозначения принадлежности множеству поначалу может сбить с толку, однако далее мы убедимся, что у нее есть свои преимущества.

[1] В теории множеств двоеточие следует читать как"такие, что" или "для которых выполняется условие".

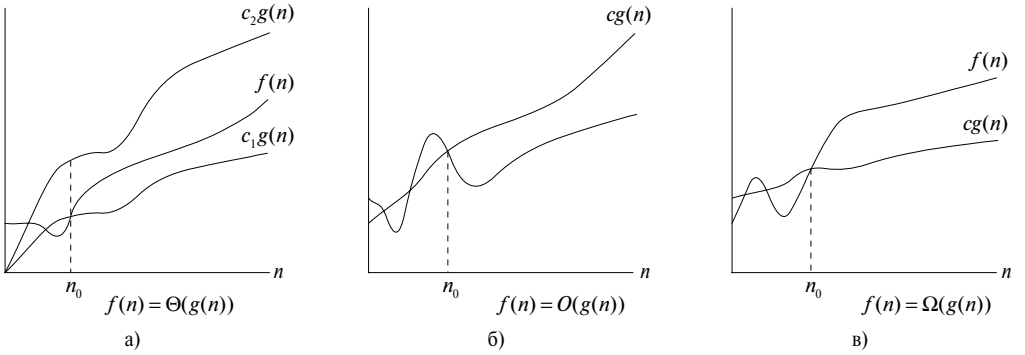


Рис. 3.1. Графические примеры Θ, O и Ω обозначений; в каждой части рисунка в качестве n_0 используется минимально возможное значение, т.е. любое большее значение также сможет выполнить роль n_0

На рис. 3.1*а* показано интуитивное изображение функций $f(n)$ и $g(n)$, таких что $f(n) = \Theta(g(n))$. Для всех значений n, лежащих справа от n_0, функция $f(n)$ больше или равна функции $c_1 g(n)$, но не превосходит функцию $c_2 g(n)$. Другими словами, для всех $n \geqslant n_0$ функция $f(n)$ равна функции $g(n)$ с точностью до постоянного множителя. Говорят, что функция $g(n)$ является ***асимптотически точной оценкой*** функции $f(n)$.

Согласно определению множества $\Theta(g(n))$, необходимо, чтобы каждый элемент $f(n) \in \Theta(g(n))$ этого множества был ***асимптотически неотрицателен***. Это означает, что при достаточно больших n функция $f(n)$ является неотрицательной. (***Асимптотически положительной*** называется такая функция, которая является положительной при любых достаточно больших n). Следовательно, функция $g(n)$ должна быть асимптотически неотрицательной, потому что в противном случае множество $\Theta(g(n))$ окажется пустым. Поэтому будем считать, что все функции, используемые в Θ-обозначениях, асимптотически неотрицательные. Это предположение также справедливо для других асимптотических обозначений, определенных в данной главе.

В главе 2 Θ-обозначения вводятся неформально. При этом игнорируется коэффициент при старшем слагаемом, а также отбрасываются слагаемые низшего порядка. Закрепим интуитивные представления, рассмотрев небольшой пример, в котором с помощью формального определения доказывается, что $n^2/2 - 3n = \Theta(n^2)$. Для этого необходимо определить, чему равны положительные константы c_1, c_2 и n_0, для которых выполняется соотношение

$$c_1 n^2 \leqslant \frac{1}{2} n^2 - 3n \leqslant c_2 n^2$$

для всех $n \geqslant n_0$. Разделив приведенное выше двойное неравенство на n^2, получим:

$$c_1 \leqslant \frac{1}{2} - \frac{3}{n} \leqslant c_2.$$

Правое неравенство выполняется для всех $n \geqslant 1$, если выбрать $c_2 \geqslant 1/2$. Аналогично, левое неравенство выполняется для всех $n \geqslant 7$, если выбрать $c_1 \leqslant 1/14$. Таким образом, выбрав $c_1 = 1/14$, $c_2 = 1/2$ и $n_0 = 7$, мы убеждаемся, что $n^2/2 - 3n = \Theta\left(n^2\right)$. Конечно же, константы можно выбрать по-другому, однако важно не то, как их выбрать, а то, что такая возможность *существует*. Обратите внимание, что эти константы зависят от вида рассматриваемой функции; чтобы доказать принадлежность множеству $\Theta\left(n^2\right)$ другой функции, скорее всего, понадобились бы другие константы.

С помощью формального определения можно также доказать, что $6n^3 \neq \Theta\left(n^2\right)$. Сделаем это методом "от противного". Сначала допустим существование таких c_2 и n_0, что $6n^3 \leqslant c_2 n^2$ для всех $n \geqslant n_0$. Однако тогда получается, что $n \leqslant c_2/6$, а это неравенство не может выполняться для больших n, поскольку c_2 — константа.

Интуитивно понятно, что при асимптотически точной оценке асимптотически положительных функций, слагаемыми низших порядков в них можно пренебречь, поскольку при больших n они становятся несущественными. Даже небольшой доли слагаемого самого высокого порядка достаточно для того, чтобы превзойти слагаемые низших порядков. Таким образом, для выполнения неравенств, фигурирующих в определении Θ-обозначений, достаточно в качестве c_1 выбрать значение, которое несколько меньше коэффициента при самом старшем слагаемом, а в качестве c_2 — значение, которое несколько больше этого коэффициента. Поэтому коэффициент при старшем слагаемом можно не учитывать, так как он лишь изменяет указанные константы.

В качестве примера рассмотрим квадратичную функцию $f(n) = an^2 + bn + c$, где a, b и c — константы, причем $a > 0$. Отбросив слагаемые низших порядков и игнорируя коэффициент, получим $f(n) = \Theta\left(n^2\right)$. Чтобы показать то же самое формально, выберем константы $c_1 = a/4$, $c_2 = 7a/4$ и $n_0 = 2\max((|b|/a), \sqrt{(|c|/a)})$. Читатель может сам убедиться, что неравенство $0 \leqslant c_1 n^2 \leqslant an^2 + bn + c \leqslant c_2 n^2$ выполняется для всех $n \geqslant n_0$. Вообще говоря, для любого полинома $p(n) = \sum_{i=0}^{d} a_i n^i$, где a_i — это константы и $a_d > 0$, $p(n) = \Theta\left(n^d\right)$ (см. задачу 3-1).

Поскольку любая константа — это полином нулевой степени, то постоянную функцию можно выразить как $\Theta\left(n^0\right)$ или $\Theta(1)$. Однако последнее обозначение не совсем точное, поскольку непонятно, по отношению к какой переменной исследуется асимптотика[2]. Мы часто будем употреблять $\Theta(1)$ для обозначения либо константы, либо постоянной функции от какой-нибудь переменной.

[2]На самом деле проблема состоит в том, что в наших обычных обозначениях функций не делается различия между функциями и обычными величинами. В λ-вычислениях четко указываются

O-обозначения

В Θ-обозначениях функция асимптотически ограничивается сверху и снизу. Если же достаточно определить только ***асимптотическую верхнюю границу***, используются O-обозначения. Для данной функции $g(n)$ обозначение $O(g(n))$ (произносится "о большое от g от n" или просто "о от g от n") означает множество функций, таких что

$$O(g(n)) = \left\{ \begin{array}{l} f(n): \text{ существуют положительные константы } c \text{ и } n_0 \\ \text{такие что } 0 \leqslant f(n) \leqslant cg(n) \text{ для всех } n \geqslant n_0 \end{array} \right\}.$$

O-обозначения применяются, когда нужно указать верхнюю границу функции с точностью до постоянного множителя. Интуитивное представление об O-обозначениях позволяет получить рис. 3.1*б*. Для всех n, лежащих справа от n_0, значение функции $f(n)$ не превышает значения функции $cg(n)$.

Чтобы указать, что функция $f(n)$ принадлежит множеству $O(g(n))$, пишут $f(n) = O(g(n))$. Обратите внимание, что из $f(n) = \Theta(g(n))$ следует $f(n) = O(g(n))$, поскольку Θ-обозначения более сильные, чем O-обозначения. В обозначениях теории множеств $\Theta(g(n)) \subset O(g(n))$. Таким образом, доказательство того, что функция an^2+bn+c, где $a > 0$, принадлежит множеству $\Theta(n^2)$, одновременно доказывает, что множеству $O(n^2)$ принадлежит любая такая квадратичная функция. Может показаться удивительным то, что любая *линейная* функция $an+b$ при $a > 0$ также принадлежит множеству $O(n^2)$, что легко проверить, выбрав $c = a + |b|$ и $n_0 = \max(1, -b/a)$.

Некоторым читателям, уже знакомым с O-обозначениями, может показаться странным, например, такое соотношение $n = O(n^2)$. В литературе O-обозначения иногда неформально используются для описания асимптотической точной оценки, т.е. так, как мы определили Θ-обозначения. Однако в данной книге, когда мы пишем $f(n) = O(g(n))$, то при этом подразумевается, что произведение некоторой константы на функцию $g(n)$ является асимптотическим верхним пределом функции $f(n)$. При этом не играет роли, насколько близко функция $f(n)$ находится к этой верхней границе. В литературе, посвященной алгоритмам, стало стандартом различать асимптотически точную оценку и верхнюю асимптотическую границу.

Чтобы записать время работы алгоритма в O-обозначениях, нередко достаточно просто изучить его общую структуру. Например, наличие двойного вложенного цикла в структуре алгоритма сортировки по методу вставок, представленному в главе 2, свидетельствует о том, что верхний предел времени работы в наихудшем

параметры функций: функцию от n^2 можно обозначить как $\lambda n.n^2$ или даже как $\lambda r.r^2$. Однако если принять более строгие обозначения, то алгебраические преобразования могут усложниться, поэтому мы предпочли более нестрогие обозначения.

случае выражается как $O\left(n^2\right)$: "стоимость" каждой итерации во внутреннем цикле ограничена сверху константой $O\left(1\right)$, индексы i и j — числом n, а внутренний цикл выполняется самое большее один раз для каждой из n^2 пар значений i и j.

Поскольку O-обозначения описывают верхнюю границу, то в ходе их использования для ограничения времени работы алгоритма в наихудшем случае мы получаем верхнюю границу этой величины для любых входных данных. Таким образом, граница $O\left(n^2\right)$ для времени работы алгоритма в наихудшем случае применима для времени решения задачи с любыми входными данными, чего нельзя сказать о Θ-обозначениях. Например, оценка $\Theta\left(n^2\right)$ для времени сортировки вставкой в наихудшем случае неприменима для *произвольных* входных данных. Например, в главе 2 мы имели возможность убедиться, что если входные элементы уже отсортированы, время работы алгоритма сортировки вставкой оценивается как $\Theta\left(n\right)$.

Если подходить формально, то неправильно говорить, что время, необходимое для сортировки по методу вставок, равняется $O\left(n^2\right)$, так как для данного n фактическое время работы алгоритма изменяется в зависимости от конкретных входных данных. Когда говорят, что "время работы равно $O\left(n^2\right)$", то подразумевается, что существует функция $f\left(n\right)$, принадлежащая $O\left(n^2\right)$ и такая, что при любом вводе размера n время решения задачи с данным вводом ограничено сверху значением функции $f\left(n\right)$. Это равнозначно тому, что в наихудшем случае время работы равно $O\left(n^2\right)$.

Ω-обозначения

Аналогично тому, как в O-обозначениях дается асимптотическая *верхняя* граница функции, в Ω-обозначениях дается ее ***асимптотическая нижняя граница***. Для данной функции $g\left(n\right)$ выражение $\Omega\left(g\left(n\right)\right)$ (произносится "омега большое от g от n" или просто "омега от g от n") обозначает множество функций, таких что

$$\Omega\left(g\left(n\right)\right) = \left\{ \begin{array}{l} f\left(n\right): \text{ существуют положительные константы } c \text{ и } n_0 \\ \text{такие что } 0 \leqslant cg\left(n\right) \leqslant f\left(n\right) \text{ для всех } n \geqslant n_0 \end{array} \right\}.$$

Интуитивное представление о Ω-обозначениях позволяет получить рис. 3.1*в*. Для всех n, лежащих справа от n_0, значения функции $f\left(n\right)$ больше или равны значениям $cg\left(n\right)$.

Пользуясь введенными определениями асимптотических обозначений, легко доказать сформулированную ниже теорему (см. упражнение 3.1-5).

Теорема 3.1. Для любых двух функций $f(n)$ и $g(n)$ соотношение $f(n) = \Theta(g(n))$ выполняется тогда и только тогда, когда $f\left(n\right) = O\left(g\left(n\right)\right)$ и $f\left(n\right) = \Omega\left(g\left(n\right)\right)$. ■

В качестве примера применения этой теоремы отметим, что из соотношения $an^2 + bn + c = \Theta\left(n^2\right)$ для произвольных констант a, b и c, где $a > 0$, непосред-

ственно следует, что $an^2 + bn + c = O\left(n^2\right)$ и $an^2 + bn + c = \Omega\left(n^2\right)$. На практике терема 3.1 применяется не для получения асимптотических верхней и нижней границ, как это сделано выше, а наоборот — для определения асимптотически точной оценки с помощью асимптотических верхней и нижней границ.

Поскольку Ω-обозначения используются для определения нижней границы времени работы алгоритма в наилучшем случае, они также дают нижнюю границу времени работы алгоритма для произвольных входных данных.

Итак, время работы алгоритма сортировки вставкой находится в пределах между $\Omega(n)$ и $O\left(n^2\right)$, т.е. между линейной и квадратичной функциями от n. Более того, эти границы охватывают асимптотику настолько плотно, насколько это возможно: например, нижняя оценка для времени работы алгоритма сортировки вставкой не может быть равной $\Omega\left(n^2\right)$, потому что существуют входные данные, для которых эта сортировка выполняется за время $\Theta(n)$ (когда входные элементы уже отсортированы), что не противоречит утверждению о том, что время работы алгоритма сортировки вставкой в наихудшем случае равно $\Omega\left(n^2\right)$, поскольку существуют входные данные, для которых этот алгоритм работает в течение времени $\Omega\left(n^2\right)$. Когда говорят, что *время работы* (без каких-либо уточнений) алгоритма равно $\Omega(g(n))$, при этом подразумевается, что *независимо от того, какие входные данные выбраны для данного размера* n, при достаточно больших n время работы алгоритма представляет собой как минимум константу, умноженную на $g(n)$.

Асимптотические обозначения в уравнениях и неравенствах

Мы уже видели, как асимптотические обозначения используются в математических формулах. Например, при введении O-обозначения мы писали "$n = O\left(n^2\right)$". Можно также написать $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. Как же интерпретируются подобные формулы?

Если в правой части уравнения (или неравенства) находится только асимптотическое обозначение, как в случае уравнения $n = O\left(n^2\right)$, то знак равенства используется для указания принадлежности множеству: $n \in O\left(n^2\right)$. Однако если асимптотические обозначения встречаются в формуле в другой ситуации, они рассматриваются как подставляемые взамен некоторой неизвестной функции, имя которой не имеет значения. Например, формула $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ означает, что $2n^2 + 3n + 1 = 2n^2 + f(n)$, где $f(n)$ — некоторая функция из множества $\Theta(n)$. В данном случае функция $f(n) = 3n + 1$, и она действительно принадлежит множеству $\Theta(n)$.

Подобное использование асимптотических обозначений позволяет избежать несущественных деталей и неразберихи в уравнениях. Например, в главе 2 время работы алгоритма сортировки методом слияния в наихудшем случае было

выражено в виде рекуррентного уравнения

$$T(n) = 2T(n/2) + \Theta(n).$$

Если нас интересует только асимптотическое поведение $T(n)$, то нет смысла точно выписывать все слагаемые низших порядков; подразумевается, что все они включены в безымянную функцию, обозначенную как $\Theta(n)$.

Предполагается, что таких функций в выражении столько, сколько раз в нем встречаются асимптотические обозначения. Например, в выражении $\sum_{i=1}^{n} O(i)$ имеется только одна функция без имени (аргументом которой является i). Таким образом, это выражение — *не одно и то же*, что и $O(1) + O(2) + \cdots + O(n)$, которое действительно не имеет однозначной интерпретации.

Иногда асимптотические обозначения появляются в левой части уравнения, как, например, в таком случае:

$$2n^2 + \Theta(n) = \Theta\left(n^2\right).$$

Подобные уравнения интерпретируются в соответствии с таким правилом: *при любом выборе безымянных функций, подставляемых вместо асимптотических обозначений в левую часть уравнения, можно выбрать и подставить в правую часть такие безымянные функции, что уравнение будет правильным*. Таким образом, смысл приведенного выше уравнения в том, что для *любой* функции $f(n) \in \Theta(n)$ существует *некоторая* функция $g(n) \in \Theta\left(n^2\right)$, такая что $2n^2 + f(n) = g(n)$ для всех n. Другими словами, правая часть уравнения предоставляет меньший уровень детализации, чем левая.

Несколько таких соотношений могут быть объединены в цепочку, например:

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta\left(n^2\right).$$

При этом каждое уравнение можно интерпретировать в соответствии со сформулированным выше правилом. Согласно первому уравнению, существует *некоторая* функция $f(n) \in \Theta(n)$, такая что для всех n выполняется соотношение $2n^2 + 3n + 1 = 2n^2 + f(n)$. Согласно второму уравнению, для любой функции $g(n) \in \Theta(n)$ существует *некоторая* функция $h(n) \in \Theta\left(n^2\right)$, такая что для всех n выполняется соотношение $2n^2 + g(n) = h(n)$. Заметим, что такая интерпретация подразумевает выполнение соотношения $2n^2 + 3n + 1 = \Theta\left(n^2\right)$, которое согласуется с нашими интуитивными представлениями о цепочке уравнений.

o-обозначения

Верхняя асимптотическая граница, предоставляемая O-обозначениями, может описывать асимптотическое поведение функции с разной точностью. Граница $2n^2 = O\left(n^2\right)$ дает правильное представление об асимптотическом поведении

функции, а граница $2n = O\left(n^2\right)$ его не обеспечивает. Для обозначения того, что верхняя граница не является асимптотически точной оценкой функции, применяются o-обозначения. Приведем формальное определение множества $o\left(g\left(n\right)\right)$ (произносится как "о малое от g от n"):

$$o\left(g\left(n\right)\right) = \left\{ \begin{array}{l} f\left(n\right): \text{ для любой положительной константы } c \text{ существует} \\ n_0 > 0, \text{ такое что } 0 \leqslant f\left(n\right) < cg\left(n\right) \text{ для всех } n \geqslant n_0 \end{array} \right\}.$$

Например: $2n = o\left(n^2\right)$, но $2n^2 \neq o\left(n^2\right)$.

Определения O-обозначений и o-обозначений похожи друг на друга. Основное отличие в том, что определение $f\left(n\right) = O\left(g\left(n\right)\right)$ ограничивает функцию $f\left(n\right)$ неравенством $0 \leqslant f\left(n\right) \leqslant cg\left(n\right)$ лишь для *некоторой* константы $c > 0$, а определение $f\left(n\right) = o\left(g\left(n\right)\right)$ ограничивает ее неравенством $0 \leqslant f\left(n\right) < cg\left(n\right)$ для *всех* констант $c > 0$. Интуитивно понятно, что в o-обозначениях функция $f\left(n\right)$ пренебрежимо мала по сравнению с функцией $g\left(n\right)$, если n стремится к бесконечности, т.е.

$$\lim_{n\to\infty} \frac{f\left(n\right)}{g\left(n\right)} = 0 \tag{3.1}$$

Некоторые авторы используют этот предел в качестве определения o-обозначений. Добавим, что определение, приведенное в этой книге, накладывает на безымянную функцию ограничение, согласно которому она должна быть асимптотически неотрицательной.

ω-обозначения

По аналогии, ω-обозначения соотносятся с Ω-обозначениями так же, как o-обозначения с O-обозначениями. С помощью ω-обозначений указывается нижний предел, не являющийся асимптотически точной оценкой. Один из возможных способов определения ω-обозначения следующий:

$$f\left(n\right) \in \omega\left(g\left(n\right)\right) \text{ тогда и только тогда, когда } g\left(n\right) \in o\left(f\left(n\right)\right).$$

Формально же $\omega\left(g\left(n\right)\right)$ (произносится как "омега малое от g от n") определяется как множество

$$o\left(g\left(n\right)\right) = \left\{ \begin{array}{l} f\left(n\right): \text{ для любой положительной константы } c \text{ существует} \\ n_0 > 0, \text{ такое что } 0 \leqslant cg\left(n\right) < f\left(n\right) \text{ для всех } n \geqslant n_0 \end{array} \right\}.$$

Например, $n^2/2 = \omega\left(n\right)$, но $n^2/2 \neq \omega\left(n^2\right)$. Соотношение $f\left(n\right) = \omega\left(g\left(n\right)\right)$ подразумевает, что $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, если этот предел существует. Таким образом, функция $f\left(n\right)$ становится сколь угодно большой по сравнению с функцией $g\left(n\right)$, если n стремится к бесконечности.

Сравнение функций

Асимптотические сравнения обладают некоторыми свойствами отношений обычных действительных чисел, показанными далее. Предполагается, что функции $f(n)$ и $g(n)$ асимптотически положительны.

Транзитивность

Из $f(n) = \Theta(g(n))$ и $g(n) = \Theta(h(n))$ следует $f(n) = \Theta(h(n))$.
Из $f(n) = O(g(n))$ и $g(n) = O(h(n))$ следует $f(n) = O(h(n))$.
Из $f(n) = \Omega(g(n))$ и $g(n) = \Omega(h(n))$ следует $f(n) = \Omega(h(n))$.
Из $f(n) = o(g(n))$ и $g(n) = o(h(n))$ следует $f(n) = o(h(n))$.
Из $f(n) = \omega(g(n))$ и $g(n) = \omega(h(n))$ следует $f(n) = \omega(h(n))$.

Рефлексивность

$$f(n) = \Theta(f(n)),$$
$$f(n) = O(f(n)),$$
$$f(n) = \Omega(f(n)).$$

Симметричность

$f(n) = \Theta(g(n))$ справедливо тогда и только тогда, когда $g(n) = \Theta(f(n))$.

Перестановочная симметрия

$f(n) = O(g(n))$ справедливо тогда и только тогда, когда $g(n) = \Omega(f(n))$,
$f(n) = o(g(n))$ справедливо тогда и только тогда, когда $g(n) = \omega(f(n))$.

Поскольку эти свойства выполняются для асимптотических обозначений, можно провести аналогию между асимптотическим сравнением двух функций f и g и сравнением двух действительных чисел a и b:

$$f(n) = O(g(n)) \approx a \leqslant b,$$
$$f(n) = \Omega(g(n)) \approx a \geqslant b,$$
$$f(n) = \Theta(g(n)) \approx a = b,$$
$$f(n) = o(g(n)) \approx a < b,$$
$$f(n) = \omega(g(n)) \approx a > b.$$

Говорят, что функция $f(n)$ ***асимптотически меньше*** функции $g(n)$, если $f(n) = o(g(n))$, и ***асимптотически больше*** функции $g(n)$, если $f(n) = \omega(g(n))$.

Однако одно из свойств действительных чисел в асимптотических обозначениях не выполняется.

***Трихотомия*:** для любых действительных чисел a и b должно выполняться только одно из соотношений $a < b$, $a = b$ или $a > b$.

Хотя любые два действительных числа можно однозначно сравнить, в отношении асимптотического сравнения функций это утверждение не является справедливым. Для двух функций $f(n)$ и $g(n)$ может не выполняться ни отношение $f(n) = O(g(n))$, ни $f(n) = \Omega(g(n))$. Например, функции n и $n^{1+\sin n}$ нельзя асимптотически сравнивать, поскольку показатель степени в функции $n^{1+\sin n}$ колеблется между значениями 0 и 2 (принимая все значения в этом интервале).

Упражнения

3.1-1. Пусть $f(n)$ и $g(n)$ — асимптотически неотрицательные функции. Докажите с помощью базового определения Θ-обозначений, что $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

3.1-2. Покажите, что для любых действительных констант a и b, где $b > 0$, справедливо соотношение

$$(n + a)^b = \Theta\left(n^b\right). \tag{3.2}$$

3.1-3. Объясните, почему выражение "время работы алгоритма A равно, как минимум, $O\left(n^2\right)$" не имеет смысла.

3.1-4. Справедливы ли соотношения $2^{n+1} = O(2^n)$ и $2^{2n} = O(2^n)$?

3.1-5. Докажите теорему 3.1.

3.1-6. Докажите, что время работы алгоритма равно $\Theta(g(n))$ тогда и только тогда, когда время работы алгоритма в наихудшем случае равно $O(g(n))$, а время работы алгоритма в наилучшем случае равно $\Omega(g(n))$.

3.1-7. Докажите, что множество $o(g(n)) \cap \omega(g(n))$ пустое.

3.1-8. Наши обозначения можно обобщить для случая двух параметров n и m, которые могут возрастать до бесконечности с разной скоростью. Для данной функции $g(n, m)$ выражение $O(g(n, m))$ обозначает множество функций, такое что

$$O(g(n,m)) = \left\{ \begin{array}{l} f(n,m): \text{ существуют положительные константы} \\ c,\ n_0 \text{ и } m_0, \text{ такие что } 0 \leqslant f(n,m) \leqslant cg(n,m) \\ \text{для всех } n \geqslant n_0 \text{ или } m \geqslant m_0 \end{array} \right\}.$$

Дайте аналогичные определения обозначений $\Omega(g(n, m))$ и $\Theta(g(n, m))$.

3.2 Стандартные обозначения и часто встречающиеся функции

В этом разделе рассматриваются некоторые стандартные математические функции и обозначения, а также исследуются взаимоотношения между ними. В нем также иллюстрируется применение асимптотических обозначений.

Монотонность

Функция $f(n)$ является ***монотонно неубывающей*** (monotonically increasing), если из неравенства $m \leqslant n$ следует неравенство $f(m) \leqslant f(n)$. Аналогично, функция $f(n)$ является ***монотонно невозрастающей*** (monotonically decreasing), если из неравенства $m \leqslant n$ следует неравенство $f(m) \geqslant f(n)$. Функция $f(n)$ ***монотонно возрастающая*** (strictly increasing), если из неравенства $m < n$ следует неравенство $f(m) < f(n)$ и ***монотонно убывающая*** (strictly decreasing), если из неравенства $m < n$ следует, что $f(m) > f(n)$.

Округление в большую и меньшую сторону

Для любого действительного числа x существует наибольшее целое число, меньшее или равное x, которое мы будем обозначать как $\lfloor x \rfloor$ (читается как "пол (floor) x"), и наименьшее целое число, большее или равное x, которое мы будем обозначать как $\lceil x \rceil$ (читается как "потолок (ceil) x"). Для всех действительных чисел

$$x - 1 \leqslant \lfloor x \rfloor \leqslant x \leqslant \lceil x \rceil < x + 1 \tag{3.3}$$

Для любого целого числа $n \lceil n/2 \rceil + \lfloor n/2 \rfloor = n$, а для любого действительного числа $n \geqslant 0$ и натуральных чисел a и b справедливы следующие соотношения:

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil, \tag{3.4}$$

$$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor, \tag{3.5}$$

$$\lceil a/b \rceil \leqslant (a + (b-1))/b, \tag{3.6}$$

$$\lfloor a/b \rfloor \geqslant (a - (b-1))/b. \tag{3.7}$$

Функция $f(x) = \lfloor x \rfloor$ является монотонно неубывающей, как и функция $f(x) = \lceil x \rceil$.

Модульная арифметика

Для любого целого числа a и любого натурального n величина $a \bmod n$ представляет собой ***остаток*** от деления a на n:

$$a \bmod n = a - \lfloor a/n \rfloor n \tag{3.8}$$

Располагая подобным определением, удобно ввести специальные обозначения для указания того, что два целых числа имеют одинаковые остатки при делении на какое-то натуральное число. Тот факт, что $(a \bmod n) = (b \bmod n)$, записывается как $a \equiv b \,(\mathrm{mod} n)$; при этом говорят, что число a ***эквивалентно***, или ***равно*** числу b по модулю n (или что числа a и b сравнимы по модулю n). Другими словами, $a \equiv b \,(\mathrm{mod} n)$, если числа a и b дают одинаковые остатки при делении на n. Это эквивалентно утверждению, что $a \equiv b \,(\mathrm{mod} n)$ тогда и только тогда, когда n является делителем числа $b - a$. Запись $a \not\equiv b \,(\mathrm{mod} n)$ означает, что число a не эквивалентно числу b по модулю n.

Полиномы

Полиномом степени d ***от аргумента*** n называется функция $p\,(n)$ следующего вида:

$$p\,(n) = \sum_{i=0}^{d} a_i n^i,$$

где константы $a_0, a_1, \ldots, a_d$ — коэффициенты полинома, и $a_d \neq 0$. Полином является асимптотически положительной функцией тогда и только тогда, когда $a_d > 0$. Для асимптотически положительных полиномов $p\,(n)$ степени d справедливо соотношение $p\,(n) = \Theta\left(n^d\right)$. Для любой действительной константы $a \geqslant 0$ функция n^a монотонно неубывающая, а для $a \leqslant 0$ эта функция монотонно невозрастающая. Говорят, что функция $f\,(n)$ ***полиномиально ограничена***, если существует такая константа k, что $f\,(n) = O\left(n^k\right)$.

Показательные функции

Для всех действительных чисел $a > 0$, m и n справедливы следующие тождества:

$$\begin{aligned} a^0 &= 1, \\ a^1 &= a, \\ a^{-1} &= 1/a, \\ (a^m)^n &= a^{mn}, \\ (a^m)^n &= (a^n)^m, \\ a^m a^n &= a^{m+n} \end{aligned}$$

Для любого n и $a \geqslant 1$ функция a^n является монотонно неубывающей функцией аргумента n. Для удобства будем считать, что $0^0 = 1$.

Соотношение между скоростями роста полиномов и показательных функций можно определить исходя из того факта, что для любых действительных констант

a и b, таких что $a > 1$,

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0 \tag{3.9}$$

откуда можно заключить, что $n^b = o\left(a^n\right)$.

Таким образом, любая показательная функция, основание a которой строго больше единицы, возрастает быстрее любой полиномиальной функции.

Обозначив через e основание натурального логарифма (приблизительно равное 2.718281828...), можем записать следующее соотношение, справедливое для любого действительного x:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{3.10}$$

где символ "!" обозначает факториал, определенный ниже в этом разделе. Для всех действительных x справедливо следующее неравенство:

$$e^x \geqslant 1 + x \tag{3.11}$$

Равенство соблюдается только в случае, если $x = 0$. При $|x| \leqslant 1$ можно использовать такое приближение:

$$1 + x \leqslant e^x \leqslant 1 + x + x^2 \tag{3.12}$$

При $x \to 0$ приближение значения функции e^x величиной $1 + x$ вполне удовлетворительно: $e^x = 1 + x + \Theta\left(x^2\right)$. (В этом уравнении асимптотические обозначения используются для описания предельного поведения при $x \to 0$, а не при $x \to \infty$.) Для произвольного x справедливо следующее равенство:

$$\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x \tag{3.13}$$

Логарифмы

Мы будем использовать следующие обозначения:

$$\begin{aligned}
\lg n &= \log_2 n && \text{(двоичный логарифм);} \\
\ln n &= \log_e n && \text{(натуральный логарифм);} \\
\lg^k n &= (\lg n)^k && \text{(возведение в степень);} \\
\lg \lg n &= \lg (\lg n) && \text{(композиция).}
\end{aligned}$$

Важное соглашение, которое мы приняли в книге, заключается в том, что *логарифмические функции применяются только к ближайшему члену выражения*.

Например, $\lg n + k$ означает $(\lg n) + k$, а не $\lg (n + k)$. Если основание логарифма $b > 1$, то при $n > 0$ функция $\log_b n$ монотонно возрастает.

Для всех действительных $a > 0$, $b > 0$, $c > 0$ и n выполняются следующие соотношения:

$$\begin{aligned} a &= b^{\log_b a}, \\ \log_c (ab) &= \log_c a + \log_c b, \\ \log_b a^n &= n \log_b a, \\ \log_b a &= \frac{\log_c a}{\log_c b}, \qquad (3.14) \\ \log_b (1/a) &= -\log_b a, \\ \log_b a &= \frac{1}{\log_a b}, \\ a^{\log_b c} &= c^{\log_b a} \qquad (3.15) \end{aligned}$$

где в каждом из приведенных выше уравнений основание логарифма отлично от 1.

Согласно уравнению (3.14), изменение основы логарифма приводит к умножению значения этого логарифма на постоянный множитель, поэтому мы часто будем использовать обозначение "$\lg n$", не заботясь о постоянном множителе, как это делается в O-обозначениях. Специалисты в области вычислительной техники считают наиболее естественной основой логарифма число 2, так как во многих алгоритмах и структурах данных производится разбиение задачи на две части.

При $|x| < 1$ для функции $\ln (1 + x)$ можно написать простое разложение в ряд:

$$\ln (1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \ldots .$$

Кроме того, при $x > -1$ выполняются следующие неравенства:

$$\frac{x}{1 + x} \leqslant \ln (1 + x) \leqslant x, \qquad (3.16)$$

причем знак равенства имеет место только при $x = 0$.

Говорят, что функция $f(n)$ ***полилогарифмически ограничена***, если существует такая константа k, что $f(n) = O\left(\lg^k n\right)$. Соотношение между скоростью роста полиномов и полилогарифмов можно найти, подставив в уравнение (3.9) $\lg n$ вместо n и 2^a вместо a, в результате чего получим:

$$\lim_{n \to \infty} \frac{\lg^b n}{\left(2^a\right)^{\lg n}} = \lim_{n \to \infty} \frac{\lg^b n}{n^a} = 0.$$

Из приведенного выше соотношения можно заключить, что для произвольной константы $a > 0 \lg^b n = o(n^a)$. Таким образом, любая положительная полиномиальная функция возрастает быстрее, чем любая полилогарифмическая функция.

Факториалы

Обозначение $n!$ (читается "n факториал") определяется для целых чисел $n \geqslant 0$ следующим образом:

$$n! = \begin{cases} 1 & \text{при } n = 0, \\ n \cdot (n-1)! & \text{при } n > 0, \end{cases}$$

т.е. $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$.

Верхнюю границу факториала можно записать в виде $n! \leqslant n^n$, поскольку все множители, которые определяют значение факториала, не превышают n. Однако эта оценка является грубой. Более точное приближение верхней (а также нижней) границы дает ***формула Стирлинга***:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) \tag{3.17}$$

где e — основание натурального логарифма. Можно доказать (см. упражнение 3.2-3), что

$$\begin{aligned} n! &= o\left(n^n\right), \\ n! &= \omega\left(2^n\right), \\ \lg\left(n!\right) &= \Theta\left(n \lg n\right) \end{aligned} \tag{3.18}$$

причем при доказательстве уравнения (3.18) удобно использовать формулу Стирлинга. Кроме того, для $n \geqslant 1$ справедливо следующее равенство:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \tag{3.19}$$

где

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}. \tag{3.20}$$

Функциональная итерация

Чтобы указать, что функция $f(n)$ последовательно i раз применяется к аргументу n, мы будем использовать обозначение $f^{(i)}(n)$. Дадим формальное определение. Пусть функция $f(n)$ задана на множестве действительных чисел. Для неотрицательных целых i можно дать такое рекурсивное определение:

$$f^{(i)}(n) = \begin{cases} n & \text{при } i = 0, \\ f\left(f^{(i-1)}(n)\right) & \text{при } i > 0. \end{cases}.$$

Например, если $f(n) = 2n$, то $f^{(i)}(n) = 2^i n$.

Итерированная логарифмическая функция

Обозначение $\lg^* n$ (читается как "логарифм со звездочкой от n") будет применяться для указания повторно применяемого логарифма, который определяется следующим образом. Пусть $\lg^{(i)} n$ представляет собой итерацию функции $f(n) = \lg n$. Поскольку логарифм от неотрицательных чисел не определен, то функция $\lg^{(i)} n$ определена, только если $\lg^{(i-1)} n > 0$. Не перепутайте обозначения $\lg^{(i)} n$ (логарифм, примененный последовательно i раз к аргументу n) и $\lg^i n$ (логарифм n, возведенный в i-ую степень). Итерированный логарифм определяется следующим образом:

$$\lg^* n = \min \left\{ i \geqslant 0 : \lg^{(i)} n \leqslant 1 \right\}.$$

(Другими словами, $\lg^* n$ — это минимальное число логарифмирований n, необходимое для получения значения, не превосходящего 1.)

Итерированный логарифм возрастает *очень* медленно:

$$\begin{aligned} \lg^* 2 &= 1, \\ \lg^* 4 &= 2, \\ \lg^* 16 &= 3, \\ \lg^* 65536 &= 4, \\ \lg^* \left(2^{65536}\right) &= 5. \end{aligned}$$

Поскольку количество атомов в наблюдаемой части Вселенной оценивается примерно в 10^{80}, что намного меньше, чем 2^{65536}, то можно с уверенностью сказать, что входные данные такого размера n, что $\lg^* n > 5$, встречаются крайне редко.

Числа Фибоначчи

Числа Фибоначчи определяются с помощью следующего рекуррентного соотношения:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_i &= F_{i-1} + F_{i-2}, \quad i \geqslant 2. \end{aligned} \tag{3.21}$$

Таким образом, числа Фибоначчи образуют последовательность, в которой каждое очередное число представляет собой сумму двух предыдущих:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots .$$

Числа Фибоначчи тесно связаны с ***золотым сечением*** ϕ и сопряженным значением $\widehat{\phi}$, которые задаются следующими формулами:

$$\begin{aligned}\phi &= \frac{1+\sqrt{5}}{2} = 1.61803\ldots \\ \widehat{\phi} &= \frac{1-\sqrt{5}}{2} = -0.61803\ldots\end{aligned} \tag{3.22}$$

В частности, справедливо такое соотношение:

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}}, \tag{3.23}$$

которое можно доказать методом индукции (см. упражнение 3.2-6). Поскольку $\left|\widehat{\phi}\right| < 1$, то выполняются неравенства $\left|\widehat{\phi}^i\right|/\sqrt{5} < 1/\sqrt{5} < 1/2$. Таким образом, i-е число Фибоначчи F_i равно $\phi^i/\sqrt{5}$, округленному до ближайшего целого числа. Следовательно, числа Фибоначчи возрастают как показательная функция.

Упражнения

3.2-1. Покажите, что если функции $f(n)$ и $g(n)$ — монотонно неубывающие, то функции $f(n) + g(n)$ и $f(g(n))$ также монотонно неубывающие. Кроме того, если к тому же функции $f(n)$ и $g(n)$ — неотрицательные, то монотонно неубывающей является также функция $f(n) \cdot g(n)$.

3.2-2. Докажите уравнение (3.15).

3.2-3. Докажите уравнение (3.18). Кроме того, докажите, что $n! = \omega(2^n)$ и $n! = o(n^n)$.

⋆ 3.2-4. Является ли функция $\lceil \lg n \rceil !$ полиномиально ограниченной? Является ли полиномиально ограниченной функция $\lceil \lg \lg n \rceil$?

⋆ 3.2-5. Какая из функций является асимптотически бóльшей: $\lg(\lg^* n)$ или $\lg^*(\lg n)$?

3.2-6. Докажите методом математической индукции, что i-е число Фибоначчи удовлетворяет следующему равенству:

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}},$$

где ϕ — золотое сечение, а $\widehat{\phi}$ — его сопряженное.

3.2-7. Докажите, что при $i \geqslant 0$ $(i+2)$-ое число Фибоначчи удовлетворяет неравенству $F_{i+2} > \phi^i$.

Задачи

3-1. **Асимптотическое поведение полиномов**

Пусть $p(n)$ представляет собой полином степени d от n:

$$p(n) = \sum_{i=0}^{d} a_i n^i,$$

где $a_d > 0$, и пусть k — некоторая константа. Докажите с помощью определения асимптотических обозначений приведенные ниже свойства.

а) Если $k \geqslant d$, то $p(n) = O\left(n^k\right)$.

б) Если $k \leqslant d$, то $p(n) = \Omega\left(n^k\right)$.

в) Если $k = d$, то $p(n) = \Theta\left(n^k\right)$.

г) Если $k > d$, то $p(n) = o\left(n^k\right)$.

д) Если $k < d$, то $p(n) = \omega\left(n^k\right)$.

3-2. **Сравнение скорости асимптотического роста**

Для каждой пары (A, B) приведенных в таблице выражений укажите, в каком отношении A находится по отношению к B: O, o, Ω, ω или Θ. Предполагается, что k, ε и c — константы, причем $k \geqslant 1$, $\varepsilon > 0$ и $c > 1$. Ответ должен быть представлен в виде таблицы, в каждой ячейке которой следует указать значения “да” или “нет”.

	A	B	O	o	Ω	ω	Θ
а.	$\lg^k n$	n^ε					
б.	n^k	c^n					
в.	$\sqrt{n}$	$n^{\sin n}$					
г.	2^n	$2^{n/2}$					
д.	$n^{\lg c}$	$c^{\lg n}$					
е.	$\lg(n!)$	$\lg\left(n^n\right)$					

3-3. **Упорядочение по скорости асимптотического роста**

а) Расположите приведенные ниже функции по скорости их асимптотического роста, т.е. постройте такую последовательность функций $g_1, g_2, \ldots, g_{30}$, что $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, $\ldots$, $g_{29} = \Omega(g_{30})$. Разбейте список на классы эквивалентности, в которых функции $f(n)$ и $g(n)$ принадлежат одному и тому же классу тогда и только тогда, когда $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$\left(\sqrt{2}\right)^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(3/2)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln\ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg\lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2\lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

б) Приведите пример неотрицательной функции $f(n)$, такой что ни для одной из представленных в задании *а*) функций $g_i(n)$ функция $f(n)$ не принадлежит ни множеству $O(g_i(n))$, ни множеству $\Omega(g_i(n))$.

3-4. **Свойства асимптотических обозначений**

Пусть $f(n)$ и $g(n)$ — асимптотически положительные функции. Докажите или опровергните справедливость каждого из приведенных ниже утверждений.

а) Из $f(n) = O(g(n))$ следует $g(n) = O(f(n))$.

б) $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

в) Из $f(n) = O(g(n))$ следует $\lg(f(n)) = O(\lg(g(n)))$, где при достаточно больших n $\lg(g(n)) \geqslant 1$ и $f(n) \geqslant 1$.

г) Из $f(n) = O(g(n))$ следует $2^{f(n)} = O\left(2^{g(n)}\right)$.

д) $f(n) = O\left((f(n))^2\right)$.

е) Из $f(n) = O(g(n))$ следует $g(n) = \Omega(f(n))$.

ж) $f(n) = \Theta(f(n/2))$.

з) $f(n) + o(f(n)) = \Theta(f(n))$.

3-5. **Вариации в определениях O и Ω**

Определения Ω некоторых авторов несколько отличаются от нашего; в качестве обозначения этих альтернативных определений мы будем использовать символ $\overset{\infty}{\Omega}$ (читается "Ω бесконечность"). Говорят, что $f(n) = \overset{\infty}{\Omega}(g(n))$, если существует положительная константа c, такая что $f(n) \geqslant cg(n) \geqslant 0$ для бесконечно большого количества целых n.

а) Покажите, что для любых двух асимптотически неотрицательных функций $f(n)$ и $g(n)$ выполняется одно из соотношений $f(n) = O(g(n))$ или $f(n) = \overset{\infty}{\Omega}(g(n))$ (или оба), в то время как при использовании Ω вместо $\overset{\infty}{\Omega}$ это утверждение оказывается ложным.

б) Опишите потенциальные преимущества и недостатки использования $\overset{\infty}{\Omega}$ вместо Ω для характеристики времени работы программы.

Определения O некоторых авторов несколько отличаются от нашего; в качестве обозначения этих альтернативных определений мы будем использовать символ O'. Говорят, что $f(n) = O'(g(n))$ тогда и только тогда, когда $|f(n)| = O(g(n))$.

в) Что произойдет, если теорему 3.1 перефразировать таким образом, что вместо O подставить O', а Ω оставить без изменений?

Некоторые авторы определяют $\tilde{O}$ (читается "о с тильдой") для обозначения O, в котором игнорируются логарифмические множители:

$$\tilde{O}(g(n)) = \left\{ \begin{array}{l} f(n): \text{ существуют положительные константы} \\ c,\ k \text{ и } n_0, \text{ такие что } 0 \leqslant f(n) \leqslant cg(n)\lg^k(n) \\ \text{для всех } n \geqslant n_0 \end{array} \right\}.$$

Дайте аналогичные определения $\tilde{\Omega}$ и $\tilde{\Theta}$. Докажите для них аналог теоремы 3.1.

3-6. **Итерированные функции**

Оператор итераций *, использующийся в функции $\lg^*$, можно применять к любой монотонно неубывающей функции $f(n)$, определенной на множестве действительных чисел. Для данной константы $c \in \mathbf{R}$ итерированная функция f_c^* определяется как

$$f_c^*(n) = \min\left\{ i \geqslant 0 : f^{(i)}(n) \leqslant c \right\},$$

где функция $f^{(i)}(n)$ должна быть корректно определена для всех i. Другими словами, величина $f_c^*(n)$ — это функция f, последовательно применяющаяся столько раз, сколько это необходимо для уменьшения аргумента до значения, не превышающего c.

Для каждой из приведенных ниже функций $f(n)$ и констант c дайте максимально точную оценку функции $f_c^*(n)$.

	$f(n)$	c	$f_c^*(n)$
а.	$n-1$	0	
б.	$\lg n$	1	
в.	$n/2$	1	
г.	$n/2$	2	
д.	$\sqrt{n}$	2	
е.	$\sqrt{n}$	1	
ж.	$n^{1/3}$	2	
з.	$n/\lg n$	2	

Заключительные замечания

Кнут (Knuth) [182] попытался выяснить происхождение O-обозначений и обнаружил, что впервые они появились в 1892 году в учебнике Бахманна (P. Bachmann) по теории чисел. O-обозначения были введены в 1909 году Ландау (E. Landau) при обсуждении распределения простых чисел. Введение Ω- и Θ-обозначений приписываются Кнуту [186], который исправил неточность популярного в литературе, но технически неаккуратного применения O-обозначений как для верхней, так и для нижней асимптотических границ. Многие продолжают использовать O-обозначения там, где более уместны были бы Θ-обозначения. Дальнейшее обсуждение исторического развития асимптотических обозначений можно найти у Кнута [182, 186], а также Брассарда (Brassard) и Брейтли (Bratley) [46].

Определения асимптотических обозначений у разных авторов иногда различаются, однако в большинстве часто встречающихся ситуаций они дают согласующиеся результаты. В некоторых альтернативных случаях подразумевается, что функции не являются асимптотически неотрицательными, поэтому ограничению подлежит их абсолютное значение.

Равенство (3.19) было получено Роббинсом (Robbins) [260]. Другие свойства элементарных математических функций можно найти в любом хорошем справочнике по математике, например, справочнике Абрамовича (Abramowitch) и Стеган (Stegun) [1] или Цвиллингера (Zwillinger) [320], а также в книгах по вычислительной математике, таких как книги Апостола (Apostol) [18] или Томаса (Thomas) и Финни (Finney) [296]. В книге Кнута [182], а также Грехема (Graham), Кнута и Паташника (Patashnik) [132] содержится множество материала по дискретной математике, который используется в информатике.

ГЛАВА 4

Рекуррентные соотношения

Как было сказано в главе 2, если алгоритм рекуррентно вызывает сам себя, время его работы часто можно описать с помощью рекуррентного соотношения. ***Рекуррентное соотношение*** (recurrence) — это уравнение или неравенство, описывающее функцию с использованием ее самой, но только с меньшими аргументами. Например, мы узнали, что время работы процедуры MERGE_SORT $T(n)$ в самом неблагоприятном случае описывается с помощью следующего рекуррентного соотношения:

$$T(n) = \begin{cases} \Theta(1) & \text{при } n = 1, \\ 2T(n/2) + \Theta(n) & \text{при } n > 1. \end{cases} \tag{4.1}$$

Решением этого соотношения является функция $T(n) = \Theta(n \lg n)$.

В настоящей главе вниманию читателя предлагаются три метода решения рекуррентных уравнений, т.е. получения асимптотических Θ- и O-оценок решения. В ***методе подстановок*** (substitution method) нужно догадаться, какой вид имеют граничные функции, а затем с помощью метода математической индукции доказать, что догадка правильная. В ***методе деревьев рекурсии*** (recursion-tree method) рекуррентное соотношение преобразуется в дерево, узлы которого представляют время выполнения каждого уровня рекурсии; затем для решения соотношения используется метод оценки сумм. В ***основном методе*** (master method) граничные оценки решений рекуррентных соотношений представляются в таком виде:

$$T(n) = aT(n/b) + f(n),$$

где $a \geqslant 1$, $b > 1$, а функция $f(n)$ — это заданная функция; для применения этого метода необходимо запомнить три различных случая, но после этого определение

асимптотических границ во многих простых рекуррентных соотношениях становится очень простым.

Технические детали

На практике в процессе формулировки и решения рекуррентных соотношений определенные технические моменты опускаются. Так, например, есть одна особенность, о которой часто умалчивают, — это допущение о том, что аргумент функции является целочисленным. Обычно время работы $T(n)$ алгоритма определено лишь для целочисленных n, поскольку в большинстве алгоритмов количество входных элементов выражается целым числом. Например, рекуррентное соотношение, описывающее время работы процедуры MERGE_SORT в наихудшем случае, на самом деле записывается так:

$$T(n) = \begin{cases} \Theta(1) & \text{при } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{при } n > 1. \end{cases} \tag{4.2}$$

Граничные условия — еще один пример технических особенностей, которые обычно игнорируются. Поскольку время работы алгоритма с входными данными фиксированного размера выражается константой, то в рекуррентных соотношениях, описывающих время работы алгоритмов, для достаточно малых n обычно справедливо соотношение $T(n) = \Theta(1)$. Поэтому для удобства граничные условия рекуррентных соотношений, как правило, опускаются и предполагается, что для малых n время работы алгоритма $T(n)$ является константой. Например, рекуррентное соотношение (4.1) обычно записывается как

$$T(n) = 2T(n/2) + \Theta(n), \tag{4.3}$$

без явного указания значений $T(n)$ для малых n. Причина состоит в том, что хотя изменение значения $T(1)$ приводит к изменению решения рекуррентного соотношения, это решение обычно изменяется не более, чем на постоянный множитель, а порядок роста остается неизменным.

Формулируя и решая рекуррентные соотношения, мы часто опускаем граничные условия, а также тот факт, что аргументами неизвестных функций являются целые числа. Мы продвигаемся вперед без этих деталей, а потом выясняем, важны они или нет. Даже если технические детали, которые опускаются при анализе алгоритмов, несущественны, то важно убедиться, что они не влияют на порядок роста решения. В подобных рассуждениях помогает опыт, а также некоторые теоремы. В этих теоремах формулируются условия, когда упомянутые детали не влияют на асимптотическое поведение рекуррентных соотношений, возникающих при анализе алгоритмов (см. теорему 4.1). Однако в данной главе мы не будем опускать технические детали, так как это позволит продемонстрировать некоторые тонкости, присущие методам решения рекуррентных соотношений.

4.1 Метод подстановки

Метод подстановки, применяющийся для решения рекуррентных уравнений, состоит из двух этапов:

1. делается догадка о виде решения;
2. с помощью метода математической индукции определяются константы и доказывается, что решение правильное.

Происхождение этого названия объясняется тем, что предполагаемое решение подставляется в рекуррентное уравнение. Это мощный метод, но он применим только в тех случаях, когда легко сделать догадку о виде решения.

Метод подстановки можно применять для определения либо верхней, либо нижней границ рекуррентного соотношения. В качестве примера определим верхнюю границу рекуррентного соотношения

$$T(n) = 2T(\lfloor n/2 \rfloor) + n, \tag{4.4}$$

подобного соотношениям (4.2) и (4.3). Мы предполагаем, что решение имеет вид $T(n) = O(n \lg n)$. Наш метод заключается в доказательстве того, что при подходящем выборе константы $c > 0$ выполняется неравенство $T(n) \leqslant cn \lg n$. Начнем с того, что предположим справедливость этого неравенства для величины $\lfloor n/2 \rfloor$, т.е. что выполняется соотношение $T(\lfloor n/2 \rfloor) \leqslant c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. После подстановки данного выражения в рекуррентное соотношение получаем:

$$\begin{aligned} T(n) &\leqslant 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \leqslant \\ &\leqslant cn \lg(n/2) + n = \\ &= cn \lg n - cn \lg 2 + n = \\ &= cn \lg n - cn + n \leqslant \\ &\leqslant cn \lg n, \end{aligned}$$

где последнее неравенство выполняется при $c \geqslant 1$.

Теперь, согласно методу математической индукции, необходимо доказать, что наше решение справедливо для граничных условий. Обычно для этого достаточно показать, что граничные условия являются подходящей базой для доказательства по индукции. В рекуррентном соотношении (4.4) необходимо доказать, что константу c можно выбрать достаточно большой для того, чтобы соотношение $T(n) \leqslant cn \lg n$ было справедливо и для граничных условий. Такое требование иногда приводит к проблемам. Предположим, что $T(1) = 1$ — единственное граничное условие рассматриваемого рекуррентного соотношения. Далее, для $n = 1$ соотношение $T(n) \leqslant cn \lg n$ дает нам $T(1) \leqslant c \cdot 1 \cdot \lg 1 = 0$, что противоречит условию $T(1) = 1$. Следовательно, данный базис индукции нашего доказательства не выполняется.

Эту сложность, возникающую при доказательстве предположения индукции для указанного граничного условия, легко обойти. Например, в рекуррентном соотношении (4.4) можно воспользоваться преимуществами асимптотических обозначений, требующих доказать неравенство $T(n) \leqslant cn \lg n$ только для $n \geqslant n_0$, где n_0 — выбранная нами константа. Идея по устранению возникшей проблемы заключается в том, чтобы в доказательстве по методу математической индукции не учитывать граничное условие $T(1) = 1$. Обратите внимание, что при $n > 3$ рассматриваемое рекуррентное соотношение явным образом от $T(1)$ не зависит. Таким образом, выбрав $n_0 = 2$, в качестве базы индукции можно рассматривать не $T(1)$, а $T(2)$ и $T(3)$. Заметим, что здесь делается различие между базой рекуррентного соотношения ($n = 1$) и базой индукции ($n = 2$ и $n = 3$). Из рекуррентного соотношения следует, что $T(2) = 4$, а $T(3) = 5$. Теперь доказательство по методу математической индукции соотношения $n \leqslant cn \lg n$ для некоторой константы $c \geqslant 1$ можно завершить, выбрав ее достаточно большой для того, чтобы были справедливы неравенства $T(2) \leqslant 2c \lg 2$ и $T(3) \leqslant 3c \lg 3$. Оказывается, что для этого достаточно выбрать $c \geqslant 2$. В большинстве рекуррентных соотношений, которые нам предстоит рассмотреть, легко расширить граничные условия таким образом, чтобы предположение индукции оказалось верным для малых n.

Как угадать решение

К сожалению, не существует общего способа, позволяющего угадать правильное решение рекуррентного соотношения. Для этого требуется опыт, удача и творческое мышление. К счастью, существуют определенные эвристические приемы, которые могут помочь сделать правильную догадку. Кроме того, для получения предполагаемого вида решения можно воспользоваться деревьями рекурсии, с которыми мы ознакомимся в разделе 4.2.

Если рекуррентное соотношение подобно тому, которое мы только что рассматривали, то разумно предположить, что решения этих соотношений будут похожими. Например, рассмотрим рекуррентное соотношение

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n,$$

которое выглядит более сложным, поскольку в аргументе функции T в его правой части добавлено слагаемое “17”. Однако интуитивно понятно, что это дополнительное слагаемое не может сильно повлиять на асимптотическое поведение решения. При достаточно больших n разность между $T(\lfloor n/2 \rfloor)$ и $T(\lfloor n/2 \rfloor + 17)$ становится несущественной: оба эти числа приблизительно равны половине числа n. Следовательно, можно предположить, что $T(n) = O(n \lg n)$, и проверить это с помощью метода подстановки (упражнение 4.1-5).

Другой способ найти решение — сделать грубую оценку его верхней и нижней границ, а затем уменьшить неопределенность до минимума. Например, в рекур-

рентном соотношении (4.4) в качестве начальной нижней границы можно было бы выбрать $T(n) = \Omega(n)$, поскольку в нем содержится слагаемое n; можно также доказать, что грубой верхней границей является $T(n) = O(n^2)$. Далее верхняя граница постепенно понижается, а нижняя — повышается до тех пор, пока не будет получено правильное асимптотическое поведение решения $T(n) = \Theta(n \lg n)$.

Тонкие нюансы

Иногда бывает так, что можно сделать правильное предположение об асимптотическом поведении решения рекуррентного соотношения, но при этом возникают трудности, связанные с выполнением доказательства по методу математической индукции. Обычно проблема заключается в том, что выбрано недостаточно сильное предположение индукции, которое не позволяет провести подробное рассмотрение. Натолкнувшись на такое препятствие, пересмотрите предположение индукции, избавившись от членов низшего порядка. При этом часто удается провести строгое математическое доказательство.

Рассмотрим рекуррентное соотношение:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1.$$

Можно предположить, что его решение — $O(n)$. Попытаемся показать, что для правильно выбранной константы c выполняется неравенство $T(n) \leqslant cn$. Подставив предполагаемое решение в рекуррентное соотношение, получим выражение

$$T(n) \leqslant c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 = cn + 1,$$

из которого не следует $T(n) \leqslant cn$ ни для какого c. Можно попытаться сделать другие предположения, например, что решение — $T(n) = O(n^2)$, но на самом деле правильным является именно наше первоначальное предположение. Однако чтобы это показать, необходимо выбрать более сильное предположение индукции.

Интуитивно понятно, что наша догадка была почти правильной: мы ошиблись всего лишь на константу, равную 1, т.е. на величину низшего порядка. Тем не менее, математическая индукция не работает, если в предположении индукции допущена даже такая, казалось бы, незначительная ошибка. Эту трудность можно преодолеть, если *вычесть* в первоначальном предположении слагаемое низшего порядка. Таким образом, теперь предположение индукции имеет вид $T(n) \leqslant cn - b$, где $b \geqslant 0$ — константа. Теперь мы имеем следующее соотношение:

$$\begin{aligned} T(n) &\leqslant (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1 = \\ &= cn - 2b + 1 \leqslant cn - b, \end{aligned}$$

которое справедливо при $b \geqslant 1$. Как и раньше, чтобы выполнялись граничные условия, константу c необходимо выбрать достаточно большой.

Большинство считает прием, при котором вычитаются слагаемые низшего порядка, трудным для понимания. Однако что же остается делать, когда математика не работает, если не усилить предположение? Чтобы понять этот шаг, важно помнить, что мы используем математическую индукцию: более сильное утверждение для данного фиксированного значения можно доказать, лишь предположив истинность более сильного утверждения для меньших значений.

Остерегайтесь ошибок

Используя асимптотические обозначения, легко допустить ошибку. Например, для рекуррентного соотношения легко "доказать", что $T(n) = O(n)$, предположив справедливость соотношения $T(n) \leqslant cn$, а затем выписав цепочку соотношений

$$\begin{aligned} T(n) &\leqslant 2(c\lfloor n/2 \rfloor) + n \leqslant cn + n = \\ &= O(n), \qquad \leftarrow \textbf{\textit{не верно!!!}} \end{aligned}$$

поскольку c — константа. Ошибка заключается в том, что не была доказана гипотеза индукции в *точном виде*, т.е. в виде $T(n) \leqslant cn$.

Замена переменных

Иногда с помощью небольших алгебраических преобразований удается добиться того, что неизвестное рекуррентное соотношение становится похожим на то, с которым мы уже знакомы. Например, рассмотрим следующее рекуррентное соотношение:

$$T(n) = 2T\left(\lfloor \sqrt{n} \rfloor\right) + \lg n,$$

которое выглядит довольно сложным. Однако его можно упростить, выполнив замену переменных. Для удобства мы не станем беспокоиться об округлении таких значений, как $\sqrt{n}$, до целых чисел. Воспользовавшись заменой $m = \lg n$, получим:

$$T(2^m) = 2T\left(2^{m/2}\right) + m.$$

Теперь можно переименовать функцию $S(m) = T(2^m)$, после чего получается новое рекуррентное соотношение:

$$S(m) = 2S(m/2) + m,$$

которое очень похоже на рекуррентное соотношение (4.4). Это рекуррентное соотношение имеет такое же решение — $S(m) = O(m \lg m)$. Сделав обратную замену $S(m)$ на $T(n)$, получим

$$T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n).$$

Упражнения

4.1-1. Покажите, что $O(\lg n)$ является решением рекуррентного соотношения $T(n) = T(\lceil n/2 \rceil) + 1$.

4.1-2. Мы убедились, что $O(n \lg n)$ является решением рекуррентного соотношения $T(n) = 2T(\lfloor n/2 \rfloor) + n$. Покажите, что $\Omega(n \lg n)$ также является решением этого рекуррентного соотношения. Отсюда можно сделать вывод, что решением рассматриваемого рекуррентного соотношения является $\Theta(n \lg n)$.

4.1-3. Покажите, что преодолеть трудность, связанную с граничным условием $T(1) = 1$ в рекуррентном соотношении (4.4), можно путем выбора другого предположения индукции, не меняя при этом граничных условий.

4.1-4. Покажите, что $\Theta(n \lg n)$ является решением "точного" рекуррентного соотношения (4.2) для сортировки слиянием.

4.1-5. Покажите, что $O(n \lg n)$ является решением рекуррентного соотношения $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$.

4.1-6. Решите рекуррентное соотношение $T(n) = 2T(\sqrt{n}) + 1$ с помощью замены переменных. Решение должно быть асимптотически точным. При решении игнорируйте тот факт, что значения являются целыми числами.

4.2 Метод деревьев рекурсии

Хотя метод подстановок способен обеспечить краткий путь к подтверждению того факта, что предполагаемое решение рекуррентного соотношения является правильным, однако сделать хорошую догадку зачастую довольно трудно. Построение дерева рекурсии, подобного тому, с которым мы имели дело в главе 2 при анализе рекуррентного соотношения, описывающего время сортировки слиянием, — прямой путь к хорошей догадке. В ***дереве рекурсии*** (recursion tree) каждый узел представляет время, необходимое для выполнения отдельно взятой подзадачи, которая решается при одном из многочисленных рекурсивных вызовов функций. Далее значения времени работы отдельных этапов суммируются в пределах каждого уровня, а затем — по всем уровням дерева, в результате чего получаем полное время работы алгоритма. Деревья рекурсии находят практическое применение при рассмотрении рекуррентных соотношений, описывающих время работы алгоритмов, построенных по принципу "разделяй и властвуй".

Деревья рекурсии лучше всего подходят для того, чтобы помочь сделать догадку о виде решения, которая затем проверяется методом подстановок. При этом в догадке часто допускается наличие небольших неточностей, поскольку впоследствии она все равно проверяется. Если же построение дерева рекурсии и суммирование времени работы по всем его составляющим производится достаточно

тщательно, то само дерево рекурсии может стать средством доказательства корректности решения. В данном разделе деревья рекурсии применяются для получения предположений о виде решения, а в разделе 4.4 — непосредственно для доказательства теоремы, на которой базируется основной метод.

Например, посмотрим, как с помощью дерева рекурсии можно догадаться о виде решения рекуррентного соотношения $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Начнем с того, что оценим решение сверху. Как известно, при решении рекуррентных соотношений тот факт, что от аргументов функций берется целая часть, обычно является несущественным (это пример отклонений, которые можно допустить), поэтому построим дерево рекурсии для рекуррентного соотношения $T(n) = 3T(n/4) + cn^2$, записанного с использованием константы $c > 0$.

На рис. 4.1 проиллюстрирован процесс построения дерева рекурсии для рекуррентного соотношения $T(n) = 3T(n/4) + cn^2$. Для удобства предположим, что n — степень четверки (еще один пример допустимых отклонений). В части *а*) показана функция $T(n)$, которая затем все более подробно расписывается в частях *б*)–*г*) в виде эквивалентного дерева рекурсии, представляющего анализируемое рекуррентное соотношение. Член cn^2 в корне дерева представляет время верхнего уровня рекурсии, а три поддерева, берущих начало из корня, — времена выполнения подзадач размера $n/4$. В части *в*) добавлен еще один шаг, т.е. выполнено представление в виде поддерева каждого узла с временем $T(n/4)$. Время выполнения, соответствующее каждому из трех дочерних поддеревьев, равно $c(n/4)^2$. Далее каждый лист дерева преобразуется в поддерево аналогичным образом, в соответствии с рекуррентным соотношением.

Поскольку по мере удаления от корня дерева размер подзадач уменьшается, в конце концов мы должны дойти до граничных условий. Сколько же уровней дерева нужно построить, чтобы их достичь? Размер вспомогательной задачи, соответствующей уровню, который находится на i-ом уровне глубины, равен $n/4^i$. Таким образом, размер подзадачи становится равным 1, когда $n/4^i = 1$ или, что то же самое, когда $i = \log_4 n$. Таким образом, всего в дереве $\log_4 n + 1$ уровней.

Затем мы определяем время выполнения для каждого уровня дерева. На каждом уровне в три раза больше узлов, чем на предыдущем уровне, поэтому количество узлов на i-м уровне равно 3^i. Поскольку размеры вспомогательных подзадач при спуске на один уровень уменьшаются в четыре раза, время выполнения каждого узла на i-м уровне ($i = \overline{0, \log_4 n - 1}$) равно $c\left(n/4^i\right)^2$. Умножая количество узлов на уровне на время выполнения одного узла, получаем, что суммарное время выполнения вспомогательных подзадач, соответствующих узлам i-го уровня, равно $3^i c\left(n/4^i\right)^2 = (3/16)^i cn^2$. Последний уровень, находящийся на глубине $\log_4 n$, имеет $3^{\log_4 n} = n^{\log_4 3}$ узлов, каждый из которых дает в общее время работы вклад, равный $T(1)$. Поэтому время работы этого уровня равно величине $n^{\log_4 3} T(1)$, которая ведет себя как $\Theta\left(n^{\log_4 3}\right)$.

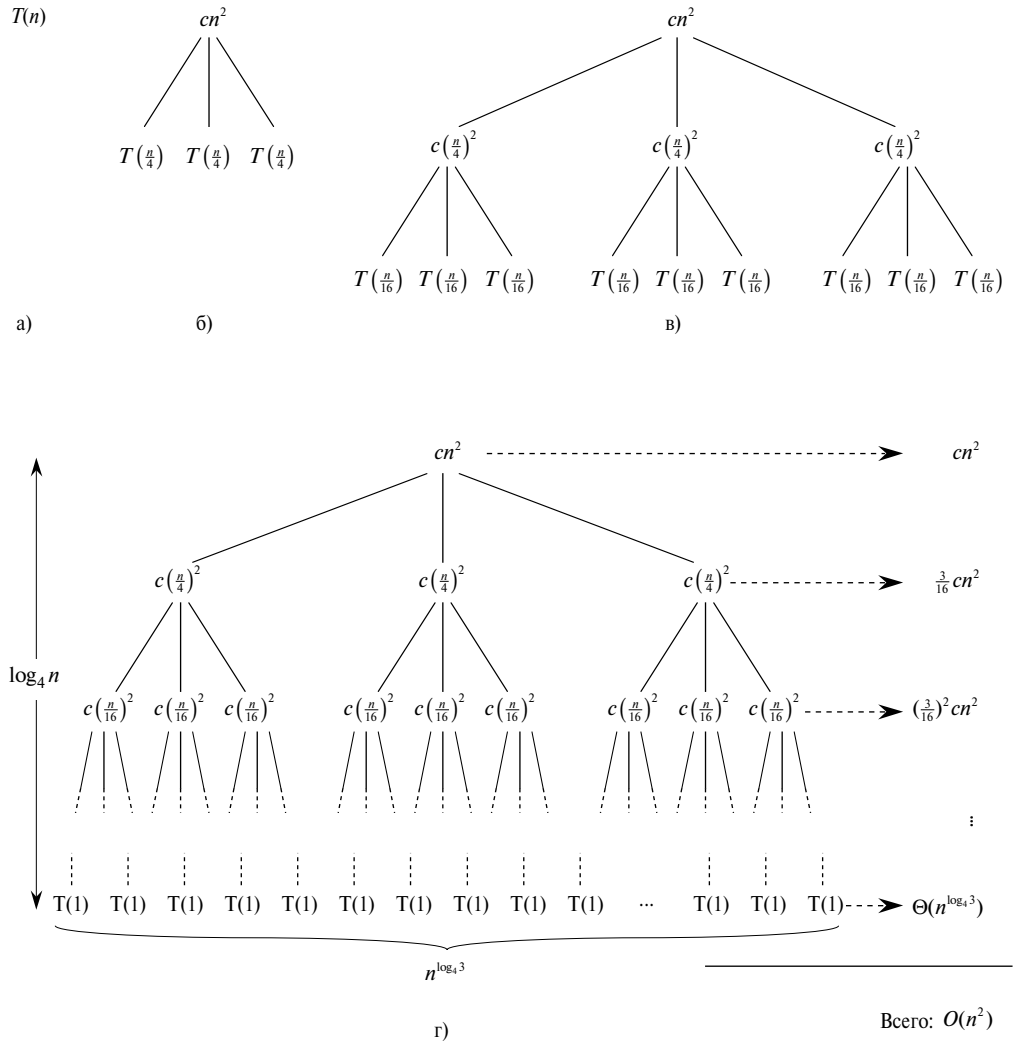


Рис. 4.1. Построение дерева рекурсии для рекуррентного соотношения $T(n) = 3T(n/4) + cn^2$

Теперь мы суммируем времена работы всех уровней дерева и определяем время работы дерева целиком:

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta\left(n^{\log_4 3}\right) =$$

$$= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) = \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta\left(n^{\log_4 3}\right).$$

Эта формула выглядит несколько запутанной, но только до тех пор, пока мы не догадаемся воспользоваться той небольшой свободой, которая допускается при асимптотических оценках, и не используем в качестве верхней границы одного из слагаемых бесконечно убывающую геометрическую прогрессию. Возвращаясь на один шаг назад и применяя формулу (A.6), получаем:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) < \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) = \\ &= \frac{1}{1-(3/16)} cn^2 + \Theta\left(n^{\log_4 3}\right) = \\ &= \frac{16}{13} cn^2 + \Theta\left(n^{\log_4 3}\right) = \\ &= O\left(n^2\right). \end{aligned}$$

Таким образом, для исходного рекуррентного соотношения $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta\left(n^2\right)$ получен предполагаемый вид решения $T(n) = O\left(n^2\right)$. В рассматриваемом примере коэффициенты при cn^2 образуют геометрическую прогрессию, а их сумма, согласно уравнению (A.6), ограничена сверху константой $16/13$. Поскольку вклад корня дерева в полное время работы равен cn^2, время работы корня представляет собой некоторую постоянную часть от общего времени работы всего дерева в целом. Другими словами, полное время работы всего дерева в основном определяется временем работы его корня.

В действительности, если $O\left(n^2\right)$ в действительности представляет собой верхнюю границу (в чем мы вскоре убедимся), то эта граница должна быть асимптотически точной оценкой. Почему? Потому что первый рекурсивный вызов дает вклад в общее время работы алгоритма, который выражается как $\Theta\left(n^2\right)$, поэтому нижняя граница решения рекуррентного соотношения представляет собой $\Omega\left(n^2\right)$.

Теперь с помощью метода подстановок проверим корректность нашей догадки, т.е. убедимся, что $T(n) = O\left(n^2\right)$ — верхняя граница рекуррентного соотношения $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta\left(n^2\right)$. Нам надо показать, что для некоторой константы $d > 0$ выполняется неравенство $T(n) \leqslant dn^2$. Используя ту же константу $c > 0$, что и раньше, получаем:

$$\begin{aligned} T(n) &\leqslant 3T(\lfloor n/4 \rfloor) + cn^2 \leqslant 3d\lfloor n/4 \rfloor^2 + cn^2 \leqslant \\ &\leqslant 3d(n/4)^2 + cn^2 = \frac{3}{16} dn^2 + cn^2 \leqslant dn^2, \end{aligned}$$

где последнее неравенство выполняется при $d \geqslant (16/13)\, c$.

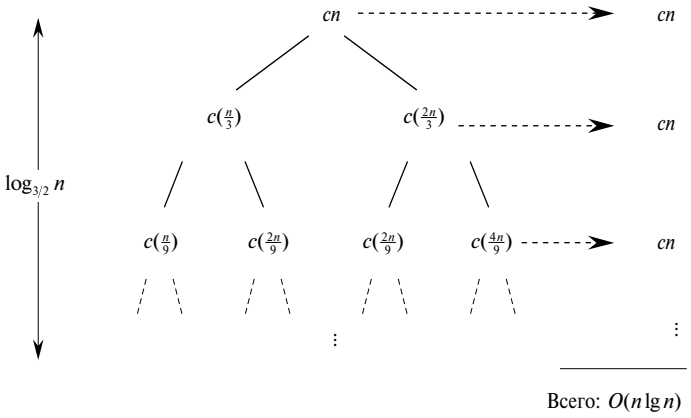


Рис. 4.2. Дерево рекурсии, соответствующее рекуррентному соотношению $T(n) = T(n/3) + T(2n/3) + cn$

Давайте теперь рассмотрим более сложный пример. На рис. 4.2 представлено дерево рекурсии для рекуррентного соотношения:

$$T(n) = T(n/3) + T(2n/3) + cn.$$

(Здесь также для простоты опущены функции "пол" и "потолок".) Пусть c, как и раньше, представляет постоянный множитель в члене $O(n)$. При суммировании величин по всем узлам дерева, принадлежащим определенному уровню, для каждого уровня мы получаем величину cn. Самый длинный путь от корня дерева до его листа имеет вид $n \to (2/3)n \to (2/3)^2 n \to \ldots \to 1$. Из $(2/3)^k n = 1$ следует, что высота дерева равна $\log_{3/2} n$.

Интуитивно понятно, что асимптотическое поведение предполагаемого решения рекуррентного соотношения определяется произведением количества уровней на время выполнения каждого уровня, т.е. $O\left(cn \log_{3/2} n\right) = O(n \lg n)$. В данном случае время выполнения равномерно распределяется по всем уровням дерева рекурсии. Здесь возникает одна сложность: нужно определить время работы каждого листа. Если бы рассматриваемое дерево рекурсии было полностью бинарным с высотой $\log_{3/2} n$, то всего имелось бы $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ листьев. Поскольку время выполнения каждого листа выражается константой, сумма этих констант по всем листьям была бы равна $\Theta\left(n^{\log_{3/2} 2}\right)$, т.е. $\omega(n \lg n)$. На самом деле это дерево рекурсии нельзя назвать полностью бинарным, поэтому у него меньше, чем $n^{\log_{3/2} 2}$ листьев. Более того, по мере удаления от корня отсутствует все большее количество внутренних узлов. Следовательно, не для всех уровней время их работы выражается как cn; более низкие уровни дают меньший вклад. Можно было бы попытаться аккуратно учесть вклады всех элементов дерева, однако вспомним, что мы всего лишь угадываем вид решения, чтобы затем воспользоваться

методом подстановок. Давайте отклонимся от точного решения и допустим, что асимптотическая верхняя граница решения ведет себя как $O(n \lg n)$.

С помощью метода подстановок мы можем убедиться, что сделанное выше предположение корректно. Покажем, что при подходящем выборе положительной константы d выполняется неравенство $T(n) \leqslant dn \lg n$. Можно записать цепочку соотношений:

$$\begin{aligned} T(n) &\leqslant T(n/3) + T(2n/3) + cn \leqslant \\ &\leqslant d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn = \\ &= (d(n/3)\lg n - d(n/3)\lg 3) + (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn = \\ &= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn = \\ &= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn = \\ &= dn\lg n - dn(\lg 3 - 2/3) + cn \leqslant \\ &\leqslant dn\lg n, \end{aligned}$$

которые выполняются при $d \geqslant c/(\lg 3 - (2/3))$. Таким образом, решение не делать более точный учет времени работы элементов, из которых состоит дерево рекурсии, вполне оправдало себя.

Упражнения

4.2-1. Определите с помощью дерева рекурсии асимптотическую верхнюю границу рекуррентного соотношения $T(n) = 3T(\lfloor n/2 \rfloor) + n$. Проверьте ответ методом подстановок.

4.2-2. Обратившись к дереву рекурсии, докажите, что решение рекуррентного соотношения $T(n) = T(n/3) + T(2n/3) + cn$, где c — константа, ведет себя как $\Omega(n \lg n)$.

4.2-3. Постройте дерево рекурсии для рекуррентного соотношения $T(n) = 4T(\lfloor n/2 \rfloor) + cn$, где c — константа, и найдите точную асимптотическую границу его решения. Проверьте ее с помощью метода подстановок.

4.2-4. Найдите с помощью дерева рекурсии точную асимптотическую оценку решения рекуррентного соотношения $T(n) = T(n-a) + T(a) + cn$, где $a \geqslant 1$ и $c > 0$ — константы.

4.2-5. Найдите с помощью дерева рекурсии точную асимптотическую оценку решения рекуррентного соотношения $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$, где $0 < \alpha < 1$ и $c > 0$ — константы.

4.3 Основной метод

Основной метод является своего рода "сборником рецептов", по которым строятся решения рекуррентных соотношений вида

$$T(n) = aT(n/b) + f(n), \tag{4.5}$$

где $a \geqslant 1$ и $b > 1$ — константы, а $f(n)$ — асимптотически положительная функция. Чтобы научиться пользоваться этим методом, необходимо запомнить три случая, что значительно облегчает решение многих рекуррентных соотношений, а часто это можно сделать даже в уме.

Рекуррентное соотношение (4.5) описывает время работы алгоритма, в котором задача размером n разбивается на a вспомогательных задач, размером n/b каждая, где a и b — положительные константы. Полученные в результате разбиения a подзадач решаются рекурсивным методом, причем время их решения равно $T(n/b)$. Время, требуемое для разбиения задачи и объединения результатов, полученных при решении вспомогательных задач, описывается функцией $f(n)$. Например, в рекуррентном соотношении, возникающем при анализе процедуры MERGE_SORT, $a = 2$, $b = 2$, а $f(n) = \Theta(n)$.

Строго говоря, при определении приведенного выше рекуррентного соотношения допущена неточность, поскольку число n/b может не быть целым. Однако замена каждого из a слагаемых $T(n/b)$ выражением $T(\lfloor n/b \rfloor)$ или $T(\lceil n/b \rceil)$ не влияет на асимптотическое поведение решения (это будет доказано в следующем разделе). Поэтому обычно при составлении рекуррентных соотношений подобного вида, полученных методом "разделяй и властвуй", обычно мы будем игнорировать функции "пол" и "потолок", с помощью которых аргумент преобразуется к целому числу.

Основная теорема

Основной метод базируется на приведенной ниже теореме.

Теорема 4.1 (Основная теорема). Пусть $a \geqslant 1$ и $b > 1$ — константы, $f(n)$ — произвольная функция, а $T(n)$ — функция, определенная на множестве неотрицательных целых чисел с помощью рекуррентного соотношения

$$T(n) = aT(n/b) + f(n),$$

где выражение n/b интерпретируется либо как $\lfloor n/b \rfloor$, либо как $\lceil n/b \rceil$. Тогда асимптотическое поведение функции $T(n)$ можно выразить следующим образом.

1. Если $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ для некоторой константы $\varepsilon > 0$, то $T(n) = \Theta\left(n^{\log_b a}\right)$.

2. Если $f(n) = \Theta\left(n^{\log_b a}\right)$, то $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.
3. Если $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$ для некоторой константы $\varepsilon > 0$, и если $af(n/b) \leqslant cf(n)$ для некоторой константы $c < 1$ и всех достаточно больших n, то $T(n) = \Theta(f(n))$. ■

Перед тем как применение основной теоремы будет продемонстрировано на конкретных примерах, попытаемся понять ее суть. В каждом из трех выделенных в теореме случаев функция $f(n)$ сравнивается с функцией $n^{\log_b a}$. Интуитивно понятно, что асимптотическое поведение решения рекуррентного соотношения определяется большей из двух функций. Если большей является функция $n^{\log_b a}$, как в случае 1, то решение — $T(n) = \Theta\left(n^{\log_b a}\right)$. Если быстрее возрастает функция $f(n)$, как в случае 3, то решение — $T(n) = \Theta(f(n))$. Если же обе функции сравнимы, как в случае 2, то происходит умножение на логарифмический множитель и решение — $T(n) = \Theta\left(n^{\log_b a} \lg n\right) = \Theta(f(n) \lg n)$.

Кроме этих интуитивных рассуждений, необходимо разобраться с некоторыми техническими деталями. В первом случае функция $f(n)$ должна быть не просто меньше функции $n^{\log_b a}$, а *полиномиально* меньше. Это означает, что функция $f(n)$ должна быть асимптотически меньше функции $n^{\log_b a}$ в n^{ε} раз, где ε — некоторая положительная константа. Аналогично, в третьем случае функция $f(n)$ должна быть не просто больше функции $n^{\log_b a}$, а полиномиально больше, и, кроме того, удовлетворять условию "регулярности", согласно которому $af(n/b) \leqslant cf(n)$. Это условие выполняется для большинства полиномиально ограниченных функций, с которыми мы будем иметь дело.

Важно понимать, что этими тремя случаями не исчерпываются все возможности поведения функции $f(n)$. Между случаями 1 и 2 есть промежуток, в котором функция $f(n)$ меньше функции $n^{\log_b a}$, но не полиномиально меньше. Аналогичный промежуток имеется между случаями 2 и 3, когда функция $f(n)$ больше функции $n^{\log_b a}$, но не полиномиально больше. Если функция $f(n)$ попадает в один из этих промежутков, или если для нее не выполняется условие регулярности из случая 3, основной метод неприменим для решения рекуррентных соотношений.

Использование основного метода

Чтобы использовать основной метод, достаточно определить, какой из частных случаев основной теоремы (если такой есть) применим в конкретной задаче, а затем записать ответ.

В качестве первого примера рассмотрим такое рекуррентное соотношение:

$$T(n) = 9T(n/3) + n.$$

В этом случае $a = 9$, $b = 3$, $f(n) = n$, так что $n^{\log_b a} = n^{\log_3 9} = \Theta\left(n^2\right)$. Поскольку $f(n) = O\left(n^{\log_3 9-\varepsilon}\right)$, где $\varepsilon = 1$, можно применить случай 1 основной теоремы и сделать вывод, что решение — $T(n) = \Theta\left(n^2\right)$.

Теперь рассмотрим следующее рекуррентное соотношение:

$$T(n) = T(2n/3) + 1,$$

в котором $a = 1$, $b = 3/2$, $f(n) = 1$, а $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Здесь применим случай 2, поскольку $f(n) = \Theta\left(n^{\log_b a}\right) = \Theta(1)$, поэтому решение — $T(n) = \Theta(\lg n)$.

Для рекуррентного соотношения

$$T(n) = 3T(n/4) + n \lg n$$

выполняются условия $a = 3$, $b = 4$, $f(n) = n \lg n$, и $n^{\log_b a} = n^{\log_4 3} = O\left(n^{0.793}\right)$. Поскольку $f(n) = \Omega\left(n^{\log_4 3+\varepsilon}\right)$, где $\varepsilon \approx 0.2$, применяется случай 3 (если удастся показать выполнение условия регулярности для функции $f(n)$). При достаточно больших n условие $af(n/b) = 3(n/4)\lg(n/4) \leqslant (3/4)\, n \lg n = cf(n)$ выполняется при $c = 3/4$. Следовательно, согласно случаю 3, решение этого рекуррентного соотношения — $T(n) = \Theta(n \lg n)$.

К рекуррентному соотношению

$$T(n) = 2T(n/2) + n \lg n$$

основной метод неприменим, несмотря на то, что оно имеет надлежащий вид: $a = 2$, $b = 2$, $f(n) = n \lg n$, и $n^{\log_b a} = n$. Может показаться, что к нему применим случай 3, поскольку функция $f(n) = n \lg n$ асимптотически больше, чем $n^{\log_b a} = n$. Однако проблема заключается в том, что данная функция не *полиномиально* больше. Отношение $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ асимптотически меньше функции n^{ε} для любой положительной константы ε. Следовательно, рассматриваемое рекуррентное соотношение находится “в промежуточном состоянии” между случаями 2 и 3. (См. упражнение 4.4-2.)

Упражнения

4.3-1. С помощью основной теоремы найдите точные асимптотические границы следующих рекуррентных соотношений.

а) $T(n) = 4T(n/2) + n$.

б) $T(n) = 4T(n/2) + n^2$.

в) $T(n) = 4T(n/2) + n^3$.

4.3-2. Рекуррентное соотношение $T(n) = 7T(n/2) + n^2$ описывает время работы алгоритма A. Время работы альтернативного алгоритма A' выражается рекуррентным соотношением $T'(n) = aT'(n/4) + n^2$. Чему равно наибольшее целое значение a, при котором алгоритм A' работает асимптотически быстрее, чем алгоритм A.

4.3-3. Покажите с помощью основного метода, что $T(n) = \Theta(\lg n)$ является решением рекуррентного соотношения $T(n) = T(n/2) + \Theta(1)$, возникающего при анализе алгоритма бинарного поиска. (Алгоритм бинарного поиска описывается в упражнении 2.3-5.)

4.3-4. Можно ли применить основной метод к рекуррентному соотношению $T(n) = 4T(n/2) + n^2 \lg n$? Обоснуйте ваш ответ. Найдите асимптотическую верхнюю границу решения этого рекуррентного соотношения.

⋆ 4.3-5. Рассмотрим условие регулярности $af(n/b) \leqslant cf(n)$ для некоторой константы $c < 1$, являющееся частью случая 3 основной теоремы. Приведите примеры констант $a \geqslant 1$ и $b > 1$ и функции $f(n)$, которые удовлетворяют всем условиям случая 3 основной теоремы, кроме условия регулярности.

⋆ 4.4 Доказательство основной теоремы

В этом разделе приводится доказательство основной теоремы (теорема 4.1). Для применения самой теоремы понимать доказательство необязательно.

Доказательство состоит из двух частей. В первой части анализируется "основное" рекуррентное соотношение (4.5) с учетом упрощающего предположения, согласно которому $T(n)$ определена только для точных степеней числа $b > 1$, т.е. для $n = 1, b, b^2, \ldots$. В этой части представлены все основные идеи, необходимые для доказательства основной теоремы. Во второй части доказательство обобщается на множество всех натуральных n; кроме того, здесь при доказательстве аккуратно выполняются все необходимые округления с использованием функций "пол" и "потолок".

В данном разделе асимптотические обозначения будут несколько видоизменены: с их помощью будет описываться поведение функций, определенных только для целых степеней числа b, хотя согласно определению асимптотических обозначений неравенства должны доказываться для всех достаточно больших чисел, а не только для степеней числа b. Поскольку можно ввести новые асимптотические обозначения, применимые к множеству чисел $\{b^i : i = 0, 1, \ldots\}$, а не ко всему множеству неотрицательных целых чисел, упомянутое видоизменение несущественно.

Тем не менее, применяя асимптотические обозначения на суженной области определения, необходимо быть внимательным, чтобы не прийти к неверным выводам. Например, если доказано, что $T(n) = O(n)$, если n — степень двойки, то это еще не означает, что $T(n) = O(n)$ для всех n. Функция $T(n)$ может быть определена так:

$$T(n) = \begin{cases} n & \text{при } n = 1, 2, 4, 8, \ldots, \\ n^2 & \text{для остальных значений } n. \end{cases}.$$

В этом случае наилучшей верхней границей, как легко доказать, является $T(n) = O(n^2)$. Во избежание подобных ошибок, мы никогда не будем использовать асимптотические обозначения на ограниченной области определения функции, если только из контекста не будет вполне очевидно, что именно мы собираемся делать.

4.4.1 Доказательство теоремы для точных степеней

В первой части доказательства основной теоремы анализируется рекуррентное соотношение (4.5)

$$T(n) = aT(n/b) + f(n),$$

в предположении, что n — точная степень числа $b > 1$ (b — не обязательно целое число). Анализ производится путем доказательства трех лемм. В первой из них решение основного рекуррентного соотношения сводится к вычислению выражения, содержащего суммирование. Во второй лемме определяются границы этой суммы. В третьей лемме с помощью первых двух доказывается версия основной теоремы для случая, когда n — точная степень b.

Лемма 4.2. Пусть $a \geqslant 1$ и $b > 1$ — константы, а $f(n)$ — неотрицательная функция, определенная для точных степеней числа b. Определим функцию $T(n)$ на множестве точных степеней числа b с помощью такого рекуррентного соотношения:

$$T(n) = \begin{cases} \Theta(1) & \text{при } n = 1, \\ aT(n/b) + f(n) & \text{при } n = b^i, \end{cases}$$

где i — положительное целое число. Тогда получаем:

$$T(n) = \Theta\left(n^{\log_b a}\right) + \sum_{j=0}^{\log_b n - 1} a^j f\left(n/b^j\right). \tag{4.6}$$

Доказательство. Воспользуемся деревом рекурсии, представленным на рис. 4.3. Время выполнения, соответствующее корню дерева, равняется $f(n)$. Корень имеет a дочерних ветвей, время выполнения каждой из которых равно $f(n/b)$. (В ходе этих рассуждений, особенно для визуального представления дерева рекурсии, удобно считать число a целым, хотя это и не следует из каких либо математических соотношений.) Каждая из этих дочерних ветвей, в свою очередь, тоже имеет a дочерних ветвей, время выполнения которых равно $f\left(n/b^2\right)$. Таким образом, получается, что на втором уровне дерева имеется a^2 узлов. Обобщая эти рассуждения, можно сказать, что на j-м уровне находится a^j узлов, а время выполнения каждого из них равно $f\left(n/b^j\right)$. Время выполнения каждого листа равно $T(1) = \Theta(1)$, и все листья находятся на глубине $\log_b n$ уровня, поскольку $n/b^{\log_b n} = 1$. Всего же у дерева $a^{\log_b n} = n^{\log_b a}$ листьев.

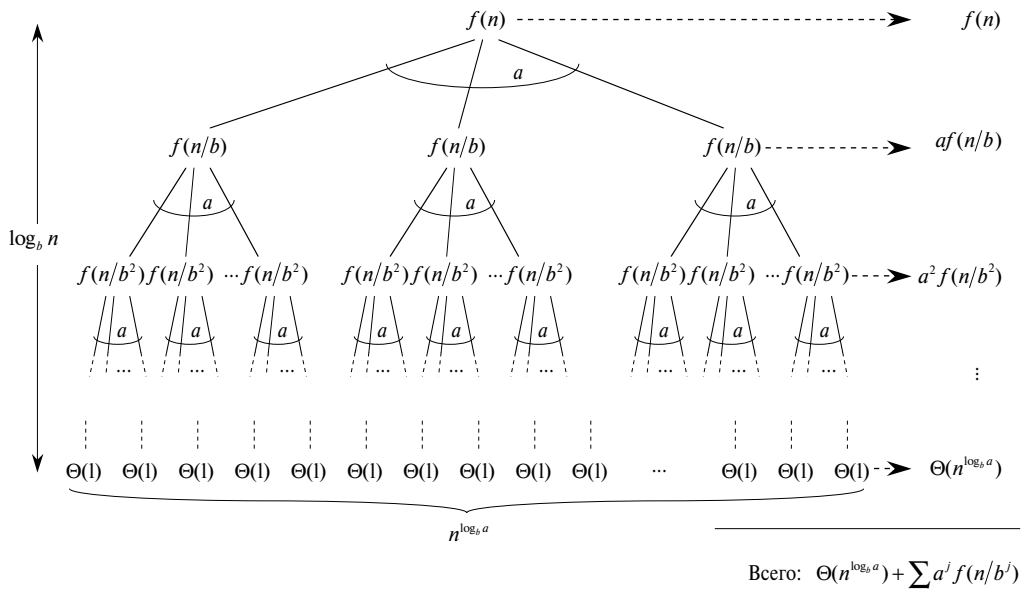


Рис. 4.3. Дерево рекурсии, генерируемое рекуррентным соотношением $T(n) = aT(n/b) + f(n)$. Дерево является полным a-арным деревом высотой $\log_b n$ с $n^{\log_b a}$ листьями. Время выполнения каждого уровня показано справа, а сумма этих величин выражается уравнением (4.6)

Уравнение (4.6) можно получить, суммируя время выполнения всех листьев дерева, как показано на рисунке. Время выполнения внутренних узлов уровня j равно $a^j f\left(n/b^j\right)$, так что полное время выполнения всех внутренних узлов можно записать так:

$$\sum_{j=0}^{\log_b n-1} a^j f\left(n/b^j\right).$$

В алгоритме разбиения, лежащем в основе анализируемого рекуррентного соотношения, эта сумма соответствует полному времени, затрачиваемому на разбиение поставленной задачи на вспомогательные подзадачи, и последующему объединению их решений. Суммарное время выполнения всех листьев, т.е. время решения всех $n^{\log_b a}$ вспомогательных задач с размером 1, равно $\Theta\left(n^{\log_b a}\right)$. ■

Три частных случая основной теоремы, выраженные в терминах дерева рекурсии, соответствуют ситуациям, когда полное время выполнения дерева 1) преимущественно определяется временем выполнения его листьев, 2) равномерно распределено по всем уровням дерева или 3) преимущественно определяется временем выполнения корня. Сумма в уравнении (4.6) описывает время, затрачиваемое на этапы разбиения и объединения в алгоритме разбиения, лежащем в основе

анализируемого рекуррентного соотношения. В следующей лемме обосновываются асимптотические оценки порядка роста этой суммы.

Лемма 4.3. Пусть $a \geqslant 1$ и $b > 1$ — константы, а $f(n)$ — неотрицательная функция, определенная для точных степеней числа b. Тогда функцию $g(n)$, определенную на множестве целых степеней числа b с помощью соотношения

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(n/b^j\right), \tag{4.7}$$

можно асимптотически оценить следующим образом.

1. Если для некоторой константы $\varepsilon > 0$ выполняется условие $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$, то $g(n) = O\left(n^{\log_b a}\right)$.
2. Если $f(n) = \Theta\left(n^{\log_b a}\right)$, то $g(n) = \Theta\left(n^{\log_b a} \lg n\right)$.
3. Если для некоторой константы $c < 1$ и всех $n \geqslant b$ справедливо соотношение $af(n/b) \leqslant cf(n)$, то $g(n) = \Theta(f(n))$.

Доказательство. В первом случае справедливо соотношение $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$, из которого следует, что $f\left(n/b^j\right) = O\left(\left(n/b^j\right)^{\log_b a - \varepsilon}\right)$. Подставив это равенство в уравнение (4.7), получим следующее соотношение:

$$g(n) = O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \varepsilon}\right). \tag{4.8}$$

Оценим сумму в O-обозначении. Для этого вынесем из-под знака суммирования постоянный множитель и выполним некоторые упрощения, в результате чего сумма преобразуется в возрастающую геометрическую прогрессию:

$$\begin{aligned}
\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \varepsilon} &= n^{\log_b a - \varepsilon} \sum_{j=0}^{\log_b n - 1} \left(\frac{ab^\varepsilon}{b^{\log_b a}}\right)^j = \\
&= n^{\log_b a - \varepsilon} \sum_{j=0}^{\log_b n - 1} (b^\varepsilon)^j = \\
&= n^{\log_b a - \varepsilon} \left(\frac{b^{\varepsilon \log_b n} - 1}{b^\varepsilon - 1}\right) = \\
&= n^{\log_b a - \varepsilon} \left(\frac{n^\varepsilon - 1}{b^\varepsilon - 1}\right).
\end{aligned}$$

Поскольку b и ε — константы, последнее выражение можно переписать в виде $n^{\log_b a-\varepsilon}O\left(n^{\varepsilon}\right)=O\left(n^{\log_b a}\right)$. Подставляя это выражение под знак суммы в уравнении (4.8), получим, что $g\left(n\right)=O\left(n^{\log_b a}\right)$, и тем самым доказываем случай 1.

В случае 2 делается предположение, что $f\left(n\right)=\Theta\left(n^{\log_b a}\right)$, поэтому $f\left(n/b^j\right)=\Theta\left(\left(n/b^j\right)^{\log_b a}\right)$. Подставляя это соотношение в уравнение (4.7), получаем:

$$g\left(n\right)=\Theta\left(\sum_{j=0}^{\log_b n-1}a^j\left(\frac{n}{b^j}\right)^{\log_b a}\right). \tag{4.9}$$

Как и в случае 1, оценим сумму, стоящую под знаком Θ, но на этот раз геометрическая прогрессия не получается. В самом деле, нетрудно убедиться, что все слагаемые суммы одинаковы:

$$\begin{aligned}\sum_{j=0}^{\log_b n-1}a^j\left(\frac{n}{b^j}\right)^{\log_b a}&=n^{\log_b a}\sum_{j=0}^{\log_b n-1}\left(\frac{a}{b^{\log_b a}}\right)^j=\\&=n^{\log_b a}\sum_{j=0}^{\log_b n-1}1=\\&=n^{\log_b a}\log_b n.\end{aligned}$$

Подставив это выражение для суммы в уравнение (4.9), получим соотношение

$$\begin{aligned}g\left(n\right)&=\Theta\left(n^{\log_b a}\log_b n\right)=\\&=\Theta\left(n^{\log_b a}\lg n\right),\end{aligned}$$

доказывающее случай 2.

Случай 3 доказывается аналогично. Поскольку функция $f\left(n\right)$ фигурирует в определении (4.7) функции $g\left(n\right)$, и все слагаемые функции $g\left(n\right)$ неотрицательные, можно заключить, что $g\left(n\right)=\Omega\left(f\left(n\right)\right)$ для точных степеней числа b. Согласно нашему предположению, для некоторой константы $c<1$ соотношение $af\left(n/b\right)\leqslant cf\left(n\right)$ справедливо для всех $n\geqslant b$, откуда $f\left(n/b\right)\leqslant\left(c/a\right)f\left(n\right)$. Итерируя j раз, получаем, что $f\left(n/b^j\right)\leqslant\left(c/a\right)^j f\left(n\right)$, или, что то же самое, $a^j f\left(n/b^j\right)\leqslant c^j f\left(n\right)$. После подстановки в уравнение (4.7) и некоторых упрощений получаем геометрическую прогрессию. В отличие от прогрессии из случая 1, члены данной прогрессии убывают (при рассмотрении приведенных преобразо-

ваний не забывайте, что c — константа):

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f\left(n/b^j\right) \leqslant \sum_{j=0}^{\log_b n-1} c^j f(n) \leqslant$$
$$\leqslant f(n) \sum_{j=0}^{\infty} c^j = f(n)\left(\frac{1}{1-c}\right) = O(f(n)).$$

Таким образом, можно прийти к заключению, что $g(n) = \Theta(f(n))$ для точных степеней числа b. Этим доказательством случая 3 завершается доказательство леммы. ■

Теперь можно приступить к доказательству основной теоремы в частном случае, когда n — точная степень числа b.

Лемма 4.4. Пусть $a \geqslant 1$ и $b > 1$ — константы, а $f(n)$ — неотрицательная функция, определенная для точных степеней числа b. Определим функцию $T(n)$ для точных степеней числа b с помощью следующего рекуррентного соотношения:

$$T(n) = \begin{cases} \Theta(1) & \text{при } n = 1, \\ aT(n/b) + f(n) & \text{при } n = b^i, \end{cases}$$

где i — положительное целое число. Тогда для этой функции можно дать следующую асимптотическую оценку (справедливую для точных степеней числа b).

1. Если для некоторой константы $\varepsilon > 0$ выполняется условие $f(n) = O\left(n^{\log_b a-\varepsilon}\right)$, то $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. Если $f(n) = \Theta\left(n^{\log_b a}\right)$, то $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.
3. Если для некоторой константы $\varepsilon > 0$ выполняется условие $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$, и если для некоторой константы $c < 1$ и всех достаточно больших n справедливо соотношение $af(n/b) \leqslant cf(n)$, то $T(n) = \Theta(f(n))$.

Доказательство. С помощью асимптотических границ, полученных в лемме 4.3, оценим сумму (4.6) из леммы 4.2. В случае 1 имеем:

$$T(n) = \Theta\left(n^{\log_b a}\right) + O\left(n^{\log_b a}\right) =$$
$$= \Theta\left(n^{\log_b a}\right),$$

в случае 2 —

$$T(n) = \Theta\left(n^{\log_b a}\right) + \Theta\left(n^{\log_b a} \lg n\right) =$$
$$= \Theta\left(n^{\log_b a} \lg n\right),$$

а в случае 3, поскольку $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$ —

$$T(n) = \Theta\left(n^{\log_b a}\right) + \Theta(f(n)) = \\ = \Theta(f(n)).$$

■

4.4.2 Учет округления чисел

Чтобы завершить доказательство основной теоремы, необходимо обобщить проведенный ранее анализ на случай, когда рекуррентное соотношение определено не только для точных степеней числа b, но и для всех целых чисел. Получить нижнюю границу для выражения

$$T(n) = aT(\lceil n/b \rceil) + f(n) \tag{4.10}$$

и верхнюю границу для выражения

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \tag{4.11}$$

не составляет труда, поскольку в первом случае для получения нужного результата можно использовать неравенство $\lceil n/b \rceil \geqslant n/b$, а во втором — неравенство $\lfloor n/b \rfloor \leqslant n/b$. Чтобы получить нижнюю границу рекуррентного соотношения (4.11), необходимо применить те же методы, что и при нахождении верхней границы для рекуррентного соотношения (4.10), поэтому здесь будет показан поиск только последней.

Дерево рекурсии, представленное на рис. 4.3, модифицируется и приобретает вид, показанный на рис. 4.4. По мере продвижения по дереву рекурсии вниз, мы получаем следующую рекурсивную последовательность аргументов:

$$\begin{matrix} n, \\ \lceil n/b \rceil, \\ \lceil \lceil n/b \rceil / b \rceil, \\ \lceil \lceil \lceil n/b \rceil / b \rceil / b \rceil, \\ \vdots \end{matrix} .$$

Обозначим j-й элемент этой последовательности как n_j, где

$$n_j = \begin{cases} n & \text{при } j = 0, \\ \lceil n_{j-1}/b \rceil & \text{при } j > 0. \end{cases} \tag{4.12}$$

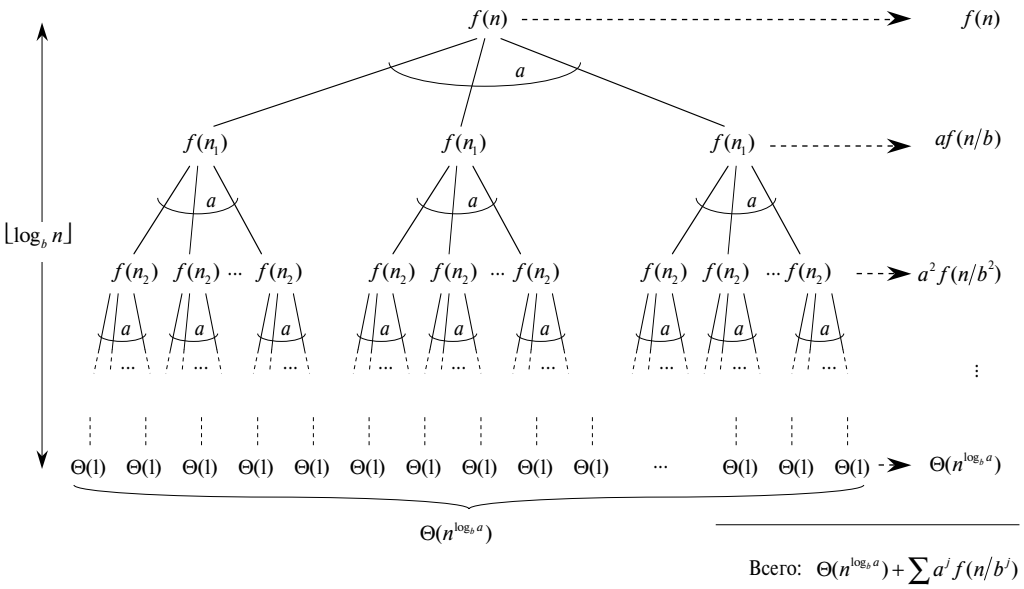


Рис. 4.4. Дерево рекурсии, определенное рекуррентным соотношением $T(n) = aT(\lceil n/b \rceil) + f(n)$. Аргумент рекурсии n_j определяется уравнением (4.12)

Сейчас наша цель — определить глубину k, такую что n_k — константа. Используя неравенство $\lceil x \rceil \leqslant x + 1$, получаем:

$$\begin{aligned} n_0 &\leqslant n, \\ n_1 &\leqslant \frac{n}{b} + 1, \\ n_2 &\leqslant \frac{n}{b^2} + \frac{1}{b} + 1, \\ n_3 &\leqslant \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1, \\ &\vdots \end{aligned}$$

В общем случае получаем:

$$n_j \leqslant \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} < \frac{n}{b^j} + \sum_{i=0}^{\infty} \frac{1}{b^i} = \frac{n}{b^j} + \frac{b}{b-1}.$$

Полагая, что $j = \lfloor \log_b n \rfloor$, мы получим:

$$n_{\lfloor \log_b n \rfloor} < \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-1} < \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} = \frac{n}{n/b} + \frac{b}{b-1} = b + \frac{b}{b-1} = O(1),$$

откуда видно, что на уровне $\lfloor \log_b n \rfloor$ размер задачи не превышает постоянную величину.

Из рис. 4.4 видно, что выполняется соотношение

$$T(n) = \Theta\left(n^{\log_b a}\right) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \tag{4.13}$$

того же вида, что и уравнение (4.6) (с тем отличием, что здесь n — произвольное целое число, а не точная степень числа b).

Теперь можно приступить к оценке суммы

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j), \tag{4.14}$$

фигурирующей в уравнении (4.13). Эта сумма будет оцениваться аналогично тому, как это делалось при доказательстве леммы 4.3. Начнем с рассмотрения случая 3. Если для $n > b + b/(b-1)$ выполняется соотношение $af(\lceil n/b \rceil) \leqslant cf(n)$, где $c < 1$ — константа, то отсюда следует, что $a^j f(n_j) \leqslant c^j f(n)$. Таким образом, сумму в уравнении (4.14) можно оценить точно так же, как это было сделано в лемме 4.3. В случае 2 выполняется условие $f(n) = \Theta\left(n^{\log_b a}\right)$. Если мы сможем показать, что $f(n_j) = O\left(n^{\log_b a}/a^j\right) = O\left(\left(n/b^j\right)^{\log_b a}\right)$, то доказательство случая 2 леммы 4.3 будет завершено. Заметим, что из неравенства $j \leqslant \lfloor \log_b n \rfloor$ следует неравенство $b^j/n \leqslant 1$. Наличие границы $f(n) = \Theta\left(n^{\log_b a}\right)$ подразумевает, что существует такая константа $c > 0$, что при достаточно больших n_j выполняются такие соотношения:

$$\begin{aligned}
f(n_j) &\leqslant c\left(\frac{n}{b_j} + \frac{b}{b-1}\right)^{\log_b a} = \\
&= c\left(\frac{n}{b^j}\left(1 + \frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} = \\
&= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \left(\frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \leqslant \\
&\leqslant c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b}{b-1}\right)^{\log_b a} = \\
&= O\left(\frac{n^{\log_b a}}{a^j}\right),
\end{aligned}$$

поскольку $c(1 + b/(b-1))^{\log_b a}$ — константа. Таким образом, случай 2 доказан. Доказательство случая 1 почти идентично только что рассмотренному. Главное —

доказать справедливость границы $f(n_j) = O\left(n^{\log_b a - \varepsilon}\right)$. Это делается аналогично доказательству в случае 2, хотя алгебраические выкладки при этом оказываются несколько более сложными.

Итак, мы доказали соблюдение верхних границ в основной теореме для всех целых n. Соблюдение нижних границ доказывается аналогично.

Упражнения

⋆ 4.4-1. Приведите простое и точное выражение для n_j в уравнении (4.12) для случая, когда b — положительное целое число (а не произвольное действительное).

⋆ 4.4-2. Покажите, что если выполняется соотношение $f(n) = \Theta\left(n^{\log_b a} \lg^k n\right)$, где $k \geqslant 0$, то решение основного рекуррентного соотношения $-T(n) = = \Theta\left(n^{\log_b a} \lg^{k+1} n\right)$. Для простоты рассмотрите только случай целых степеней b.

⋆ 4.4-3. Покажите, что в случае 3 основной теоремы одно из условий излишнее в том плане, что из условия регулярности $af(n/b) \leqslant cf(n)$ для некоторой константы $c < 1$ следует, что существует константа $\varepsilon > 0$, такая что $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$.

Задачи

4-1. **Примеры рекуррентных соотношений**

Определите верхнюю и нижнюю асимптотические границы функции $T(n)$ для каждого из представленных ниже рекуррентных соотношений. Считаем, что при $n \leqslant 2T(n)$ — константа. Попытайтесь сделать эту оценку как можно более точной и обоснуйте свой ответ.

а) $T(n) = 2T(n/2) + n^3$.
б) $T(n) = T(9n/10) + n$.
в) $T(n) = 16T(n/4) + n^2$.
г) $T(n) = 7T(n/3) + n^2$.
д) $T(n) = 7T(n/2) + n^2$.
е) $T(n) = 2T(n/4) + \sqrt{n}$.
ж) $T(n) = T(n-1) + n$.
з) $T(n) = T(\sqrt{n}) + 1$.

4-2. **Поиск отсутствующего целого числа**

Массив $A[1..n]$ содержит все целые числа от 0 до n за исключением одного. Отсутствующее число можно легко определить за время $O(n)$,

располагая вспомогательным массивом $B[1..n]$, предназначенным для записи имеющихся чисел из массива A. Однако в этой задаче мы лишены средств, позволяющих получить доступ к целому числу из массива A посредством одной операции. Элементы массива A представлены в двоичной системе счисления, и единственная операция, с помощью которой можно осуществлять к ним доступ, это "извлечение j-го бита элемента $A[i]$", которая выполняется в течение фиксированного времени.

Покажите, что, располагая только этой операцией, отсутствующее целое число все же можно определить за время $O(n)$.

4-3. **Время, затрачиваемое на передачу констант**

В этой книге предполагается, что передача параметров при вызове процедуры занимает фиксированное время, даже если передается N-элементный массив. Для большинства вычислительных систем это предположение справедливо, поскольку передается не сам массив, а указатель на него. В данной задаче исследуются три стратегии передачи параметров.

i) Массив передается посредством указателя. Время выполнения этой операции $T = \Theta(1)$.

ii) Массив передается путем копирования. Время выполнения этой операции $T = \Theta(N)$, где N — размер массива.

iii) Массив передается путем копирования только некоторого поддиапазона, к которому обращается вызываемая процедура. Время передачи подмассива $A[p..q]$ равно $T = \Theta(q - p + 1)$.

а) Рассмотрите рекурсивный алгоритм бинарного поиска, предназначенный для нахождения числа в отсортированном массиве (см. упражнение 2.3-5). Приведите рекуррентные соотношения, описывающие время бинарного поиска в наихудшем случае, если массивы передаются с помощью каждого из описанных выше методов, и дайте точные верхние границы решений этих рекуррентных соотношений. Пусть размер исходной задачи равен N, а размер подзадачи — n.

б) Выполните задание части *а*) для алгоритма MERGE_SORT, описанного в разделе 2.3.1.

4-4. **Примеры рекуррентных соотношений**

Дайте верхнюю и нижнюю асимптотические оценки функции $T(n)$ в каждом из приведенных ниже рекуррентных соотношений. Предполагается, что для достаточно малых значений n $T(n)$ — постоянная величина. Постарайтесь, чтобы оценки были как можно более точными и обоснуйте ответы.

а) $T(n) = 3T(n/2) + n \lg n$.

б) $T(n) = 5T(n/5) + n/\lg n$.

в) $T(n) = 4T(n/2) + n^2\sqrt{n}$.

г) $T(n) = 3T(n/3 + 5) + n/2$.

д) $T(n) = 2T(n/2) + n/\lg n$.

е) $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

ж) $T(n) = T(n-1) + 1/n$.

з) $T(n) = T(n-1) + \lg n$.

и) $T(n) = T(n-2) + 2\lg n$.

к) $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

4-5. **Числа Фибоначчи**

В этой задаче раскрываются свойства чисел Фибоначчи, определенных с помощью рекуррентного соотношения (3.21). Воспользуемся для решения рекуррентного соотношения Фибоначчи методом генераторных функций. Определим ***производящую функцию*** (или ***формальный степенной ряд***) $\mathcal{F}$ как

$$\mathcal{F} = \sum_{i=0}^{\infty} F_i z^i = 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \ldots,$$

где F_i — i-е число Фибоначчи.

а) Покажите, что $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

б) Покажите, что

$$\mathcal{F}(z) = \frac{z}{1 - z - z^2} = \frac{z}{(1 - \phi z)\left(1 - \widehat{\phi} z\right)} = \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \phi z} - \frac{1}{1 - \widehat{\phi} z}\right),$$

где

$$\phi = \frac{1 + \sqrt{5}}{2} = 1.61803\ldots$$

и

$$\widehat{\phi} = \frac{1 - \sqrt{5}}{2} = -0.61803\ldots.$$

в) Покажите, что $\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}\left(\phi^i - \widehat{\phi}^i\right) z^i$.

г) Докажите, что $F_i = \phi^i/\sqrt{5}$ (округленное до ближайшего целого) для всех $i > 0$. (*Указание*: обратите внимание на то, что $\left|\widehat{\phi}\right| < 1$.)

д) Докажите, что $F_{i+2} \geqslant \phi^i$ для $i \geqslant 0$.

4-6. **Тестирование СБИС**

В распоряжении профессора есть n предположительно идентичных СБИС[1], которые в принципе способны тестировать друг друга. В тестирующее приспособление за один раз можно поместить две микросхемы. При этом каждая микросхема тестирует соседнюю и выдает отчет о результатах тестирования. Исправная микросхема всегда выдает правильные результаты тестирования, а результатам неисправной микросхемы доверять нельзя. Таким образом, возможны четыре варианта результатов тестирования, приведенные в таблице ниже.

Отчет микросхемы А	Отчет микросхемы Б	Вывод
Б исправна	А исправна	Либо обе микросхемы исправны, либо обе неисправны
Б исправна	А неисправна	Неисправна хотя бы одна микросхема
Б неисправна	А исправна	Неисправна хотя бы одна микросхема
Б неисправна	А неисправна	Неисправна хотя бы одна микросхема

а) Покажите, что если неисправно более $n/2$ микросхем, то с помощью какой бы то ни было стратегии, основанной на попарном тестировании, профессор не сможет точно определить, какие микросхемы исправны. (Предполагается, что неисправные микросхемы не смогут тайно сговориться, чтобы обмануть профессора.)

б) Рассмотрим задачу о поиске одной исправной микросхемы среди n микросхем, если предполагается, что исправно более половины всех микросхем. Покажите, что $\lfloor n/2 \rfloor$ попарных тестирований достаточно для сведения этой задачи к подзадаче, размер которой приблизительно в два раза меньше.

в) Покажите, что можно найти все исправные микросхемы с помощью $\Theta(n)$ попарных тестирований, если предполагается, что более половины микросхем исправны. Сформулируйте и решите рекуррентное соотношение, описывающее количество тестирований.

[1] СБИС — это аббревиатура от “сверхбольшие интегральные схемы” (микросхемы, созданные по технологии, которая используется при производстве большинства современных микропроцессоров).

4-7. **Массивы Монжа**

Массив A, состоящий из действительных чисел и имеющий размер $m \times n$, называется ***массивом Монжа*** (Monge array), если для всех i, j, k и l, таких что $1 \leqslant i < k \leqslant m$ и $1 \leqslant j < l \leqslant n$, выполняется соотношение

$$A[i,j] + A[k,l] \leqslant A[i,l] + A[k,j].$$

Другими словами, при любом выборе двух срок и двух столбцов из массива Монжа (при этом элементы массива на пересечении выбранных строк и столбцов образуют прямоугольник) сумма элементов в левом верхнем и правом нижнем углах полученного прямоугольника не превышает сумму элементов в левом нижнем и правом верхнем его углах. Приведем пример массива Монжа:

$$\begin{array}{ccccc}
10 & 17 & 13 & 28 & 23 \\
17 & 22 & 16 & 29 & 23 \\
24 & 28 & 22 & 34 & 24 \\
11 & 13 & 6 & 17 & 7 \\
45 & 44 & 32 & 37 & 23 \\
36 & 33 & 19 & 21 & 6 \\
75 & 66 & 51 & 53 & 34
\end{array}$$

а) Докажите, что массив является массивом Монжа тогда и только тогда, когда для всех $i = 1, 2, \ldots, m-1$ и $j = 1, 2, \ldots, n-1$ справедливо соотношение

$$A[i,j] + A[i+1,j+1] \leqslant A[i,j+1] + A[i+1,j].$$

(*Указание*: для прямого доказательства воспользуйтесь методом математической индукции; индукцию нужно провести отдельно для строк и столбцов.)

б) Приведенный ниже массив не является массивом Монжа. Измените в нем один элемент таким образом, чтобы он стал массивом Монжа. (*Указание*: воспользуйтесь результатами части a.)

$$\begin{array}{cccc}
37 & 23 & 22 & 32 \\
21 & 6 & 7 & 10 \\
53 & 34 & 30 & 31 \\
32 & 13 & 9 & 6 \\
43 & 21 & 15 & 8
\end{array}$$

в) Пусть $f(i)$ — индекс столбца, содержащего крайний слева минимальный элемент в строке i. Докажите, что для любого массива Монжа размером $m \times n$ справедливо соотношение $f(1) \leqslant f(2) \leqslant \leqslant \cdots \leqslant f(m)$.

г) Ниже описан алгоритм разбиения, предназначенный для вычисления крайнего слева минимального элемента в каждой строке, входящей в состав массива Монжа A размером $m \times n$.

> Постройте подматрицу A' матрицы A, состоящую из ее нечетных строк. С помощью рекурсивной процедуры найдите в каждой строке матрицы A' крайний слева минимальный элемент. Затем найдите крайний слева минимальный элемент среди четных строк матрицы A.

Объясните, как найти крайний слева минимальный элемент среди нечетных строк матрицы A (предполагается, что крайний слева минимальный элемент среди четных столбцов этой матрицы известен) за время $O(m+n)$.

д) Запишите рекуррентное соотношение, описывающее время работы алгоритма из части *г*. Покажите, что его решение — $O(m + n \log m)$.

Заключительные замечания

Рекуррентные соотношения изучались еще с 1202 года, со времен Леонардо Фибоначчи (L. Fibonacci). А. де Муавр (A. De Moivre) впервые использовал метод производящих функций (см. задачу 4-5) для решения рекуррентных соотношений. Основной метод был принят на вооружение после появления работы Бентли (Bentley), Хакена (Haken) и Сакса (Saxe) [41], в которой был предложено расширение метода, обоснованного в упражнении 4.4-2. Кнут (Knuth) и Лью (Liu) [205] показали, каким образом можно решать линейные рекуррентные соотношения с использованием производящих функций. Подробное обсуждение методов решения рекуррентных соотношений можно найти в книгах Пурдома (Purdom) и Брауна (Brown) [252], а также Грехема (Graham), Кнута и Паташника (Patashnik) [132].

Некоторые исследователи, в число которых входят Акра (Akra) и Баззи (Bazzi) [13], Роура (Roura) [262] и Верма (Verma) [306], предложили методы решения более общих рекуррентных соотношений для алгоритмов разбиения. Ниже представлены результаты Акра и Баззи. Их метод подходит для решения рекуррентных соотношений вида

$$T(n) = \sum_{i=1}^{k} a_i T(\lfloor n/b_i \rfloor) + f(n), \tag{4.15}$$

где $k \geqslant 1$; все коэффициенты a_i положительны, и в сумме содержится хотя бы одно слагаемое; все b_i больше 1; функция $f(n)$ ограниченная, положительная и неубывающая. Кроме того, для любой константы $c > 1$ существуют константы n_0 и $d > 0$, такие что для всех $n > n_0$ справедливо соотношение $f(n/c) \geqslant df(n)$. Этот метод позволяет найти решение рекуррентного соотношения $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + O(n)$, для которого неприменим основной метод. Чтобы решить рекуррентное соотношение (4.15), сначала найдем такое значение p, при котором $\sum_{i=1}^{k} a_i b_i^{-p} = 1$. (Такое p всегда существует, причем оно всегда единственное и положительное.) Решением рекуррентного соотношения является

$$T(n) = \Theta(n^p) + \Theta\left(n^p \int_{n'}^{n} \frac{f(x)}{x^{p+1}} dx\right),$$

где n' — достаточно большая константа. Метод Акры-Баззи на практике может оказаться слишком сложным в применении, однако он позволяет решать рекуррентные соотношения, моделирующие разбиение задачи на существенно неравные подзадачи. Основной метод более прост в использовании, однако он применим только в случае подзадач одинаковых размеров.

ГЛАВА 5

Вероятностный анализ и рандомизированные алгоритмы

Эта глава знакомит читателя с вероятностным анализом и рандомизированными алгоритмами. Тем, кто не знаком с основами теории вероятности, следует обратиться к приложению В, в котором представлен обзор этого материала. В данной книге мы будем несколько раз возвращаться к вероятностному анализу и рандомизированным алгоритмам.

5.1 Задача о найме сотрудника

Предположим, предприниматель хочет взять на работу нового офис-менеджера. Сначала он попытался найти подходящую кандидатуру самостоятельно, но неудачно, поэтому решил обратиться в агентство по трудоустройству. Согласно договоренности, агентство должно присылать работодателю по одному кандидату в день. Предприниматель проводит с каждым из этих кандидатов собеседование, после чего принимает окончательное решение, брать его на работу или нет. За каждого направленного ему кандидата предприниматель платит агентству небольшую сумму. Однако наем выбранного кандидата на работу фактически стоит дороже, поскольку при этом необходимо уволить офис-менеджера, работающего в данное время, а также уплатить агентству более значительную сумму за выбранного кандидата. Предприниматель пытается сделать так, чтобы должность всегда занимал наиболее достойный из всех кандидатов. Как только квалификация очередного претендента окажется выше квалификации офис-менеджера, который работает в данное время, этот офис-менеджер будет уволен, а его место займет более

подходящий кандидат. Работодатель готов понести все необходимые расходы, но хочет оценить, во что обойдется ему подбор нового сотрудника.

В представленной ниже процедуре HIRE_ASSISTANT эта стратегия найма представлена в виде псевдокода. В ней предполагается, что кандидаты на должность офис-менеджера пронумерованы от 1 до n. Предполагается также, что после собеседования с i-м кандидатом есть возможность определить, является ли он лучшим, чем все предыдущие. В процессе инициализации процедура создает фиктивного кандидата номер 0, квалификация которого ниже квалификации всех остальных:

```
HIRE_ASSISTANT(n)
1  Best ← 0   ▷ Кандидат 0 — наименее квалифицированный
              ▷ фиктивный кандидат
2  for i ← 1 to n
3      do Собеседование с кандидатом i
4         if кандидат i лучше кандидата Best
5           then Best ← i
6                Нанимаем кандидата i
```

Модель стоимости этой задачи отличается от модели, описанной в главе 2. Если ранее в качестве стоимости алгоритма мы рассматривали время его работы, то теперь нас интересует денежная сумма, затрачиваемая на собеседование и наем при использовании данного алгоритма найма работника. На первый взгляд может показаться, что анализ стоимости этого алгоритма очень сильно отличается, например, от анализа времени работы алгоритма сортировки методом слияния. Однако оказывается, что при анализе стоимости и времени выполнения применяются одинаковые аналитические методы. В обоих случаях подсчитывается, сколько раз выполняется та или иная операция.

Обозначим стоимость собеседования через c_i, а стоимость найма — через c_h. Пусть m — количество нанятых сотрудников. Тогда полная стоимость, затраченная при работе этого алгоритма, равна $O(nc_i + mc_h)$. Независимо от того, сколько сотрудников было нанято, интервью нужно провести с n кандидатами, поэтому суммарная стоимость всех интервью является фиксированной и равна nc_i. Таким образом, нас интересует анализ величины mc_h — стоимости найма, которая, как нетрудно понять, меняется при каждом выполнении алгоритма.

Этот сценарий служит моделью распространенной вычислительной парадигмы. Часто встречаются ситуации, когда нужно найти максимальное или минимальное значение последовательности, для чего каждый ее элемент сравнивается с текущим “претендентом” на звание подходящего. Задача о найме моделирует частоту, с которой придется заново присваивать метку “победителя” текущему элементу.

Анализ наихудшего случая

В наихудшем случае работодателю приходится нанимать каждого кандидата, с которым проведено собеседование. Эта ситуация возникает, когда кандидаты приходят в порядке возрастания их квалификации. При этом процедура найма повторяется n раз, а полная стоимость найма равна $O(nc_h)$.

Однако маловероятно, чтобы кандидаты поступали в указанном порядке. Фактически мы не имеем представления о том, в каком порядке они будут приходить, и никак не можем повлиять на этот порядок. Таким образом, возникает закономерный вопрос, чего следует ожидать в типичном (или среднем) случае.

Вероятностный анализ

Вероятностный анализ — это анализ задачи, при котором используются вероятности тех или иных событий. Чаще всего он используется при определении времени работы алгоритма. Иногда такой анализ применяется и для оценки других величин, таких как стоимость найма в процедуре HIRE_ASSISTANT. Для проведения вероятностного анализа необходимо располагать знаниями (или сделать предположение) о распределении входных данных. При этом в ходе анализа алгоритма вычисляется математическое ожидание времени его работы. На эту величину влияет распределение входных величин, поэтому производится усреднение времени работы по всем возможным входным данным.

Делая предположение о распределении входных величин, нужно быть очень осторожным. В некоторых задачах такие предположения вполне оправданны и позволяют воспользоваться вероятностным анализом как методом разработки эффективных алгоритмов и как средством для более глубокого понимания задачи. В других задачах разумное описание распределения входных величин не удается, и в таких случаях вероятностный анализ неприменим.

В задаче о найме сотрудника можно предположить, что претенденты приходят на собеседование в случайном порядке. Что это означает в контексте рассматриваемой задачи? Предполагается, что любых двух кандидатов можно сравнить и решить, кто из них квалифицированнее, т.е. в принципе всех кандидатов можно расположить в определенном порядке. (Определение полностью упорядоченных множеств приведено в приложении Б.) Таким образом, каждому кандидату можно присвоить уникальный порядковый номер от 1 до n. Назовем операцию присвоения порядкового номера i-му претенденту $rank(i)$ и примем соглашение, что более высокий ранг соответствует специалисту с более высокой квалификацией. Упорядоченное множество $\langle rank(1), rank(2), \ldots, rank(n)\rangle$ является перестановкой множества $\langle 1, 2, \ldots, n\rangle$. Утверждение, что кандидаты приходят на собеседование в случайном порядке, эквивалентно утверждению, что вероятность любого порядка рангов одинакова и равна количеству перестановок чисел от 1 до n, т.е. $n!$. Другими словами, ранги образуют ***случайную равновероятную пере-***

становку, т.е. каждая из $n!$ возможных перестановок появляется с одинаковой вероятностью.

Вероятностный анализ задачи о найме сотрудника приведен в разделе 5.2.

Рандомизированные алгоритмы

Чтобы провести вероятностный анализ, нужно иметь некоторые сведения о распределении входных данных. Во многих случаях мы знаем о них очень мало. Даже если об этом распределении что-то известно, может случиться так, что знания, которыми мы располагаем, нельзя численно смоделировать. Несмотря на это вероятность и случайность часто можно использовать в качестве инструмента при разработке и анализе алгоритмов, путем придания случайного характера части алгоритма.

В задаче о найме сотрудника все выглядит так, как будто кандидатов присылают на собеседование в случайном порядке, но на самом деле у нас нет никакой возможности узнать, так ли это на самом деле. Поэтому, чтобы разработать рандомизированный алгоритм для этой задачи, необходимо повысить степень контроля над порядком поступления претендентов на должность. Внесем в модель небольшие изменения. Допустим, бюро по трудоустройству подобрало n кандидатов. Работодатель договаривается, чтобы ему заранее прислали список кандидатов, и каждый день самостоятельно случайным образом выбирает, с кем из претендентов проводить собеседование. Несмотря на то, что работодатель по-прежнему ничего не знает о претендентах на должность, задача существенно изменилась. Теперь не нужно гадать, будет ли очередность кандидатов случайной; вместо этого мы взяли этот процесс под свой контроль и сами сделали случайным порядок проведения собеседований.

В общем случае алгоритм называется ***рандомизированным*** (randomized), если его поведение определяется не только набором входных величин, но и значениями, которые выдает ***генератор случайных чисел***. Будем предполагать, что в нашем распоряжении имеется генератор дискретных случайных чисел RANDOM. При вызове процедура $\text{RANDOM}(a, b)$ возвращает целое число, принадлежащее интервалу от a до b и равномерно распределенное в этом интервале. Например, $\text{RANDOM}(0, 1)$ с вероятностью $1/2$ выдает 0 и с той же вероятностью — 1. В результате вызова $\text{RANDOM}(3, 7)$ числа $3, 4, 5, 6$ и 7 возвращаются с вероятностью $1/5$. Вероятность возврата процедурой RANDOM любого целого числа не зависит от того, какие числа были возвращены в предыдущем вызове этой процедуры. Процедуру RANDOM можно представить в виде рулетки с $(b - a + 1)$ делениями. На практике большинство сред программирования предоставляют в распоряжение программиста ***генератор псевдослучайных чисел***, т.е. детерминированный алгоритм, который возвращающий числа, которые ведут себя при статистическом анализе как случайные.

Упражнения

5.1-1. Покажите, что из предположения о том, что в строке 4 алгоритма HIRE_ASSISTANT всегда можно определить, какой кандидат лучше, следует, что мы знаем общий порядок рангов кандидатов.

⋆ 5.1-2. Опишите реализацию процедуры RANDOM(a, b), которая может использовать только один вызов — процедуры RANDOM$(0, 1)$. Чему равно математическое ожидание времени работы вашей реализации и как оно зависит от a и b?

⋆ 5.1-3. Предположим, что нам надо выводить 0 и 1 с вероятностью 50%. В нашем распоряжении имеется процедура BIASED_RANDOM, которая с вероятностью p выдает 0, и с вероятностью $1 - p$ — число 1; значение p нам неизвестно. Сформулируйте алгоритм, использующий в качестве подпрограммы процедуру BIASED_RANDOM и возвращающий равномерно распределенные числа 0 и 1, т.е. вероятность вывода каждого из них равна 50%. Чему равно математическое ожидание времени работы такой процедуры и как оно зависит от p?

5.2 Индикаторная случайная величина

При анализе многих алгоритмов, в том числе того, с помощью которого решается задача о найме сотрудника, будет использоваться индикаторная случайная величина. Она предоставляет удобный метод перехода от вероятности к математическому ожиданию. Предположим, что в нашем распоряжении имеется пространство выборки S и событие A. Тогда ***индикаторная случайная величина*** (indicator random variable) $\mathrm{I}\{A\}$, связанная с событием A, определяется так:

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{если событие } A \text{ произошло,} \\ 0 & \text{если событие } A \text{ не произошло.} \end{cases} \tag{5.1}$$

В качестве примера определим математическое ожидание того, что при подбрасывании монеты выпадет орел. Пространство событий в этом случае имеет простой вид $S = \{H, T\}$, где вероятности выпадения орла (это событие обозначено как H (head)) и решки (T (tail)) равны: $\Pr\{H\} = \Pr\{T\} = 1/2$. Далее, можно определить индикаторную случайную величину X_H, связанную с выпадением орла, т.е. событием H. Эта величина подсчитывает количество выпадений орла. Если выпадает орел, она равна 1, а если решка — 0. Запишем это с помощью формальных обозначений:

$$X_H = \mathrm{I}\{H\} = \begin{cases} 1 & \text{если выпал орел,} \\ 0 & \text{если выпала решка.} \end{cases}$$

Математическое ожидание того, что выпадет орел, равняется математическому ожиданию индикаторной величины X_H:

$$\begin{aligned}\mathrm{E}[X_H] &= \mathrm{E}[\mathrm{I}\{H\}] = \\ &= 1 \cdot \Pr\{H\} + 0 \cdot \Pr\{T\} = \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) = \\ &= 1/2.\end{aligned}$$

Таким образом, математическое ожидание того, что выпадет орел, равно $1/2$. Как показано в приведенной ниже лемме, математическое ожидание индикаторной случайной величины, связанной с событием A, равно вероятности этого события.

Лемма 5.1. Пусть имеется пространство событий S и событие A, принадлежащее этому пространству, и пусть $X_A = \mathrm{I}\{A\}$. Тогда $\mathrm{E}[X_A] = \Pr\{A\}$.

Доказательство. Согласно определению (5.1) индикаторной случайной величины и определению математического ожидания, имеем:

$$\begin{aligned}\mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] = \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\left\{\bar{A}\right\} = \\ &= \Pr\{A\}.\end{aligned}$$

где $\bar{A}$ обозначает $S - A$, т.е. дополнение события A. ■

Хотя индикаторные случайные величины могут показаться неудобными в применении, например, при вычислении математического ожидания выпадений орла при бросании монеты, однако они оказываются полезны при анализе ситуаций, в которых делаются повторные испытания. Например, индикаторная случайная величина позволяет простым путем получить результат в уравнении (B.36). В этом уравнении количество выпадений орла при n бросаниях монеты вычисляется путем отдельного рассмотрения вероятности того события, что орел выпадет 0, 1, 2 и т.д. раз. В уравнении (B.37) предлагается более простой метод, в котором, по сути, неявно используется индикаторные случайные величины. Чтобы было понятнее, обозначим через X_i индикаторную случайную величину, связанную с событием, когда при i-м выбрасывании выпадает орел: $X_i = \mathrm{I}\{\text{При } i\text{-м броске выпал орел}\}$. Пусть X — случайная величина, равная общему количеству выпадений орла при n бросаниях монеты. Тогда

$$X = \sum_{i=1}^{n} X_i.$$

Нам надо вычислить математическое ожидание выпадения орла. Для этого применим операцию математического ожидания к обеим частям приведенного выше уравнения:

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right].$$

Левая часть полученного уравнения представляет собой математическое ожидание суммы n случайных величин. Математическое ожидание каждой из этих случайных величин легко вычисляется при помощи леммы 5.1. Пользуясь уравнением (B.20), выражающим свойство линейности математического ожидания, математическое ожидание суммы случайных величин можно выразить как сумму математических ожиданий каждой величины. Благодаря линейности математического ожидания использование индикаторной случайной величины становится мощным аналитическим методом, который применим даже в том случае, когда между случайными величинами имеется зависимость. Теперь мы можем легко вычислить математическое ожидание количества выпадений орла:

$$\begin{aligned}\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \\ &= \sum_{i=1}^{n} \mathrm{E}[X_i] = \\ &= \sum_{i=1}^{n} \frac{1}{2} = \\ &= \frac{n}{2}.\end{aligned}$$

Таким образом, индикаторные случайные величины значительно упростили вычисления по сравнению с методом, использующимся в уравнении (B.36). Вы еще не раз встретитесь с этими величинами в данной книге.

Анализ задачи о найме сотрудника с помощью индикаторных случайных величин

Вернемся к задаче о найме сотрудника, в которой нужно вычислить математическое ожидание события, соответствующего найму нового менеджера. Чтобы провести вероятностный анализ, предположим, что кандидаты приходят на собеседование в случайном порядке (этот вопрос уже обсуждался в предыдущем разделе; в разделе 5.3 будет показано, каким образом можно обойтись без этого предположения). Пусть X — случайная величина, значение которой равно количеству наймов нового офис-менеджера. Далее можно воспользоваться определением

математического ожидания (уравнение B.19) и получить соотношение

$$\mathrm{E}[X] = \sum_{x=1}^{n} x \Pr\{X = x\},$$

но эти вычисления окажутся слишком громоздкими. Вместо этого воспользуемся индикаторными случайными величинами, благодаря чему вычисления значительно упростятся.

Чтобы воспользоваться индикаторными случайными величинами, вместо вычисления величины $\mathrm{E}[X]$ путем определения одного значения, связанного с числом наймов нового менеджера, определим n величин, связанных с наймом конкретных кандидатов. В частности, пусть X_i — индикаторная случайная величина, связанная с событием найма i-го кандидата. Таким образом,

$$X_i = \mathrm{I}\{i\text{-й кандидат нанят}\} = \begin{cases} 1 & \text{если } i\text{-й кандидат нанят,} \\ 0 & \text{если } i\text{-й кандидат отклонен,} \end{cases} \tag{5.2}$$

и

$$X = X_1 + X_2 + \cdots + X_n. \tag{5.3}$$

Из леммы 5.1 следует, что

$$\mathrm{E}[X_i] = \Pr\{i\text{-й кандидат нанят}\},$$

и теперь нам надо вычислить вероятность выполнения строк 5 и 6 процедуры HIRE_ASSISTANT.

Работодатель нанимает кандидата под номером i (строка 5), только если он окажется лучше всех предыдущих (от 1-го до $i-1$-го). Поскольку мы предположили, что кандидаты приходят на собеседование в случайном порядке, это означает, что первые i кандидатов также прибыли в случайном порядке. Любой из первых i кандидатов с равной вероятностью может оказаться лучшим. Поэтому вероятность того, что квалификация претендента с номером i выше квалификации претендентов с номерами от 1 до $i-1$, и что он будет взят на работу, равна $1/i$. Пользуясь леммой 5.1, можно прийти к заключению, что

$$\mathrm{E}[X_i] = 1/i. \tag{5.4}$$

Теперь можно приступить к вычислению величины $\mathrm{E}[X]$:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = && \text{(согласно (5.3))} && (5.5)\\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] = && \text{(согласно линейности математического ожидания)}\\
&= \sum_{i=1}^{n} \frac{1}{i} = && \text{(согласно (5.4))}\\
&= \ln n + O(1) && \text{(согласно (A.7))} && (5.6)
\end{aligned}$$

Даже если проведено интервью с n кандидатами, в среднем будет нанято только $\ln n$ из них. Этот результат подытожен в приведенной ниже лемме.

Лемма 5.2. В случае собеседования с кандидатами в случайном порядке полная стоимость найма при использовании алгоритма HIRE_ASSISTANT равна $O(c_h \ln n)$.

Доказательство. Эта оценка непосредственно следует из определения стоимости найма и уравнения (5.6). ■

Порядок роста представленной стоимости найма значительно ниже порядка роста функции $O(c_h n)$, характеризующей стоимость найма в наихудшем случае.

Упражнения

5.2-1. Чему равна вероятность того, что в процедуре HIRE_ASSISTANT будет нанят ровно один кандидат, если предполагается, что собеседование с кандидатами проводится в случайном порядке?

5.2-2. Чему равна вероятность того, что в процедуре HIRE_ASSISTANT будет нанято ровно два кандидата, если предполагается, что собеседование с кандидатами проводится в случайном порядке? Чему равна вероятность того, что будет нанято ровно n кандидатов?

5.2-3. Вычислите математическое ожидание суммы очков на n игральных костях с помощью индикаторных случайных величин.

5.2-4. Решите с помощью индикаторных случайных величин задачу, известную как ***задача о гардеробщике***. Каждый из n посетителей ресторана сдает свою шляпу в гардероб. Гардеробщик возвращает шляпы случайным образом. Чему равно математическое ожидание количества посетителей, получивших обратно свои собственные шляпы?

5.2-5. Пусть $A[1..n]$ — массив, состоящий из n различных чисел. Если $i < j$ и $A[i] > A[j]$, то пара (i, j) называется ***инверсией*** массива A (более детальную информацию об инверсиях можно найти в задаче 2-4). Предположим, что элементы массива A образуют равномерную случайную перестановку чисел $\langle 1, 2, \ldots, n\rangle$. Воспользуйтесь индикаторными случайными величинами для поиска математического ожидания количества инверсий в массиве.

5.3 Рандомизированные алгоритмы

В предыдущем разделе было показано, как знание информации о распределении входных данных может помочь проанализировать поведение алгоритма в среднем случае. Однако часто встречаются ситуации, когда такими знаниями мы не располагаем, и анализ среднего поведения невозможен. Но, как упоминалось в разделе 5.1, иногда есть возможность использовать рандомизированные алгоритмы.

В задачах наподобие задачи о найме сотрудника, в которой важную роль играет предположение о равновероятности всех перестановок входных данных, вероятностный анализ приводит к разработке рандомизированного алгоритма. Вместо того чтобы предполагать то или иное распределение входных данных, мы попросту навязываем его. В частности, перед запуском алгоритма мы производим случайную перестановку кандидатов, чтобы добиться выполнения условий равновероятности всех перестановок. Такая модификация алгоритма не влияет на величину математического ожидания найма сотрудника, примерно равную $\ln n$. Это означает, что такое поведение алгоритма присуще *любому* множеству входных данных, независимо от его конкретного распределения.

Давайте рассмотрим отличия вероятностного анализа от рандомизированных алгоритмов. В разделе 5.2 мы выяснили, что если кандидаты приходят на собеседование в случайном порядке, то математическое ожидание количества наймов новых менеджеров приблизительно равно $\ln n$. Обратите внимание на то, что представленный алгоритм является детерминированным, т.е. для любых конкретных входных данных количество наймов новых менеджеров всегда будет одним и тем же. Кроме того, эта величина различна для разных входных данных и зависит от распределения рангов кандидатов. Поскольку количество наймов зависит только от рангов кандидатов, каждый отдельно взятый набор входных данных можно представить в виде перечисления рангов кандидатов в порядке нумерации последних, т.е. в виде последовательности $\langle rank(1), rank(2), \ldots, rank(n)\rangle$. В случае списка рангов $A_1 = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle$ новый менеджер будет всегда наниматься 10 раз, поскольку каждый очередной кандидат лучше предыдущего, и строки 5–6 будут выполняться при каждой итерации алгоритма. Если же список

рангов имеет вид $A_2 = \langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1\rangle$, то новый менеджер будет нанят только один раз, во время первой итерации. Список $A_3 = \langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6\rangle$ дает три найма, после интервью с кандидатами, имеющими ранг 5, 8 и 10. Напомним, что стоимость алгоритма зависит от того, сколько раз происходит найм нового сотрудника. Легко убедиться в том, что есть дорогостоящие входные данные (такие как A_1), дешевые входные данные (такие как A_2), и входные данные средней стоимости, такие как A_3.

Рассмотрим теперь рандомизированный алгоритм, в котором сначала выполняется перестановка кандидатов на должность и только после этого определяется, кто из них самый лучший. В этом случае рандомизация представляет собой часть алгоритма, а не входных данных. При этом для отдельно взятого набора входных данных, например, для представленного выше массива A_3, нельзя заранее сказать, сколько наймов будет выполнено, поскольку эта величина изменяется при каждом запуске алгоритма. При первом запуске алгоритма с входными данными A_3 в результате перестановки могут получиться входные данные A_1, и найм произойдет 10 раз, а при втором запуске перестановка может дать входные данные A_2, и найм будет только один. Запустив алгоритм третий раз, можно получить еще какое-нибудь количество наймов. При каждом запуске алгоритма стоимость его работы зависит от случайного выбора и, скорее всего, будет отличаться от стоимости предыдущего запуска. В этом и во многих других рандомизированных алгоритмах *никакие входные данные не могут вызвать наихудшее поведение алгоритма*. Даже ваш злейший враг не сможет подобрать плохой входной массив, поскольку дальнейшая случайная перестановка приводит к тому, что порядок входных элементов становится несущественным. Рандомизированные алгоритмы плохо ведут себя лишь тогда, когда генератор случайных чисел выдаст “неудачную” перестановку.

Единственное изменение, которое нужно внести в алгоритм найма сотрудника для рандомизации, — это добавить код случайной перестановки.

```
RANDOMIZED_HIRE_ASSISTANT(n)
 1  Случайная перестановка списка кандидатов.
 2  Best ← 0   ▷ Кандидат 0 — наименее квалифицированный
                ▷ фиктивный кандидат
 3  for i ← 1 to n
 4      do собеседование с кандидатом i
 5         if кандидат i лучше кандидата Best
 6           then Best ← i
 7                Нанимаем кандидата i
```

С помощью этого простого изменения создается рандомизированный алгоритм, производительность которого такая же, как и в случае предположения о том, что собеседование с кандидатами производится в случайном порядке.

Лемма 5.3. Математическое ожидание стоимости процедуры RANDOMIZED_HIRE_ASSISTANT равно $O(c_h \ln n)$.

Доказательство. После перестановки элементов входного массива возникает ситуация, идентичная рассмотренной при вероятностном анализе алгоритма HIRE_ASSISTANT. ■

При сравнении леммы 5.2 и леммы 5.3 проявляется различие между вероятностным анализом и рандомизированными алгоритмами. В лемме 5.2 делается предположение о виде входных данных. В лемме 5.3 такие предположения отсутствуют, хотя для рандомизации входных величин требуется дополнительное время. В оставшейся части настоящего раздела обсуждаются некоторые вопросы, связанные с входными данными, которые подверглись случайной перестановке.

Массивы, полученные в результате случайной перестановки

Во многих алгоритмах входные данные рандомизируются путем перестановки элементов исходного входного массива (впрочем, существуют и другие способы рандомизации). Мы рассмотрим два метода, позволяющие выполнить эту операцию. Будем считать, что у нас имеется массив A, который содержит элементы от 1 до n. Наша цель — случайным образом переставить элементы массива.

Один из распространенных методов заключается в том, чтобы присвоить каждому элементу $A[i]$ случайный приоритет $P[i]$, а затем отсортировать элементы массива A в соответствии с их приоритетами. Например, если исходный массив имеет вид $A = \langle 1, 2, 3, 4 \rangle$ и выбраны случайные приоритеты $P = \langle 36, 3, 97, 19 \rangle$, то в результате будет получен массив $B = \langle 2, 4, 1, 3 \rangle$, поскольку приоритет второго элемента самый низкий, за ним по приоритету идет четвертый элемент, после него — первый и, наконец, третий. Назовем эту процедуру PERMUTE_BY_SORTING:

```
PERMUTE_BY_SORTING(A)
1  n ← length[A]
2  for i ← 1 to n
3       do P[i] = RANDOM(1, n^3)
4   Отсортировать массив A с ключом сортировки P
5  return A
```

В строке 3 выбирается случайное число от 1 до n^3. Такой интервал выбран для того, чтобы снизить вероятность наличия одинаковых приоритетов в массиве P. (В упражнении 5.3-5 требуется доказать, что вероятность отсутствия в массиве одинаковых приоритетов составляет по меньшей мере $1 - 1/n$, а в упражнении 5.3-6 — реализовать алгоритм, даже если несколько приоритетов могут иметь одинаковые значения.) Будем считать, что одинаковых приоритетов в массиве нет.

В представленном выше псевдокоде наиболее трудоемкая процедура — сортировка в строке 4. Как будет показано в главе 8, если использовать сортировку сравнением, то время ее работы будет выражаться как $\Omega(n \lg n)$. Эта нижняя грань достижима, поскольку мы уже убедились, что сортировка слиянием выполняется в течение времени $\Theta(n \lg n)$. (Во второй части книги мы познакомимся и с другими видами сортировки, время работы которых также имеет вид $\Theta(n \lg n)$.) Если приоритет $P[i]$ является j-м в порядке возрастания, то после сортировки элемент $A[i]$ будет расположен в позиции с номером j. Таким образом мы достигнем требуемой перестановки входных данных. Осталось доказать, что в результате выполнения этой процедуры будет получена ***случайная перестановка с равномерным распределением***, другими словами, что все перестановки чисел от 1 до n генерируются с одинаковой вероятностью.

Лемма 5.4. В предположении отсутствия одинаковых приоритетов в результате выполнения процедуры PERMUTE_BY_SORTING получается случайная перестановка входных значений с равномерным распределением.

Доказательство. Начнем с рассмотрения частной перестановки, в которой каждый элемент $A[i]$ получает i-й приоритет (в порядке возрастания). Покажем, что вероятность такой перестановки равна $1/n!$. Обозначим через X_i ($i = 1, 2, \ldots, n$) событие, состоящее в том, что элемент $A[i]$ получает i-й приоритет. Теперь вычислим вероятность того, что события X_i происходят для всех i. Эта вероятность равна

$$\Pr\{X_1 \cap X_2 \cap X_3 \cap \ldots \cap X_{n-1} \cap X_n\}.$$

Воспользовавшись результатами упражнения B.2-6, можно показать, что эта вероятность равна

$$\Pr\{X_1\} \cdot \Pr\{X_2 \mid X_1\} \cdot \Pr\{X_3 \mid X_2 \cap X_1\} \cdot \Pr\{X_4 \mid X_3 \cap X_2 \cap X_1\} \cdot \ldots \cdot \\ \cdot \Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \ldots \cap X_1\} \cdot \ldots \cdot \Pr\{X_n \mid X_{n-1} \cap \ldots \cap X_1\}.$$

Вероятность события X_1 равна $\Pr\{X_1\} = 1/n$, поскольку это вероятность того, что приоритет выбранного наугад одного из n элементов окажется минимальным. Теперь заметим, что $\Pr\{X_2 \mid X_1\} = 1/(n-1)$, поскольку при условии, что приоритет элемента $A[1]$ минимальный, каждый из оставшихся $n-1$ элементов имеет одинаковые шансы располагать вторым наименьшим приоритетом. В общем случае при $i = 2, 3, \ldots, n$ выполняется соотношение $\Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \ldots \cap \cap X_1\} = 1/(n-i+1)$, поскольку при условии, что приоритеты элементов от $A[1]$ до $A[i-1]$ равны от 1 до $i-1$ (в порядке возрастания), каждый из оставшихся $n-(i-1)$ элементов имеет одинаковые шансы располагать i-м наименьшим приоритетом. Таким образом, справедливо следующее выражение:

$$\Pr\{X_1 \cap X_2 \cap X_3 \cap \ldots \cap X_{n-1} \cap X_n\} = \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right)\ldots\left(\frac{1}{2}\right)\left(\frac{1}{1}\right) = \frac{1}{n!}.$$

Тем самым доказано, что вероятность получения тождественной перестановки равна $1/n!$.

Это доказательство можно обобщить на случай произвольной перестановки приоритетов. Рассмотрим произвольную фиксированную перестановку $\sigma = \langle \sigma(1), \sigma(2), \ldots, \sigma(n) \rangle$ множества $\{1, 2, \ldots, n\}$. Обозначим через r_i ранг приоритета, присвоенного элементу $A[i]$, причем элемент с j-м приоритетом имеет ранг, равный j. Если мы определим X_i как событие, при котором элемент $A[i]$ получает $\sigma(i)$-й приоритет (т.е. $r_i = \sigma(i)$), то можно провести доказательство, аналогичное приведенному выше. Таким образом, при вычислении вероятности получения той или иной конкретной перестановки рассуждения и расчеты будут идентичными изложенным выше. Поэтому вероятность получения такой перестановки также равна $1/n!$. ■

Возможно, некоторые читатели сделают вывод, что для доказательства того, что все случайные перестановки распределены с равной вероятностью, достаточно показать, что для каждого элемента $A[i]$ вероятность оказаться в позиции j равна $1/n$. Выполнив упражнение 5.3-4, можно убедиться, что это условие недостаточное.

Более предпочтительным методом получения случайной перестановки является способ, который заключается в перестановке элементов заданного массива "на месте". С помощью процедуры RANDOMIZE_IN_PLACE эту операцию можно выполнить за время $O(n)$. В ходе i-й итерации элемент $A[i]$ случайным образом выбирается из множества элементов от $A[i]$ до элемента $A[n]$, после чего в последующих итерациях этот элемент больше не изменяется:

```
RANDOMIZE_IN_PLACE(A)
1  n ← length[A]
2  for i ← 1 to n
3       do Обменять A[i] ↔ A[RANDOM(i, n)]
```

Покажем с помощью инварианта цикла, что в результате выполнения процедуры RANDOMIZE_IN_PLACE получаются случайные перестановки с равномерным распределением. Назовем k-перестановкой (размещением) данного n-элементного множества последовательность, состоящую из k элементов, выбранных среди n элементов исходного множества (см. приложение В). Всего имеется $n!/(n-k)!$ возможных k-перестановок.

Лемма 5.5. В результате выполнения процедуры RANDOMIZE_IN_PLACE получаются равномерно распределенные перестановки.

Доказательство. Воспользуемся сформулированным ниже инвариантом цикла.

> Непосредственно перед i-й итерацией цикла **for** из строк 2–3, вероятность того, что в подмассиве $A[1..i-1]$ находится определенная $(i-1)$-перестановка, равна $(n-i+1)!/n!$.

Необходимо показать, что это утверждение истинно перед первой итерацией цикла, что итерации сохраняют истинность инварианта и что оно позволяет показать корректность алгоритма по завершении цикла.

Инициализация. Рассмотрим ситуацию, которая имеет место непосредственно перед первой итерацией цикла ($i = 1$). Согласно формулировке инварианта цикла, вероятность нахождения каждого размещения из 0 элементов в подмассиве $A[1..0]$ равна $(n-i+1)!/n! = n!/n! = 1$. Подмассив $A[1..0]$ — пустой, а 0-размещение по определению не содержит ни одного элемента. Таким образом, подмассив $A[1..0]$ содержит любое 0-размещение с вероятностью 1, и инвариант цикла выполняется перед первой итерацией.

Сохранение. Мы считаем, что перед i-ой итерацией вероятность того, что в подмассиве $A[1..i-1]$ содержится заданное размещение $i-1$ элементов, равна $(n-i+1)!/n!$. Теперь нам нужно показать, что после i-ой итерации каждая из возможных i-перестановок может находиться в подмассиве $A[1..i]$ с вероятностью $(n-i)!/n!$.

Рассмотрим i-ю итерацию. Рассмотрим некоторое конкретное размещение i элементов и обозначим его элементы как $\langle x_1, x_2, \ldots, x_i\rangle$. Это размещение состоит из размещения $i-1$ элемента $\langle x_1, x_2, \ldots, x_{i-1}\rangle$, за которым следует значение x_i, которое помещается в ходе выполнения алгоритма в элемент $A[i]$. Пусть E_1 обозначает событие, при котором в результате первых $i-1$ итераций в подмассиве $A[1..i-1]$ создается определенное размещение $i-1$ элементов $\langle x_1, x_2, \ldots, x_{i-1}\rangle$. Согласно инварианту цикла, $\Pr\{E_1\} = (n-i+1)!/n!$. Пусть теперь E_2 — событие, при котором в ходе i-ой итерации элементу $A[i]$ присваивается значение x_i. Размещение $\langle x_1, x_2, \ldots, x_i\rangle$ формируется в подмассиве $A[1..i]$ только при условии, что происходят оба события — и E_1, и E_2, так что мы должны вычислить значение $\Pr\{E_2 \cap E_1\}$. Воспользовавшись уравнением (B.14), получаем:

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\}.$$

Вероятность $\Pr\{E_2 \mid E_1\}$ равна $1/(n-i+1)$, поскольку в строке 3 алгоритма производится случайный выбор величины x_i из $n-i+1$ элемента подмассива $A[i..n]$. Таким образом, получаем:

$$\begin{aligned}\Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\} \Pr\{E_1\} = \\ &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} = \frac{(n-i)!}{n!}.\end{aligned}$$

Завершение. После завершения алгоритма $i = n+1$, поэтому заданное размещение n элементов находится в подмассиве $A[1..n]$ с вероятностью $(n-n)!/n! = 1/n!$.

Таким образом, в результате выполнения процедуры RANDOMIZE_IN_PLACE получаются равномерно распределенные случайные перестановки. ■

Рандомизированный алгоритм зачастую является самым простым и наиболее эффективным путем решения задачи. Время от времени вы будете встречаться с такими алгоритмами в данной книге.

Упражнения

5.3-1. У профессора возникли возражения против инварианта цикла, использующегося при доказательстве леммы 5.5. Он сомневается, что этот инвариант выполняется перед первой итерацией. Согласно его доводам, пустой подмассив не содержит никаких размещений из 0 элементов, поэтому вероятность того, что в таком подмассиве находится то или иное размещение, должна быть равна 0. Из этих рассуждений следует, что инвариант цикла перед первой итерацией не выполняется. Перепишите процедуру RANDOMIZE_IN_PLACE таким образом, чтобы связанный с нею инвариант цикла перед первой итерацией применялся к непустому подмассиву, и соответствующим образом модифицируйте доказательство леммы 5.5.

5.3-2. Профессор решил разработать алгоритм, в результате выполнения которого получались бы все случайные перестановки, кроме тождественной. Он предложил такую процедуру:

```
PERMUTE_WITHOUT_IDENTITY(A)
1  n ← length[A]
2  for i ← 1 to n − 1
3        do Обменять A[i] ↔ A[RANDOM(i + 1, n)]
```

Добьется ли профессор поставленной цели с помощью этого кода?

5.3-3. Предположим, что вместо того, чтобы менять местами элемент $A[i]$ со случайно выбранным элементом из подмассива $A[i..n]$, мы меняем его местами с любым случайно выбранным элементом массива A:

```
PERMUTE_WITH_ALL(A)
1  n ← length[A]
2  for i ← 1 to n
3        do Обменять A[i] ↔ A[RANDOM(1, n)]
```

Получится ли в результате выполнения этого кода равномерная случайная перестановка? Обоснуйте ваш ответ.

5.3-4. Профессор предложил для генерирования случайных перестановок с однородным распределением такую процедуру:

```
PERMUTE_BY_CYCLIC(A)
1  n ← length[A]
2  offset ← RANDOM(1, n)
3  for i ← 1 to n
4       do dest ← i + offset
5          if dest > n
6             then dest ← dest − n
7          B[dest] ← A[i]
8  return B
```

Покажите, что элемент $A[i]$ оказывается в определенной позиции массива B с вероятностью $1/n$. Затем покажите, что алгоритм профессора ошибочен в том смысле, что полученные в результате его выполнения перестановки не будут распределены равномерно.

⋆ 5.3-5. Докажите, что в результате выполнения процедуры PERMUTE_BY_SORTING вероятность отсутствия одинаковых элементов в массиве P не меньше величины $1 - 1/n$.

5.3-6. Объясните, как следует реализовать алгоритм PERMUTE_BY_SORTING в случае, когда два или более приоритета идентичны. Другими словами, алгоритм должен выдавать случайные равномерно распределенные перестановки даже в случае, когда два или большее количество приоритетов имеют одинаковые значения.

⋆ 5.4 Вероятностный анализ и дальнейшее применение индикаторных случайных величин

В этом разделе повышенной сложности на четырех примерах иллюстрируется дальнейшее применение вероятностного анализа. В первом примере вычисляется вероятность того, что двое из k человек родились в один и тот же день года. Во втором примере рассматривается процесс случайного наполнения урн шарами, в третьем — исследуется событие, при котором в процессе бросания монеты несколько раз подряд выпадает орел. В последнем примере анализируется разновидность задачи о найме сотрудника, в которой решение о найме принимается без проведения собеседования со всеми кандидатами.

5.4.1 Парадокс дней рождения

Первым будет рассмотрен ***парадокс дней рождения***. Сколько людей нужно собрать в одной комнате, чтобы вероятность совпадения даты рождения у двух из них достигла 50%? Полученное в результате решения этой задачи число на удивление мало. Парадокс в том, что это число намного меньше, чем количество дней в году.

Чтобы решить задачу, присвоим всем, кто находится в комнате, номера от 1 до k, где k — количество людей в комнате. Наличие високосных годов проигнорируем и предположим, что в каждом году $n = 365$ дней. Пусть b_i — дата, на которую приходится день рождения i-й персоны ($i = 1, 2, \ldots, k$, $1 \leqslant b_i \leqslant n$). Предположим также, что дни рождения равномерно распределены по всему году, так что $\Pr\{b_i = r\} = 1/n$ для всех $i = 1, 2, \ldots, k$ и $r = 1, 2, \ldots, n$.

Вероятность того, что даты рождения двух человек i и j совпадают, зависит от того, является ли случайный выбор этих дат независимым. В дальнейшем предполагается, что дни рождения независимы, поэтому вероятность того, что i-й и j-й посетители комнаты родились в день r, можно вычислить следующим образом:

$$\Pr\{b_i = r \text{ и } b_j = r\} = \Pr\{b_i = r\} \Pr\{b_j = r\} = 1/n^2.$$

Таким образом, вероятность того, что эти люди родились в один и тот же день, равна

$$\Pr\{b_i = b_j\} = \sum_{r=1}^{n} \Pr\{b_i = r \text{ и } b_j = r\} = \sum_{r=1}^{n} \frac{1}{n^2} = \frac{1}{n}. \tag{5.7}$$

Интуитивно легче понять такое утверждение: если день рождения человека b_i зафиксирован, то вероятность того, что день рождения человека b_j выпадет на эту же дату, равна $1/n$. Таким образом, вероятность того, что даты рождения двух человек i и j совпадают, равна вероятности того, что день рождения одного из них выпадет на заданную дату. Однако обратите внимание, что это совпадение основано на предположении о независимости дней рождения.

Теперь можно проанализировать вероятность того, что хотя бы двое из k людей родились в один и тот же день, рассматривая дополняющее событие. Вероятность совпадения хотя бы двух дней рождения равна 1, уменьшенной на величину вероятности того, что все дни рождения различаются. Событие, при котором все дни рождения различаются, можно представить так:

$$B_k = \bigcap_{i=1}^{n} A_i,$$

где A_i — событие, состоящее в том, что день рождения i-го человека отличается от дня рождения j-го человека для всех $i < j$. Поскольку мы можем записать

$B_k = A_k \cap B_{k-1}$, из уравнения (B.16) мы получим следующее рекуррентное соотношение:

$$\Pr\{B_k\} = \Pr\{B_{k-1}\} \Pr\{A_k \mid B_{k-1}\}, \tag{5.8}$$

начальное условие которого — $\Pr\{B_1\} = \Pr\{A_1\} = 1$. Другими словами, вероятность того, что дни рождения $b_1, b_2, \ldots, b_k$ различны, равна произведению вероятности того, что различны дни рождения $b_1, b_2, \ldots, b_{k-1}$, на вероятность того, что $b_k \neq b_i$ для $i = 1, 2, \ldots, k-1$ при условии, что $b_1, b_2, \ldots, b_{k-1}$ различны.

Если $b_1, b_2, \ldots, b_{k-1}$ различны, то условная вероятность того, что $b_k \neq b_i$ для $i = 1, 2, \ldots, k-1$, равна $\Pr\{A_k | B_{k-1}\} = (n-k+1)/n$, поскольку из n дней незанятыми остались только $n-(k-1)$. Итеративно применяя рекуррентное соотношение (5.8), получим:

$$\begin{aligned}
\Pr\{B_k\} &= \Pr\{B_{k-1}\} \Pr\{A_k \mid B_{k-1}\} = \\
&= \Pr\{B_{k-2}\} \Pr\{A_{k-1} \mid B_{k-2}\} \Pr\{A_k \mid B_{k-1}\} = \\
&\;\;\vdots \\
&= \Pr\{B_1\} \Pr\{A_2 \mid B_1\} \Pr\{A_3 \mid B_2\} \ldots \Pr\{A_k \mid B_{k-1}\} = \\
&= 1 \cdot \left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right) \ldots \left(\frac{n-k+1}{n}\right) = \\
&= 1 \cdot \left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right) \ldots \left(1-\frac{k-1}{n}\right).
\end{aligned}$$

Неравенство (3.11), $1 + x \leqslant e^x$, дает нам следующее соотношение:

$$\Pr\{B_k\} \leqslant e^{-1/n} e^{-2/n} \ldots e^{-(k-1)/n} = e^{-\sum_{i=1}^{k-1} i/n} = e^{-k(k-1)/2n} \leqslant 1/2,$$

которое справедливо при $-k(k-1)/2n \leqslant \ln(1/2)$. Итак, вероятность того, что все k дней рождения различаются, не меньше величины $1/2$ при условии, что $k(k-1) \geqslant 2n \ln 2$. Решая квадратное уравнение относительно k, получим, что это условие выполняется при $k \geqslant \left(1 + \sqrt{1 + 8n \ln 2}\right)/2$, что при $n = 365$ дает условие $k \geqslant 23$. Таким образом, если в комнате не менее 23 человек, вероятность того, что хотя бы двое из них родились в один и тот же день, не меньше $1/2$. А, например, на Марсе год длится 669 дней, поэтому для того, чтобы добиться того же эффекта исходя из продолжительности марсианского года, понадобилось бы собрать не менее 31 марсианина.

Анализ с помощью индикаторных случайных величин

Индикаторные случайные величины дают возможность проведения простого (хотя и приближенного) анализа парадокса дней рождения. Определим для каждой

пары (i, j) находящихся в комнате k людей индикаторную случайную величину X_{ij} $(1 \leqslant i < j \leqslant k)$ следующим образом:

$$X_{ij} = \mathrm{I}\{\text{Дни рождения } i \text{ и } j \text{ совпадают}\} = \\ = \begin{cases} 1 & \text{если дни рождения } i \text{ и } j \text{ совпадают,} \\ 0 & \text{в противном случае.} \end{cases}$$

Согласно уравнению (5.7) вероятность того, что дни рождения двух людей совпадают, равна $1/n$, поэтому, в соответствии с леммой 5.1, получаем:

$$\mathrm{E}[X_{ij}] = \Pr\{\text{Дни рождения } i \text{ и } j \text{ совпадают}\} = 1/n.$$

Полагая X случайной величиной, которая представляет собой количество пар людей, дни рождения которых совпадают, получим:

$$X = \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}.$$

Применяя к обеим частям этого равенства операцию вычисления математического ожидания и воспользовавшись свойством ее линейности, получаем:

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}\right] = \sum_{i=1}^{k} \sum_{j=i+1}^{k} \mathrm{E}[X_{ij}] = \binom{k}{2} \cdot \frac{1}{n} = \frac{k(k-1)}{2n}.$$

Поэтому если $k(k-1) \geqslant 2n$, математическое ожидание количества пар людей, родившихся в один и тот же день, не меньше 1. Таким образом, если в комнате как минимум $\sqrt{2n} + 1$ людей, то можно ожидать, что хотя бы у двух из них дни рождения совпадают. При $n = 365$ и $k = 28$ математическое ожидание количества пар, родившихся в один и тот же день, равно $(27 \cdot 28)/(2 \cdot 365) \approx 1.0356$. Таким образом, если в комнате находится 28 человек, то следует ожидать, что хотя бы у двух из них день рождения совпадет. На Марсе, где год длится 669 дней, для того, чтобы добиться того же эффекта, понадобилось бы собрать не менее 38 марсиан.

В первом анализе, в котором использовались только вероятности, определялось, сколько людей нужно собрать, чтобы вероятность существования пары с совпадающими днями рождения превысила $1/2$. В ходе второго анализа, в котором использовались индикаторные случайные величины, определялось количество людей, при котором математическое ожидание числа пар с одним днем рождения на двоих равно 1. Несмотря на то, что в этих двух ситуациях точное количество людей различается, в асимптотическом пределе оно одинаково: $\Theta(\sqrt{n})$.

5.4.2 Шары и урны

Рассмотрим процесс случайного наполнения b урн одинаковыми шарами, пронумерованными натуральными числами от 1 до b. Шары опускаются в урны независимо друг от друга, и вероятность того, что шар окажется в некоторой из урн, одинакова для всех урн и равна $1/b$. Таким образом, процесс заполнения урн шарами представляет собой последовательность испытаний по схеме Бернулли (см. приложение В) с вероятностью успеха $1/b$, где успех состоит в том, что шар попадает в заданную урну. В частности, эта модель может оказаться полезной при анализе хеширования (см. главу 11).

Сколько шаров окажется в заданной урне? Количество шаров, попавших в определенную урну, подчиняется биномиальному распределению $b(kn, 1/b)$. Если всего в урны было опущено n шаров, то, согласно уравнению (B.36), математическое ожидание количества шаров в урне равно n/b.

Сколько в среднем требуется шаров для того, чтобы в данной урне оказался один шар? Количество шаров, которое требуется для того, чтобы в данной урне оказался шар, подчиняется геометрическому распределению с вероятностью $1/b$ и, согласно уравнению (B.31), математическое ожидание количества шаров, которое следует опустить в урны, равно $1/(1/b) = b$.

Сколько шаров нужно опустить в урны для того, чтобы в каждой из них оказалось хотя бы по одному шару? Назовем "попаданием" опускание шара в пустую урну. Нам надо определить математическое ожидание количества опусканий n, необходимых для b попаданий.

С помощью попаданий, n опусканий можно разбить на этапы. Этап под номером i длится от $i-1$-го попадания до i-го попадания. Первый этап состоит из одного (первого) опускания, поскольку если все урны пусты, то мы с необходимостью получим попадание. На i-м этапе имеется $i-1$ корзина с шарами и $b-i+1$ пустых корзин. Таким образом, при каждом опускании шара на i-м этапе вероятность попадания равна $(b-i+1)/b$.

Пусть n_i — количество опусканий на i-м этапе. Тогда количество шаров, необходимых для попадания в b урн, равно $n = \sum_{i=1}^{b} n_i$. Все случайные значения n_i подчиняются геометрическому распределению с вероятностью успеха $(b-i+1)/b$, и, в соответствии с уравнением (B.31),

$$\mathrm{E}[n_i] = \frac{b}{b-i+1}.$$

Пользуясь линейностью математического ожидания, получаем:

$$\mathrm{E}[n] = \mathrm{E}\left[\sum_{i=1}^{b} n_i\right] = \sum_{i=1}^{b} \mathrm{E}[n_i] = \sum_{i=1}^{b} \frac{b}{b-i+1} = b\sum_{i=1}^{b} \frac{1}{i} = b(\ln b + O(1)).$$

Последняя строка этой цепочки следует из границы (A.7) конечного гармонического ряда. Таким образом, прежде чем в каждой урне окажется хотя бы по одному шару, понадобится приблизительно $b \ln b$ шаров.

5.4.3 Последовательности выпадения орлов

Предположим, что монета подбрасывается n раз. Какое количество последовательных выпадений орла можно ожидать? Как покажет последующий анализ, эта величина ведет себя как $\Theta(\lg n)$.

Докажем сначала, что математическое ожидание длины наибольшей последовательности орлов представляет собой $O(\lg n)$. Вероятность того, что при очередном подбрасывании выпадет орел, равна $1/2$. Пусть A_{ik} — событие, когда последовательность выпадений орлов длиной не менее k начинается с i-го подбрасывания, или, более строго, A_{ik} — событие, когда при k последовательных подбрасываниях монеты $i, i+1, \ldots, i+k-1$ ($1 \leqslant k \leqslant n$ и $1 \leqslant i \leqslant n-k+1$) будут выпадать одни орлы. Поскольку подбрасывания монеты осуществляются независимо, для каждого данного события A_{ik} вероятность того, что во всех k подбрасываниях выпадут одни орлы, определяется следующим образом:

$$\Pr\{A_{ik}\} = 1/2^k. \tag{5.9}$$

Для $k = 2\lceil \lg n \rceil$ можно записать

$$\Pr\left\{A_{i,2\lceil \lg n \rceil}\right\} = 1/2^{2\lceil \lg n \rceil} \leqslant 1/2^{2 \lg n} = 1/n^2,$$

так что вероятность того, что последовательность повторных выпадений орлов длиной не менее $2\lceil \lg n \rceil$ начинается с i-го подбрасывания, довольно невелика. Имеется не более $n - 2\lceil \lg n \rceil + 1$ подбрасываний, с которых может начаться указанная последовательность орлов. Таким образом, вероятность того, что последовательность повторных выпадений орлов длиной не менее $2\lceil \lg n \rceil$ начинается при любом подбрасывании, равна

$$\Pr\left\{\bigcup_{i=1}^{n-2\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil}\right\} \leqslant \sum_{i=1}^{n-2\lceil \lg n \rceil+1} \frac{1}{n^2} < \sum_{i=1}^{n} \frac{1}{n^2} = \frac{1}{n}. \tag{5.10}$$

Справедливость этого соотношения следует из неравенства Буля (B.18), согласно которому вероятность объединения событий не превышает сумму вероятностей отдельных событий. (Заметим, что неравенство Буля выполняется даже для тех событий, которые не являются независимыми.)

Теперь воспользуемся неравенством (5.10) для ограничения длины самой длинной последовательности выпадения орлов. Пусть L_j ($j = 0, 1, 2, \ldots, n$) —событие,

когда длина самой длинной последовательности выпадения орлов равна j. В соответствии с определением математического ожидания

$$\mathrm{E}[L] = \sum_{j=0}^{n} j \Pr\{L_j\}. \tag{5.11}$$

Можно попытаться оценить эту сумму с помощью верхних границ каждой из величин $\Pr\{L_j\}$ аналогично тому, как это было сделано в неравенстве (5.10). К сожалению, этот метод не может обеспечить хороших оценок. Однако достаточно точную оценку можно получить с помощью некоторых интуитивных рассуждений, которые вытекают из проведенного выше анализа. Присмотревшись внимательнее, можно заметить, что в сумме (5.11) нет ни одного слагаемого, в котором оба множителя — j и $\Pr\{L_j\}$ — были бы большими. Почему? При $j \geqslant 2\lceil \lg n \rceil$ величина $\Pr\{L_j\}$ очень мала, а при $j < 2\lceil \lg n \rceil$ оказывается невелико само значение j. Выражаясь более формально, можно заметить, что события $L_j, j = 0, 1, \ldots, n$ несовместимые, поэтому вероятность того, что непрерывная последовательность выпадения орлов длиной не менее $2\lceil \lg n \rceil$ начинается с любого подбрасывания монеты, равна $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\}$. Согласно неравенству (5.10), мы имеем $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\} < 1/n$. Кроме того, из $\sum_{j=0}^{n} \Pr\{L_j\} = 1$ вытекает $\sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\{L_j\} \leqslant 1$. Таким образом, мы получаем:

$$\begin{aligned}
\mathrm{E}[L] = \sum_{j=0}^{n} j \Pr\{L_j\} &= \sum_{j=0}^{2\lceil \lg n \rceil - 1} j \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} j \Pr\{L_j\} < \\
&< \sum_{j=0}^{2\lceil \lg n \rceil - 1} (2\lceil \lg n \rceil) \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} n \Pr\{L_j\} = \\
&= 2\lceil \lg n \rceil \sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\{L_j\} + n \sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\} < \\
&< 2\lceil \lg n \rceil \cdot 1 + n \cdot (1/n) = O(\lg n).
\end{aligned}$$

Шансы на то, что длина последовательности непрерывных выпадений орла превысит величину $r\lceil \lg n \rceil$, быстро убывает с ростом r. Для $r \geqslant 1$ вероятность того, что последовательность $r\lceil \lg n \rceil$ начнется с i-го подбрасывания, равна

$$\Pr\left\{A_{i, r\lceil \lg n \rceil}\right\} = 1/2^{r\lceil \lg n \rceil} \leqslant 1/n^r.$$

Таким образом, вероятность образования непрерывной цепочки из последовательных выпадений орла, имеющей длину не менее $r\lceil \lg n \rceil$, не превышает $n/n^r = 1/n^{r-1}$. Это утверждение эквивалентно утверждению, что длина такой цепочки меньше величины $r\lceil \lg n \rceil$ с вероятностью не менее чем $1 - 1/n^{r-1}$.

В качестве примера рассмотрим серию из $n = 1000$ подбрасываний монеты. Вероятность того, что в этой серии орел последовательно выпадет не менее $2\lceil \lg n \rceil = 20$ раз, не превышает $1/n = 1/1000$. Вероятность непрерывного выпадения орла более $3\lceil \lg n \rceil = 30$ раз не превышает $1/n^2 = 1/1\,000\,000$.

Теперь давайте рассмотрим дополняющую нижнюю границу и докажем, что математическое ожидание длины самой длинной непрерывной последовательности выпадений орлов в серии из n подбрасываний равно $\Omega(\lg n)$. Чтобы доказать справедливость этого утверждения, разобьем серию из n подбрасываний приблизительно на n/s групп по s подбрасываний в каждой. Если выбрать $s = \lfloor (\lg n)/2 \rfloor$, то можно показать, что с большой вероятностью по крайней мере в одной из этих групп окажутся все орлы, т.е. самая длинная последовательность выпадения орлов имеет длину как минимум $s = \Omega(\lg n)$. Затем мы покажем, что математическое ожидание длины такой последовательности равно $\Omega(\lg n)$.

Итак, разобьем серию из n испытаний на несколько групп. Количество групп должно быть не менее $\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor$; их длина не превышает $\lfloor (\lg n)/2 \rfloor$ подбрасываний. Оценим вероятность того, что в каждой из групп выпадет хотя бы по одной решке. Согласно уравнению (5.9), вероятность того, что в группе, которая начинается с i-го подбрасывания, выпадут все орлы, определяется как

$$\Pr\left\{A_{i,\lfloor (\lg n)/2 \rfloor}\right\} = \frac{1}{2^{\lfloor (\lg n)/2 \rfloor}} \geqslant \frac{1}{\sqrt{n}}.$$

Таким образом, вероятность того, что последовательность непрерывного выпадения орлов длиной не менее $\lfloor (\lg n)/2 \rfloor$ не начинается с i-го подбрасывания, не превышает величину $1 - 1/\sqrt{n}$. Поскольку все $\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor$ групп образуются из взаимно исключающих независимых подбрасываний, вероятность того, что каждая такая группа *не* будет последовательностью выпадений орлов длиной $\lfloor (\lg n)/2 \rfloor$, не превышает величину

$$\begin{aligned}\left(1 - 1/\sqrt{n}\right)^{\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor} &\leqslant \left(1 - \sqrt{n}\right)^{n/\lfloor (\lg n)/2 \rfloor - 1} \leqslant \left(1 - \sqrt{n}\right)^{2n/\lg n - 1} \leqslant \\ &\leqslant e^{-(2n/\lg n - 1)/\sqrt{n}} = O\left(e^{-\lg n}\right) = O(1/n).\end{aligned}$$

В приведенной выше цепочке соотношений было использовано неравенство (3.11), $1 + x \leqslant e^x$, и тот факт, что при достаточно больших n справедливо соотношение $(2n/\lg n - 1)/\sqrt{n} \geqslant \lg n$ (при желании вы можете в этом убедиться самостоятельно).

Таким образом, вероятность того, что длина самой большой последовательности выпадений орлов превосходит величину $\lfloor (\lg n)/2 \rfloor$, равна

$$\sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^{n} \Pr\{L_j\} \geqslant 1 - O(1/n). \tag{5.12}$$

Теперь можно вычислить нижнюю границу математического ожидания длины самой длинной последовательности орлов. Воспользовавшись в качестве отправной точки уравнением (5.11) и выполнив преобразования, аналогичные проведенным при анализе верхней границы, с учетом неравенства (5.12), получим:

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} = \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor} j \Pr\{L_j\} + \sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^{n} j \Pr\{L_j\} \geqslant \\
&\geqslant \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor} 0 \cdot \Pr\{L_j\} + \sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^{n} \lfloor (\lg n)/2 \rfloor \Pr\{L_j\} = \\
&= 0 \cdot \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor} \Pr\{L_j\} + \lfloor (\lg n)/2 \rfloor \sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^{n} \Pr\{L_j\} \geqslant \\
&\geqslant 0 + \lfloor (\lg n)/2 \rfloor (1 - O(1/n)) = \Omega(\lg n).
\end{aligned}$$

Как в предыдущем примере, более простой, но менее точный анализ можно провести с помощью индикаторных случайных величин. Пусть $X_{ik} = \mathrm{I}\{A_{ik}\}$ — индикаторная случайная величина, связанная с последовательным выпадением не менее k орлов, начиная с i-го подбрасывания монеты. Чтобы подсчитать количество таких последовательностей, определим:

$$X = \sum_{i=1}^{n-k+1} X_{ik}.$$

Вычисляя от обеих частей этого равенства математическое ожидание, получим:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-k+1} X_{ik}\right] = \sum_{i=1}^{n-k+1} \mathrm{E}[X_{ik}] = \\
&= \sum_{i=1}^{n-k+1} \Pr\{A_{ik}\} = \sum_{i=1}^{n-k+1} \frac{1}{2^k} = \frac{n-k+1}{2^k}.
\end{aligned}$$

Подставляя в полученное соотношение различные значения k, можно определить математическое ожидание количества последовательностей длины k. Если это число окажется большим (намного превышающим единицу), то последовательности выпадения орлов, имеющие длину k, встречаются с большой вероятностью. Если же это число намного меньше единицы, то встретить такую последовательность в серии испытаний маловероятно. Если $k = c \lg n$ для некоторой положительной константы c, то мы получим:

$$\mathrm{E}[X] = \frac{n - c\lg n + 1}{2^{c \lg n}} = \frac{n - c\lg n + 1}{n^c} = \frac{1}{n^{c-1}} - \frac{(c\lg n - 1)/n}{n^{c-1}} = \Theta\left(\frac{1}{n^{c-1}}\right).$$

Если число c достаточно большое, математическое ожидание количества непрерывных выпадений орла длиной $c \lg n$ очень мало, из чего можно заключить, что это событие маловероятно. С другой стороны, если $c < 1/2$, то мы получаем, что $\mathrm{E}[X] = \Theta\left(1/n^{1/2-1}\right) = \Theta\left(n^{1/2}\right)$, и можно ожидать, что будет немало последовательностей орлов длины $(1/2)\lg n$. Поэтому вероятность того, что встретится хотя бы одна такая последовательность, достаточно велика. Исходя лишь из этих грубых оценок, можно заключить, что ожидаемая длина самой длинной последовательности орлов равна $\Theta(\lg n)$.

5.4.4 Задача о найме сотрудника в оперативном режиме

В качестве последнего примера рассмотрим одну из разновидностей задачи о найме сотрудника. Предположим, что в целях выбора наиболее подходящего кандидата мы не будем проводить собеседование со всеми претендентами. Мы не хотим повторять процедуру оформления на работу нового сотрудника и увольнения старого в поисках наиболее подходящей кандидатуры. Вместо этого мы попытаемся подыскать такого кандидата, который максимально приблизится к наивысшей степени соответствия должности. При этом необходимо соблюдать одно условие: после каждого интервью претенденту нужно либо сразу предложить должность, либо сообщить ему об отказе. Как достичь компромисса между количеством проведенных интервью и квалификацией взятого на работу кандидата?

Смоделируем эту задачу таким образом. После встречи с очередным кандидатом каждому из них можно дать оценку. Обозначим оценку i-го кандидата как $score(i)$ и предположим, что всем претендентам выставлены разные оценки. После встречи с j кандидатами будет известно, какой из этих j претендентов на должность получил максимальную оценку, однако остается неизвестным, найдется ли среди оставшихся $n-j$ кандидатов человек с более высокой квалификацией. Будем придерживаться следующей стратегии: выберем положительное целое число $k < n$, проведем интервью с k претендентами, отказав им в должности, а затем возьмем на работу первого из последующих $n-k$ претендентов, оценка которого будет превышать оценки всех предыдущих кандидатов. Если же самый квалифицированный специалист окажется среди первых k претендентов, то придется взять на работу n-го кандидата. Формальная реализация этой схемы представлена в приведенной ниже процедуре ON_LINE_MAXIMUM(k, n), которая возвращает номер нанимаемого кандидата:

```
ON_LINE_MAXIMUM(k, n)
1  bestscore ← −∞
2  for i ← 1 to k
3      do if score(i) > bestscore
4           then bestscore ← score(i)
5  for i ← k + 1 to n
```

```
6       do if score(i) > bestscore
7             then return i
8   return n
```

Определим для каждого положительного k вероятность того, что будет нанят наиболее квалифицированный претендент. Затем выберем наилучшее из всех значений k и реализуем описанную стратегию с этим значением. Пока что полагаем k фиксированным. Обозначим как $M(j) = \max_{1 \leqslant i \leqslant j} \{score(i)\}$ наивысшую оценку кандидатов с номерами от 1 до j. Пусть S — событие, определяемое как выбор самого квалифицированного кандидата. а S_i — событие, при котором самым квалифицированным нанятым на работу кандидатом оказался i-й. Поскольку все события S_i являются взаимоисключающими, выполняется соотношение $\Pr\{S\} = \sum_{i=1}^{n} \Pr\{S_i\}$. Поскольку, согласно нашей стратеги, ни один из первых k претендентов на работу не принимается, $\Pr\{S_i\} = 0$ для $i = 1, 2, \ldots, k$. Таким образом, получаем соотношение:

$$\Pr\{S\} = \sum_{i=k+1}^{n} \Pr\{S_i\}. \tag{5.13}$$

Теперь вычислим величину $\Pr\{S_i\}$. Чтобы был принят на работу i-й кандидат, необходимо выполнение двух условий. Во-первых, на i-й позиции должен оказаться самый квалифицированный кандидат (обозначим это событие как B_i), а во-вторых, в ходе выполнения алгоритма не должен быть выбран ни один из претендентов, пребывающий на позициях с $k+1$-й по $i-1$-ю. Второе событие произойдет только тогда, когда при всех j, таких что $k+1 \leqslant j \leqslant i-1$, в строке 6 будет выполняться условие $score(j) < bestscore$. (Поскольку оценки не повторяются, возможность равенства $score(j) = bestscore$ можно проигнорировать.) Другими словами, все оценки от $score(k+1)$ до $score(i-1)$ должны быть меньше $M(k)$; если же одна из них окажется больше $M(k)$, то будет возвращен индекс первой из оценок, превышающей все предыдущие. Обозначим через O_i событие, что не взят на работу ни один из претендентов, проходивших собеседование под номерами от $k+1$ до $i-1$. К счастью, события B_i и O_i независимы. Событие O_i зависит только от порядка нумерации кандидатов, которые находятся на позициях от 1 до $i-1$, а событие B_i зависит только от того, превышает ли оценка кандидата i оценки всех прочих кандидатов. Порядок оценок в позициях от 1 до $i-1$ не влияет на то, превышает ли оценка i-го претендента все предыдущие оценки, а квалификация i-го кандидата не влияет на расположение кандидатов с порядковыми номерами от 1 до $i-1$. Применив уравнение (B.15), получим:

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\} \Pr\{O_i\}.$$

Ясно, что вероятность $\Pr\{B_i\}$ равна $1/n$, поскольку каждый из n кандидатов может получить наивысшую оценку с равной вероятностью. Чтобы произошло

событие O_i, наиболее квалифицированный кандидат среди претендентов с номерами от 1 до $i-1$ должен находиться на одной из первых k позиций. Он с равной вероятностью может оказаться на любой из этих $i-1$ позиций, поэтому $\Pr\{O_i\} = k/(i-1)$ и, соответственно, $\Pr\{S_i\} = k/(n(i-1))$. Воспользовавшись уравнением (5.13), получим:

$$\Pr\{S\} = \sum_{i=k+1}^{n} \Pr\{S_i\} = \sum_{i=k+1}^{n} \frac{k}{n(i-1)} = \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i-1} = \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i}.$$

Ограничим приведенную выше сумму сверху и снизу, заменив суммирование интегрированием. Согласно неравенствам (A.12), получаем:

$$\int_{k}^{n} \frac{1}{x} dx \leqslant \sum_{i=k}^{n-1} \frac{1}{i} \leqslant \int_{k-1}^{n-1} \frac{1}{x} dx.$$

Вычисление этих определенных интегралов дает нам величины верхней и нижней границ:

$$\frac{k}{n}(\ln n - \ln k) \leqslant \Pr\{S\} \leqslant \frac{k}{n}(\ln(n-1) - \ln(k-1)),$$

что приводит к достаточно точной оценке величины $\Pr\{S\}$. Поскольку нам нужно максимально повысить вероятность успешного исхода, постараемся выбрать значение k, при котором нижняя граница $\Pr\{S\}$ имеет максимум. (Выбор нижней границы продиктован еще и тем, что найти ее максимум легче, чем максимум верхней границы.) Дифференцируя $(n/k)(\ln n - \ln k)$ по k, получаем:

$$\frac{1}{n}(\ln n - \ln k - 1).$$

Приравнивая производную к нулю, найдем, что нижняя граница интересующей нас вероятности достигает максимального значения, когда $\ln k = \ln n - 1$ или, что то же самое, когда $k = n/e$. Таким образом, реализовав описанную выше стратегию при $k = n/e$, мы наймем самого достойного кандидата с вероятностью, не меньшей $1/e$.

Упражнения

5.4-1. Сколько человек должно собраться в комнате, чтобы вероятность того, что день рождения у кого-нибудь из них совпадет с вашим, была не меньшей $1/2$? Сколько человек необходимо, чтобы вероятность того, что хотя бы двое из них родились 20 февраля, превысила величину $1/2$?

5.4-2. Предположим, что b урн наполняются шарами. Каждое опускание шара происходит независимо от других, и шар с равной вероятностью может оказаться в любой урне. Чему равно математическое ожидание количества шаров, которое необходимо опустить для того, чтобы хотя бы в одной урне оказалось два шара?

⋆ 5.4-3. При анализе парадокса дней рождения было принято предположение о взаимной независимости всех дней рождения. Является ли это предположение существенным, или достаточно попарной независимости? Обоснуйте ваш ответ.

⋆ 5.4-4. Сколько человек нужно пригласить на вечеринку, чтобы вероятность того, что *трое* из них родились в один и тот же день, достигла заметной величины?

⋆ 5.4-5. Какова вероятность того, что строка длиной k, составленная из символов n-элементного множества, является размещением k элементов этого множества? Как этот вопрос связан с парадоксом дней рождения?

⋆ 5.4-6. Предположим, что n шаров распределяется по n урнам. Каждый шар опускается независимо от других и с равной вероятностью может оказаться в любой из урн. Чему равно математическое ожидание количества пустых урн? Чему равно математическое ожидание количества урн с одним шаром?

⋆ 5.4-7. Уточните нижнюю оценку длины последовательности выпадений орлов. Для этого покажите, что при n подбрасываниях симметричной монеты вероятность того, что такая последовательность будет не длиннее $\lg n - 2 \lg \lg n$, меньше $1/n$.

Задачи

5-1. **Вероятностный подсчет**

С помощью b-битового счетчика можно вести подсчет до $2^b - 1$ элементов. Предложенный Моррисом (R. Morris) ***вероятностный подсчет*** (probabilistic counting) позволяет проводить нумерацию намного большего количества элементов ценой потери точности.

Пусть значение переменной-счетчика $i = 0, 1, \ldots, 2^b - 1$ означает номер элемента n_i возрастающей последовательности неотрицательных чисел. Предположим, что начальное значение счетчика равно нулю, т.е. $n_0 = 0$. Операция INCREMENT увеличивает значение счетчика i случайным образом. Если $i = 2^b - 1$, то в результате этого действия выдается сообщение о переполнении. В противном случае значение счетчика с вероятностью

$1/(n_{i+1} - n_i)$ возрастает на единицу и остается неизменным с вероятностью $1 - 1/(n_{i+1} - n_i)$.

Если для всех $i \geqslant 0$ выбрать $n_i = i$, то мы получим обычный счетчик. Более интересная ситуация возникает, если выбрать, скажем, $n_i = 2^i$ или $n_i = F_i$ (i-е число Фибоначчи; см. раздел 3.2).

В задаче предполагается, что число n_{2^b-1} достаточно большое, чтобы вероятностью переполнения можно было пренебречь.

а) Покажите, что математическое ожидание значение счетчика после применения к нему n операций INCREMENT равно n.

б) Дисперсионный анализ значения счетчика зависит от выбора последовательности n_i. Рассмотрим простой случай, когда $n_i = 100i$ для всех $i \geqslant 0$. Оцените дисперсию значения счетчика после выполнения операции INCREMENT n раз.

5-2. **Поиск в неотсортированном массиве**

В этой задаче исследуются три алгоритма поиска значения x в неотсортированном n-элементном массиве A.

Рассмотрим следующую рандомизированную стратегию. Выбираем элемент массива A с произвольным индексом i и проверяем справедливость равенства $A[i] = x$. Если оно выполняется, то алгоритм завершается. В противном случае продолжаем поиск, случайным образом выбирая новые элементы массива A. Перебор индексов продолжается до тех пор, пока не будет найден такой индекс j, что $A[j] = x$, или пока не будут проверены все элементы массива. Заметим, что выбор каждый раз производится среди всех индексов массива, поэтому круг поиска не сужается и один и тот же элемент может проверяться неоднократно.

а) Напишите псевдокод процедуры RANDOM_SEARCH, реализующей описанную стратегию. Позаботьтесь, чтобы алгоритм прекращал работу после того, как будут проверены все индексы массива.

б) Предположим, что имеется всего один индекс i, такой что $A[i] = x$. Чему равно математическое ожидание количества индексов в массиве A, которые будут проверены до того, как будет найден элемент x и процедура RANDOM_SEARCH завершит работу?

в) Обобщите решение сформулированной в части *б* задачи, при условии, что имеется $k \geqslant 1$ индексов i, таких что $A[i] = x$. Чему равно математическое ожидание количества индексов в массиве A, которые будут проверены до того, как будет найден элемент x и процедура RANDOM_SEARCH завершит работу? Ответ должен зависеть от величин n и k.

г) Предположим, что условие $A[i] = x$ не выполняется ни для какого элемента массива A. Чему равно математическое ожидание количества индексов в массиве A, которые придется проверить до того, как будут проверены все элементы и процедура RANDOM_SEARCH завершит работу?

Теперь рассмотрим детерминированный алгоритм линейного поиска DETERMINISTIC_SEARCH. В этом алгоритме поиск элемента x производится путем последовательной проверки элементов $A[1], A[2], \ldots, A[n]$ до тех пор, пока не произойдет одно из двух событий: либо будет найден элемент $A[i] = x$, либо будет достигнут конец массива. Предполагается, что все возможные перестановки элементов входного массива встречаются с одинаковой вероятностью.

д) Предположим, что имеется всего один индекс i, такой что $A[i] = x$. Чему равно математическое ожидание времени работы процедуры DETERMINISTIC_SEARCH? Как ведет себя эта величина в наихудшем случае?

е) Обобщите решение сформулированной в части *д* задачи, при условии, что имеется $k \geqslant 1$ индексов i, для которых $A[i] = x$. Чему равно математическое ожидание времени работы процедуры DETERMINISTIC_SEARCH? Как ведет себя эта величина в наихудшем случае? Ответ должен зависеть от величин n и k.

ж) Предположим, что условие $A[i] = x$ не выполняется ни для какого элемента массива A. Чему равно математическое ожидание времени работы процедуры DETERMINISTIC_SEARCH? Как ведет себя эта величина в наихудшем случае?

Наконец, рассмотрим рандомизированный алгоритм SCRAMBLE_SEARCH, в котором сначала выполняется случайная перестановка элементов входного массива, а затем в полученном массиве выполняется описанный выше линейный детерминированный поиск.

з) Пусть k — количество индексов i, таких что $A[i] = x$. Определите математическое ожидание времени работы процедуры SCRAMBLE_SEARCH и время ее работы в наихудшем случае для значений $k = 0$ и $k = 1$. Обобщите решение для случая $k \geqslant 1$.

и) Какой из трех представленных алгоритмов вы бы предпочли? Объясните свой ответ.

Заключительные замечания

Описание большого количества усовершенствованных вероятностных методов можно найти в книгах Боллобаса (Bollobás) [44], Гофри (Hofri) [151] и лекциях Спенсера (Spencer) [283]. Обзор и обсуждение преимуществ рандомизированных алгоритмов представлен в работах Карпа (Karp) [174] и Рабина (Rabin) [253]. Кроме того, рандомизированные алгоритмы подробно рассмотрены в учебнике Мотвани (Motwani) и Рагтвагана (Raghtvagan) [228].

Различные варианты задачи о найме сотрудника изучались многими исследователями. Этот класс задач более известен как “задачи о секретарях”. Примером работы в этой области является статья Айтаи (Ajtai), Меггидо (Meggido) и Ваартса (Waarts) [12].

ЧАСТЬ II

Сортировка и порядковая статистика

Введение

В этой части представлено несколько алгоритмов, с помощью которых можно решить следующую ***задачу сортировки***.

Вход: последовательность, состоящая из n чисел $\langle a_1, a_2, \ldots, a_n \rangle$.

Выход: перестановка (изменение порядка) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ входной последовательности таким образом, что $a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$.

Входная последовательность обычно представляется в виде n-элементного массива, хотя она может иметь и другое представление, например, в виде связанного списка.

Структура данных

На практике сортируемые числа редко являются изолированными значениями. Обычно каждое из них входит в состав набора данных, который называется ***записью*** (record). В каждой записи содержится ***ключ*** (key), представляющий собой сортируемое значение, в то время как остальная часть записи нередко состоит из сопутствующих данных, дополняющих ключ. Алгоритм сортировки на практике должен быть реализован так, чтобы он вместе с ключами переставлял и сопутствующие данные. Если каждая запись включает в себя сопутствующие данные большого объема, то с целью свести к минимуму перемещение данных сортировка часто производится не в исходном массиве, а в массиве указателей на записи.

В определенном смысле сделанное выше замечание относится к особенностям реализации, отличающим алгоритм от конечной программы. *Метод*, с помощью которого процедура сортировки размещает сортируемые величины в нужном порядке, не зависит от того, сортируются ли отдельные числа или большие записи. Таким образом, если речь идет о задаче сортировки, обычно предполагается, что входные данные состоят только из чисел. Преобразование алгоритма, предназначенного для сортировки чисел, в программу для сортировки записей не представляет концептуальных трудностей, хотя в конкретной практической ситуации иногда возникают различные тонкие нюансы, усложняющие задачу программиста.

Почему сортировка?

Многие исследователи, работающие в области вычислительной техники, считают сортировку наиболее фундаментальной задачей при изучении алгоритмов. Тому есть несколько причин.

- Иногда в приложении не обойтись без сортировки информации. Например, чтобы подготовить отчет о состоянии счетов клиентов, банку необходимо выполнить сортировку чеков по их номерам.

- Часто в алгоритмах сортировка используется в качестве основной подпрограммы. Например, программе, выполняющей визуализацию перекрывающихся графических объектов, которые находятся на разных уровнях, сначала может понадобиться отсортировать эти объекты по уровням снизу вверх, чтобы установить порядок их вывода. В этой книге мы ознакомимся с многочисленными алгоритмами, в которых сортировка используется в качестве подпрограммы.
- Имеется большой выбор алгоритмов сортировки, в которых применяются самые разные технологии. Фактически в алгоритмах сортировки используются многие важные методы (зачастую разработанные еще на заре компьютерное эры), применяемые при разработке различных классов алгоритмов. В этом отношении задача сортировки представляет также исторический интерес.
- Сортировка — это задача, для которой можно доказать наличие нетривиальной нижней границы (что будет сделано в главе 8). Найденные верхние границы в асимптотическом пределе совпадают с нижней границей, откуда можно заключить, что наши алгоритмы сортировки являются асимптотически оптимальными. Кроме того, нижние оценки алгоритмов сортировки можно использовать для поиска нижних границ в некоторых других задачах.
- В процессе реализации алгоритмов сортировки на передний план выходят многие прикладные проблемы. Выбор наиболее производительной программы сортировки в той или иной ситуации может зависеть от многих факторов, таких как предварительные знания о ключах и сопутствующих данных, об иерархической организации памяти компьютера (наличии кэша и виртуальной памяти) и программной среды. Многие из этих вопросов можно рассматривать на уровне алгоритмов, а не кода.

Алгоритмы сортировки

В главе 2 читатель имел возможность ознакомиться с двумя алгоритмами, производящих сортировку n действительных чисел. Сортировка методом вставок в наихудшем случае выполняется за время $\Theta\left(n^2\right)$. Несмотря на далеко не самое оптимальное асимптотическое поведение этого алгоритма, благодаря компактности его внутренних циклов он быстро справляется с сортировкой массивов с небольшим количеством элементов “на месте” (без дополнительной памяти, т.е. без выделения отдельного массива для работы и хранения выходных данных). (Напомним, что сортировка выполняется без дополнительной памяти лишь тогда, когда алгоритму требуется только определенное постоянное количество элементов вне исходного массива.) Сортировка методом слияния в асимптотическом пределе ведет себя как $\Theta\left(n \lg n\right)$, что лучше, чем $\Theta\left(n^2\right)$, но процедура MERGE, которая используется в этом алгоритме, не работает без дополнительной памяти.

В этой части вы ознакомитесь еще с двумя алгоритмами, предназначенными для сортировки произвольных действительных чисел. Представленный в главе 6 метод пирамидальной сортировки позволяет отсортировать n чисел за время $O(n \lg n)$. В нем используется важная структура данных, пирамида, или куча (heap), которая также позволяет реализовать приоритетную очередь.

Рассмотренный в главе 7 алгоритм быстрой сортировки также сортирует n чисел "на месте", но время его работы в наихудшем случае равно $\Theta(n^2)$. Тем не менее, в среднем этот алгоритм выполняется за время $\Theta(n \lg n)$ и на практике по производительности превосходит алгоритм пирамидальной сортировки. Код алгоритма быстрой сортировки такой же компактный, как и код алгоритма сортировки вставкой, поэтому скрытый постоянный множитель, влияющий на величину времени работы этого алгоритма, довольно мал. Алгоритм быстрой сортировки приобрел широкую популярность для сортировки больших массивов.

Алгоритмы сортировки, работающие по методу вставок и слияния, а также алгоритмы пирамидальной и быстрой сортировки имеют одну общую особенность — все они работают по принципу попарного сравнения элементов входного массива. В начале главы 8 рассматривается модель дерева принятия решения, позволяющая исследовать ограничения производительности, присущие алгоритмам данного типа. С помощью этой модели доказывается, что в наихудшем случае нижняя оценка, ограничивающая время работы любого алгоритма, работающего по методу сравнения, равна $\Omega(n \lg n)$. Это означает, что алгоритмы пирамидальной сортировки и сортировки слиянием являются асимптотически оптимальными.

Далее в главе 8 показано, что нижнюю границу $\Omega(n \lg n)$ можно превзойти, если информацию о порядке расположения входных элементов можно получить методами, отличными от попарного сравнения. Например, в алгоритме сортировки, работающем по методу перечисления, предполагается, что входные данные образуют множество чисел $\{0, 1, 2, \ldots, k\}$. Используя в качестве инструмента для определения относительного порядка элементов массива механизм индексации, алгоритм сортировки перечислением может выполнить сортировку n чисел за время $\Theta(n + k)$. Таким образом, если $k = O(n)$, то время работы этого алгоритма линейно зависит от размера входного массива. Родственный алгоритм поразрядной сортировки может быть использован для расширения области применения сортировки перечислением. Если нужно выполнить сортировку n d-значных чисел, где каждая цифра может принимать до k возможных значений, то алгоритм поразрядной сортировки справится с этой задачей за время $\Theta(d(n + k))$. Если d — константа, а величина k ведет себя как $O(n)$, то время выполнения этого алгоритма линейно зависит от размера входного массива. Для применения третьего алгоритма, алгоритма блочной сортировки, необходимы знания о распределении чисел во входном массиве. Этот алгоритм позволяет выполнить сортировку n равномерно распределенных на полуоткрытом интервале $[0, 1)$ чисел в среднем за время $O(n)$.

Порядковая статистика

В множестве, состоящем из n чисел, i-й порядковой статистикой называется i-е по величине значение в порядке возрастания. Конечно же, i-ю порядковую статистику можно выбрать путем сортировки входных элементов и индексирования i-го значения в выходных данных. Если не делается никаких предположений о распределении входных элементов, время работы данного метода равно $\Omega(n \lg n)$, как следует из величины нижней границы, найденной в главе 8.

В главе 9 будет показано, что i-й по величине (в порядке возрастания) элемент можно найти за время $O(n)$, что справедливо даже для случая произвольных действительных чисел. Мы покажем в этой главе алгоритм с компактным псевдокодом, время работы которого в наихудшем случае равно $\Theta(n^2)$, а в среднем линейно возрастает с увеличением количества входных элементов. Там же описан и более сложный алгоритм, время работы которого даже в наихудшем случае равно $O(n)$.

Теоретическая подготовка

В основном в этой части не используется сложная математика, однако для освоения некоторых разделов требуется знание определенных разделов математики. В частности, при анализе алгоритмов быстрой сортировки, блочной сортировки и алгоритма порядковой статистики используются некоторые положения теории вероятности, рассмотренные в приложении В, а также материал по вероятностному анализу и рандомизированным алгоритмам из главы 5. Анализ алгоритма порядковой статистики в наихудшем случае содержит более сложные математические выкладки, чем используемые при анализе других рассмотренных в этой части алгоритмов.

ГЛАВА 6

Пирамидальная сортировка

В этой главе описывается еще один алгоритм сортировки, а именно — пирамидальная сортировка. Время работы этого алгоритма, как и время работы алгоритма сортировки слиянием (и в отличие от времени работы алгоритма сортировки вставкой), равно $O(n \lg n)$. Как и сортировка методом вставок, и в отличие от сортировки слиянием, пирамидальная сортировка выполняется без привлечения дополнительной памяти: в любой момент времени требуется память для хранения вне массива только некоторого постоянного количества элементов. Таким образом, в пирамидальной сортировке сочетаются лучшие особенности двух рассмотренных ранее алгоритмов сортировки.

В ходе рассмотрения пирамидальной сортировки мы также познакомимся с еще одним методом разработки алгоритмов, а именно — использованием специализированных структур данных для управления информацией в ходе выполнения алгоритма. В рассматриваемом случае такая структура данных называется "пирамидой" (heap) и может оказаться полезной не только при пирамидальной сортировке, но и при создании эффективной очереди с приоритетами. В последующих главах эта структура данных появится снова.

Изначально термин "heap" использовался в контексте пирамидальной сортировки (heapsort), но в последнее время его основной смысл изменился, и он стал обозначать память со сборкой мусора, в частности, в языках программирования Lisp и Java (и переводиться как "куча"). Однако в данной книге термину heap (который здесь переводится как "пирамида") возвращен его первоначальный смысл.

6.1 Пирамиды

Пирамида (binary heap) — это структура данных, представляющая собой объект-массив, который можно рассматривать как почти полное бинарное дерево (см. раздел 5.3 приложения Б, а также рис. 6.1). Каждый узел этого дерева соответствует определенному элементу массива. На всех уровнях, кроме, может быть, последнего, дерево полностью заполнено (заполненный уровень — это такой, который содержит максимально возможное количество узлов). Последний уровень заполняется слева направо до тех пор, пока в массиве не закончатся элементы. Представляющий пирамиду массив A является объектом с двумя атрибутами: $length\,[A]$, т.е. количество элементов массива, и $heap\_size\,[A]$, т.е. количество элементов пирамиды, содержащихся в массиве A. Другими словами, несмотря на то, что в массиве $A\,[1..\,length\,[A]]$ все элементы могут быть корректными числами, ни один из элементов, следующих после элемента $A\,[heap\_size\,[A]]$, где $heap\_size\,[A] \leqslant length\,[A]$, не является элементом пирамиды. В корне дерева находится элемент $A\,[1]$, а дальше оно строится по следующему принципу: если какому-то узлу соответствует индекс i, то индекс его родительского узла вычисляется с помощью представленной ниже процедуры PARENT(i), индекс левого дочернего узла — с помощью процедуры LEFT(i), а индекс правого дочернего узла — с помощью процедуры RIGHT(i):

```
PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

В пирамиде, представленной на рис. 6.1, число в окружности, представляющей каждый узел дерева, является значением, сортируемым в данном узле. Число над узлом — это соответствующий индекс массива. Линии, попарно соединяющие элементы массива, обозначают взаимосвязь вида “родитель-потомок”. Родительские элементы всегда расположены слева от дочерних. Данное дерево имеет высоту, равную 3; узел с индексом 4 (и значением 8) расположен на первом уровне.

На большинстве компьютеров операция $2i$ в процедуре LEFT выполняется при помощи одной команды процессора путем побитового сдвига числа i на один бит влево. Операция $2i + 1$ в процедуре RIGHT тоже выполняется очень быстро — достаточно биты двоичного представления числа i сдвинуть на одну позицию влево, а затем младший бит установить равным 1. Процедура PARENT выполняется путем сдвига числа i на один бит вправо. При реализации пирамидальной сортировки эти функции часто представляются в виде макросов или встраиваемых процедур.

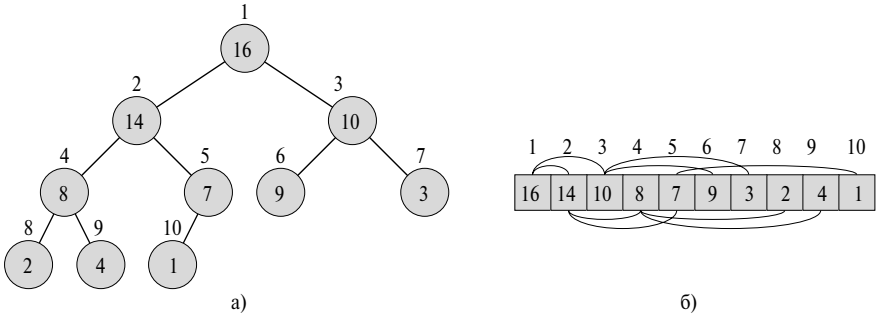


Рис. 6.1. Пирамида, представленная в виде *а*) бинарного дерева и *б*) массива

Различают два вида бинарных пирамид: неубывающие и невозрастающие. В пирамидах обоих видов значения, расположенные в узлах, удовлетворяют ***свойству пирамиды*** (heap property), являющемуся отличительной чертой пирамиды того или иного вида. ***Свойство невозрастающих пирамид*** (max-heap property) заключается в том, что для каждого отличного от корневого узла с индексом i выполняется следующее неравенство:

$$A\left[\textsc{Parent}\left(i\right)\right] \geqslant A\left[i\right].$$

Другими словами, значение узла не превышает значение родительского по отношению к нему узла. Таким образом, в невозрастающей пирамиде самый большой элемент находится в корне дерева, а значения узлов поддерева, берущего начало в каком-то элементе, не превышают значения самого этого элемента. Принцип организации ***неубывающей пирамиды*** (min-heap) прямо противоположный. Свойство ***неубывающих пирамид*** (min-heap property) заключается в том, что для всех отличных от корневого узлов с индексом i выполняется такое неравенство:

$$A\left[\textsc{Parent}\left(i\right)\right] \leqslant A\left[i\right].$$

Таким образом, наименьший элемент такой пирамиды находится в ее корне.

В алгоритме пирамидальной сортировки используются невозрастающие пирамиды. Неубывающие пирамиды часто применяются в приоритетных очередях (этот вопрос обсуждается в разделе 6.5). Для каждого приложения будет указано, с пирамидами какого вида мы будем иметь дело — с неубывающими или невозрастающими. При описании свойств как неубывающих, так и невозрастающих пирамид будет использоваться общий термин “пирамида”.

Рассматривая пирамиду как дерево, определим ***высоту*** ее узла как число ребер в самом длинном простом нисходящем пути от этого узла к какому-то из листьев дерева. Высота пирамиды определяется как высота его корня. Поскольку n-элементная пирамида строится по принципу полного бинарного дерева, то ее

высота равна $\Theta(\lg n)$ (см. упражнение 6.1-2). Мы сможем убедиться, что время выполнения основных операций в пирамиде приблизительно пропорционально высоте дерева, и, таким образом, эти операции требуют для работы время $O(\lg n)$. В остальных разделах этой главы представлены несколько базовых процедур и продемонстрировано их использование в алгоритме сортировки и в структуре данных очереди с приоритетами.

- Процедура MAX_HEAPIFY выполняется в течение времени $O(\lg n)$ и служит для поддержки свойства невозрастания пирамиды.
- Время выполнения процедуры BUILD_MAX_HEAP увеличивается с ростом количества элементов линейно. Эта процедура предназначена для создания невозрастающей пирамиды из неупорядоченного входного массива.
- Процедура HEAPSORT выполняется в течение времени $O(n \lg n)$ и сортирует массив без привлечения дополнительной памяти.
- Процедуры MAX_HEAP_INSERT, HEAP_EXTRACT_MAX, HEAP_INCREASE_KEY и HEAP_MAXIMUM выполняются в течение времени $O(\lg n)$ и позволяют использовать пирамиду в качестве очереди с приоритетами.

Упражнения

6.1-1. Чему равно минимальное и максимальное количество элементов в пирамиде высотой h?

6.1-2. Покажите, что n-элементная пирамида имеет высоту $\lfloor \lg n \rfloor$.

6.1-3. Покажите, что в любом поддереве невозрастающей пирамиды значение корня этого поддерева не превышает значений, содержащихся в других его узлах.

6.1-4. Где в невозрастающей пирамиде может находиться наименьший ее элемент, если все элементы различаются по величине?

6.1-5. Является ли массив с отсортированными элементами неубывающей пирамидой?

6.1-6. Является ли последовательность $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ невозрастающей пирамидой?

6.1-7. Покажите, что если n-элементную пирамиду представить в виде массива, то ее листьями будут элементы с индексами $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2 Поддержка свойства пирамиды

Процедура MAX_HEAPIFY является важной подпрограммой, предназначенной для работы с элементами невозрастающих пирамид. На ее вход подается массив A и индекс i этого массива. При вызове процедуры MAX_HEAPIFY предполагается, что бинарные деревья, корнями которых являются элементы LEFT(i) и RIGHT(i), являются невозрастающими пирамидами, но сам элемент $A[i]$ может быть меньше своих дочерних элементов, нарушая тем самым свойство невозрастающей пирамиды. Функция MAX_HEAPIFY опускает значение элемента $A[i]$ вниз по пирамиде до тех пор, пока поддерево с корнем, отвечающем индексу i, не становится невозрастающей пирамидой:

```
MAX_HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ⩽ heap_size[A] и A[l] > A[i]
 4     then largest ← l
 5     else largest ← i
 6  if r ⩽ heap_size[A] и A[r] > A[largest]
 7     then largest ← r
 8  if largest ≠ i
 9     then Обменять A[i] ↔ A[largest]
10          MAX_HEAPIFY(A, largest)
```

Действие процедуры MAX_HEAPIFY проиллюстрировано на рис. 6.2. На каждом этапе ее работы определяется, какой из элементов — $A[i]$, $A[Left(i)]$ или $A[Right(i)]$ — является максимальным, и его индекс присваивается переменной $largest$. Если наибольший элемент — $A[i]$, то поддерево, корень которого находится в узле с индексом i, — невозрастающая пирамида, и процедура завершает свою работу. В противном случае максимальное значение имеет один из дочерних элементов, и элемент $A[i]$ меняется местами с элементом $A[largest]$. После этого узел с индексом i и его дочерние узлы станут удовлетворять свойству невозрастающей пирамиды. Однако теперь исходное значение элемента $A[i]$ присвоено элементу с индексом $largest$, и свойство невозрастающей пирамиды может нарушиться в поддереве с этим корнем. Для устранения нарушения для этого дерева необходимо рекурсивно вызвать процедуру MAX_HEAPIFY.

На рис. 6.2 показана работа процедуры MAX_HEAPIFY$(A, 2)$. В части a этого рисунка показана начальная конфигурация, в которой элемент $A[2]$ нарушает свойство невозрастающей пирамиды, поскольку он меньше, чем оба дочерних узла. Поменяв местами элементы $A[2]$ и $A[4]$, мы восстанавливаем это свойство в узле 2, но нарушаем его в узле 4 (часть *б* рисунка). Теперь в рекурсивном вызове процедуры MAX_HEAPIFY$(A, 4)$ в качестве параметра выступает значение

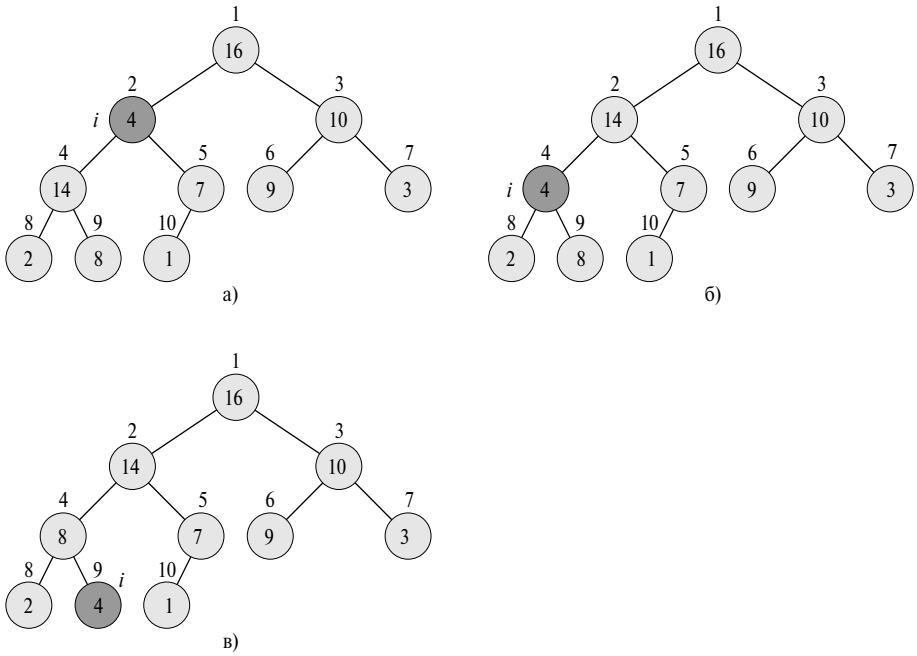


Рис. 6.2. Работа процедуры MAX_HEAPIFY$(A, 2)$ при $heap\_size\,[A] = 10$

$i = 4$. После перестановки элементов $A\,[4]$ и $A\,[9]$ (часть *в* рисунка) ситуация в узле 4 исправляется, а рекурсивный вызов процедуры MAX_HEAPIFY$(A, 9)$ не вносит никаких изменений в рассматриваемую структуру данных.

Время работы процедуры MAX_HEAPIFY на поддереве размера n с корнем в заданном узле i вычисляется как время $\Theta\,(1)$, необходимое для исправления отношений между элементами $A\,[i]$, $A\,[Left\,(i)]$ или $A\,[Right\,(i)]$, плюс время работы этой процедуры с поддеревом, корень которого находится в одном из дочерних узлов узла i. Размер каждого из таких дочерних поддеревьев не превышает величину $2n/3$, причем наихудший случай — это когда последний уровень заполнен наполовину. Таким образом, время работы процедуры MAX_HEAPIFY описывается следующим рекуррентным соотношением:

$$T\,(n) \leqslant T\,(2n/3) + \Theta\,(1)\,.$$

Решение этого рекуррентного соотношения, в соответствии со вторым случаем основной теоремы (теоремы 4.1), равно $T\,(n) = O\,(\lg n)$. По-другому время работы процедуры MAX_HEAPIFY с узлом, который находится на высоте h, можно выразить как $O\,(h)$.

Упражнения

6.2-1. Используя в качестве модели рис. 6.2, проиллюстрируйте работу процедуры MAX_HEAPIFY$(A, 3)$ с массивом $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$.

6.2-2. Используя в качестве отправной точки процедуру MAX_HEAPIFY, составьте псевдокод процедуры MIN_HEAPIFY(A, i), выполняющей соответствующие действия в неубывающей пирамиде. Сравните время работы этих двух процедур.

6.2-3. Как работает процедура MAX_HEAPIFY(A, i) в случае, когда элемент $A[i]$ больше своих дочерних элементов?

6.2-4. К чему приведет вызов процедуры MAX_HEAPIFY(A, i) при $i > > heap\_size\,[A]/2$?

6.2-5. Код процедуры MAX_HEAPIFY достаточно рационален, если не считать рекурсивного вызова в строке 10, из-за которого некоторые компиляторы могут сгенерировать неэффективный код. Напишите эффективную процедуру MAX_HEAPIFY, в которой вместо рекурсивного вызова использовалась бы итеративная управляющая конструкция (цикл).

6.2-6. Покажите, что в наихудшем случае время работы процедуры MAX_HEAPIFY на пирамиде размера n равно $\Omega(\lg n)$. (*Указание*: в пирамиде с n узлами присвойте узлам такие значения, чтобы процедура MAX_HEAPIFY рекурсивно вызывалась в каждом узле, расположенном на пути от корня до одного из листьев.)

6.3 Создание пирамиды

С помощью процедуры MAX_HEAPIFY можно преобразовать массив $A[1..n]$, где $n = length\,[A]$, в невозрастающую пирамиду в направлении снизу вверх. Из упражнения 6.1-7 известно, что все элементы подмассива $A[(\lfloor n/2\rfloor + 1)\,..n]$ являются листьями дерева, поэтому каждый из них можно считать одноэлементной пирамидой, с которой можно начать процесс построения. Процедура BUILD_MAX_HEAP проходит по остальным узлам и для каждого из них выполняет процедуру MAX_HEAPIFY:

```
BUILD_MAX_HEAP(A)
1  heap_size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3      do MAX_HEAPIFY(A, i)
```

Пример работы процедуры BUILD_MAX_HEAP показан на рис. 6.3. В части *a* этого рисунка изображен 10-элементный входной массив A и представляющее

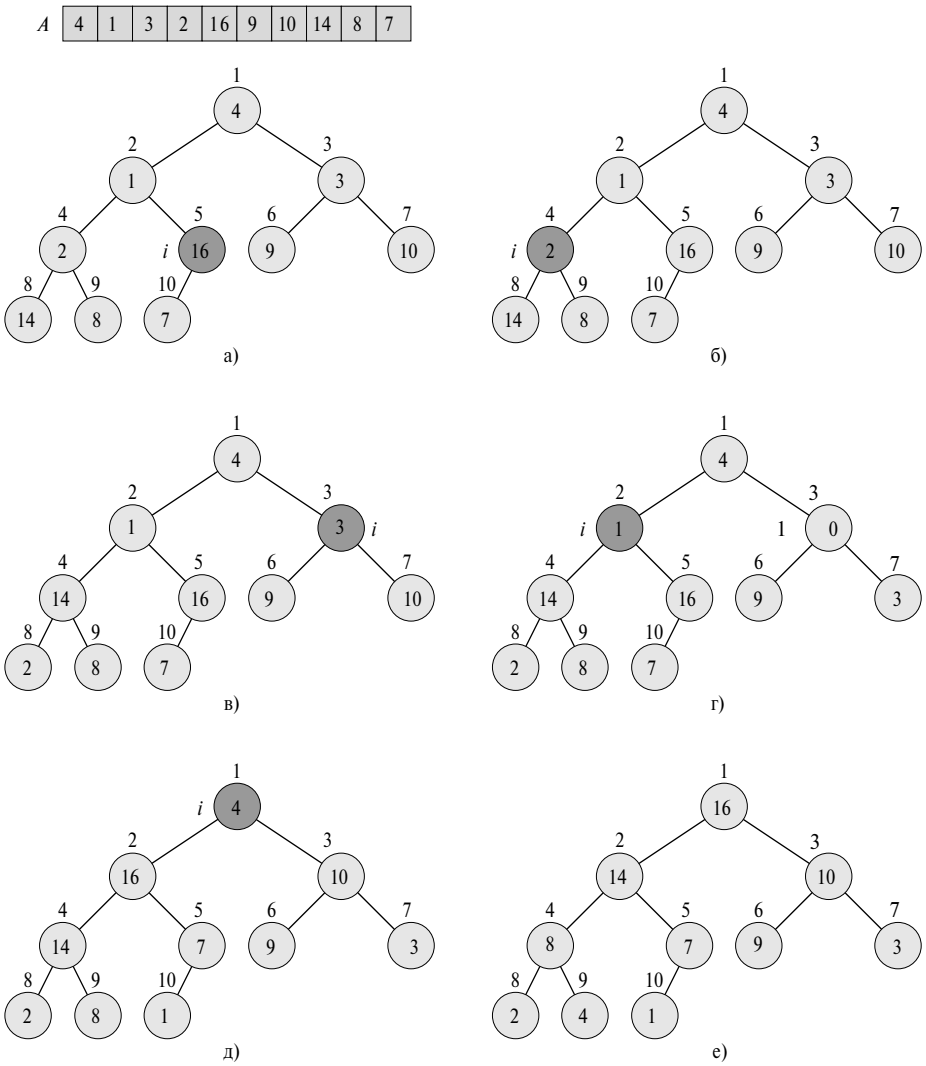


Рис. 6.3. Схема работы процедуры BUILD_MAX_HEAP

его бинарное дерево. Из этого рисунка видно, что перед вызовом процедуры MAX_HEAPIFY(A, i) индекс цикла i указывает на 5-й узел. Получившаяся в результате структура данных показана в части *б*. В следующей итерации индекс цикла i указывает на узел 4. Последующие итерации цикла **for** в процедуре BUILD_MAX_HEAP показаны в частях *в–д* рисунка. Обратите внимание, что при вызове процедуры MAX_HEAPIFY для любого узла поддеревья с корнями в его дочерних узлах являются невозрастающими пирамидами. В части *е* показана невозрастающая пирамида, полученная в результате работы процедуры BUILD_MAX_HEAP.

Чтобы показать, что процедура BUILD_MAX_HEAP работает корректно, воспользуемся сформулированным ниже инвариантом цикла.

> Перед каждой итерацией цикла **for** в строках 2–3 процедуры BUILD_MAX_HEAP все узлы с индексами $i+1, i+2, \ldots, n$ являются корнями невозрастающих пирамид.

Необходимо показать, что этот инвариант справедлив перед первой итерацией цикла, что он сохраняется при каждой итерации, и что он позволяет продемонстрировать корректность алгоритма после его завершения.

Инициализация. Перед первой итерацией цикла $i = \lfloor n/2 \rfloor$. Все узлы с индексами $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ — листья, поэтому каждый из них является корнем тривиальной невозрастающей пирамиды.

Сохранение. Чтобы убедиться, что каждая итерация сохраняет инвариант цикла, заметим, что дочерние по отношению к узлу i имеют номера, которые больше i. В соответствии с инвариантом цикла, оба эти узла являются корнями невозрастающих пирамид. Это именно то условие, которое требуется для вызова процедуры MAX_HEAPIFY(A, i), чтобы преобразовать узел с индексом i в корень невозрастающей пирамиды. Кроме того, при вызове процедуры MAX_HEAPIFY сохраняется свойство пирамиды, заключающееся в том, что все узлы с индексами $i+1, i+2, \ldots, n$ являются корнями невозрастающих пирамид. Уменьшение индекса i в цикле **for** обеспечивает выполнение инварианта цикла для следующей итерации.

Завершение. После завершения цикла $i = 0$. В соответствии с инвариантом цикла, все узлы с индексами $1, 2, \ldots, n$ являются корнями невозрастающих пирамид. В частности, таким корнем является узел 1.

Простую верхнюю оценку времени работы процедуры BUILD_MAX_HEAP можно получить следующим простым способом. Каждый вызов процедуры MAX_HEAPIFY занимает время $O(\lg n)$, и всего имеется $O(n)$ таких вызовов. Таким образом, время работы алгоритма равно $O(n \lg n)$. Эта верхняя граница вполне корректна, однако не является асимптотически точной.

Чтобы получить более точную оценку, заметим, что время работы MAX_HEAPIFY в том или ином узле зависит от высоты этого узла, и при этом большинство узлов расположено на малой высоте. При более тщательном анализе принимается во внимание тот факт, что высота n-элементной пирамиды равна $\lfloor \lg n \rfloor$ (упражнение 6.1-2) и что на любом уровне, находящемся на высоте h, содержится не более $\lceil n/2^{h+1} \rceil$ узлов (упражнение 6.3-3).

Время работы процедуры MAX_HEAPIFY при ее вызове для работы с узлом, который находится на высоте h, равно $O(h)$, поэтому верхнюю оценку полного времени работы процедуры BUILD_MAX_HEAP можно записать следующим

образом:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right).$$

Сумму в последнем выражении можно оценить, подставив $x = 1/2$ в формулу (A.8), в результате чего получим:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2.$$

Таким образом, время работы процедуры BUILD_MAX_HEAP можно ограничить следующим образом:

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n).$$

Получается, что время, которое требуется для преобразования неупорядоченного массива в невозрастающую пирамиду, линейно зависит от размера входных данных.

Неубывающую пирамиду можно создать с помощью процедуры BUILD_MIN_HEAP, полученной в результате преобразования процедуры BUILD_MAX_HEAP путем замены в строке 3 вызова функции MAX_HEAPIFY вызовом функции MIN_HEAPIFY (см. упражнение 6.2-2). Процедура BUILD_MIN_HEAP создает неубывающую пирамиду из неупорядоченного линейного массива за время, линейно зависящее от размера входных данных.

Упражнения

6.3-1. Используя в качестве модели рис. 6.3, проиллюстрируйте работу процедуры BUILD_MAX_HEAP с входным массивом $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2. Почему индекс цикла i в строке 2 процедуры BUILD_MAX_HEAP убывает от $\lfloor length[A]/2 \rfloor$ до 1, а не возрастает от 1 до $\lfloor length[A]/2 \rfloor$?

6.3-3. Покажите, что в любой n-элементной пирамиде на высоте h находится не более $\lceil n/2^{h+1} \rceil$ узлов.

6.4 Алгоритм пирамидальной сортировки

Работа алгоритма пирамидальной сортировки начинается с вызова процедуры BUILD_MAX_HEAP, с помощью которой из входного массива $A[1..n]$, где

$n = length\,[A]$, создается невозрастающая пирамида. Поскольку наибольший элемент массива находится в корне, т.е. в элементе $A\,[1]$, его можно поместить в окончательную позицию в отсортированном массиве, поменяв его местами с элементом $A\,[n]$. Выбросив из пирамиды узел n (путем уменьшения на единицу величины $heap\_size\,[A]$), мы обнаружим, что подмассив $A\,[1..\,(n-1)]$ легко преобразуется в невозрастающую пирамиду. Пирамиды, дочерние по отношению к корневому узлу, после обмена элементов $A\,[1]$ и $A\,[n]$ и уменьшения размера массива остаются невозрастающими, однако новый корневой элемент может нарушить свойство невозрастания пирамиды. Для восстановления этого свойства достаточно вызвать процедуру MAX_HEAPIFY$(A, 1)$, после чего подмассив $A\,[1..\,(n-1)]$ превратится в невозрастающую пирамиду. Затем алгоритм пирамидальной сортировки повторяет описанный процесс для невозрастающих пирамид размера $n-1, n-2, \ldots, 2$. (См. упражнение 6.4-2, посвященное точной формулировке инварианта цикла данного алгоритма.)

```
HEAPSORT(A)
1  BUILD_MAX_HEAP(A)
2  for i ← length[A] downto 2
3      do Обменять A[1] ↔ A[i]
4         heap_size[A] ← heap_size[A] − 1
5         MAX_HEAPIFY(A, 1)
```

На рис. 6.4 показан пример пирамидальной сортировки после предварительного построения невозрастающей пирамиды. В каждой части этого рисунка изображена невозрастающая пирамида перед выполнением очередной итерации цикла **for** в строках 2–5. В части *а*) этого рисунка показана исходная невозрастающая пирамида, полученная при помощи процедуры BUILD_MAX_HEAP. В частях *б*)–*к*) показаны пирамиды, получающиеся в результате вызова процедуры MAX_HEAPIFY в строке 5. В каждой из этих частей указано значение индекса i. В пирамиде содержатся только узлы, закрашенные светло-серым цветом. В части *л*) показан получившийся в конечном итоге массив A.

Время работы процедуры HEAPSORT равно $O\,(n \lg n)$, поскольку вызов процедуры BUILD_MAX_HEAP требует времени $O\,(n)$, а каждый из $n-1$ вызовов процедуры MAX_HEAPIFY — времени $O\,(\lg n)$.

Упражнения

6.4-1. Используя в качестве модели рис. 6.4, проиллюстрируйте работу процедуры HEAPSORT с входным массивом $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

6.4-2. Докажите корректность процедуры HEAPSORT с помощью сформулированного ниже инварианта цикла.

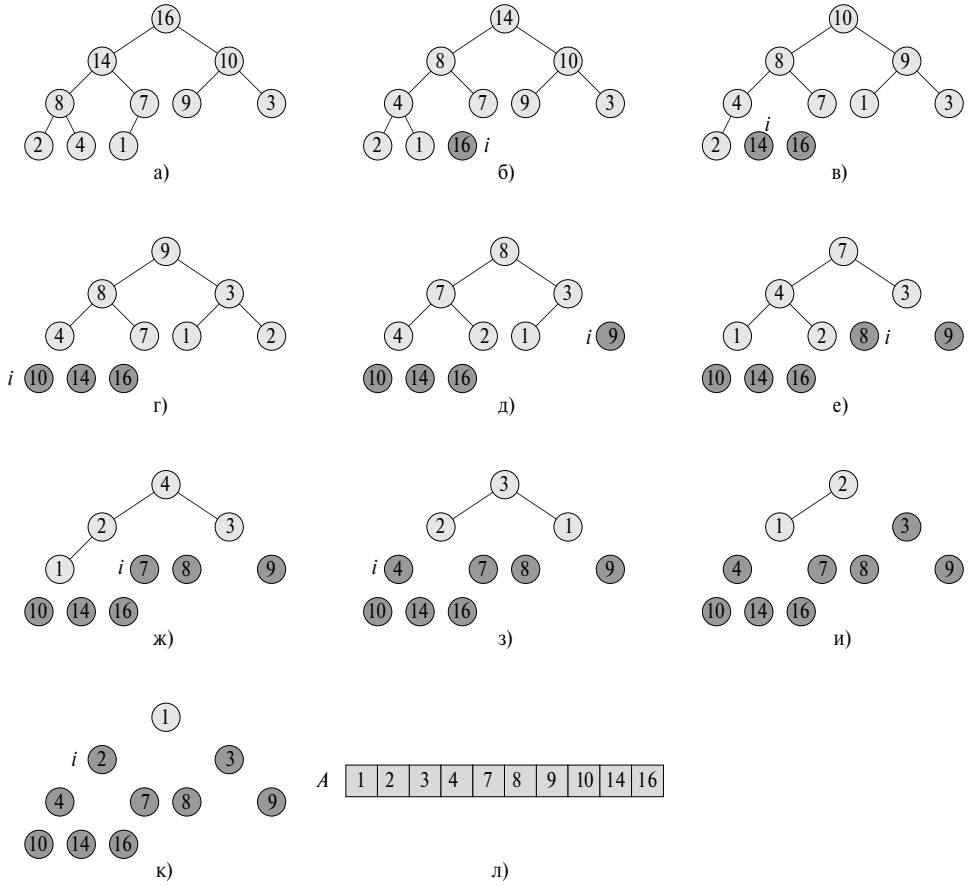


Рис. 6.4. Работа процедуры HEAPSORT

В начале каждой итерации цикла **for** в строках 2–5 подмассив $A[1..i]$ является невозрастающей пирамидой, содержащей i наименьших элементов массива $A[1..n]$, а в подмассиве $A[i+1..n]$ находятся $n-i$ отсортированных наибольших элементов массива $A[1..n]$.

6.4-3. Чему равно время работы алгоритма пирамидальной сортировки массива A длины n, в котором элементы отсортированы и расположены в порядке возрастания? В порядке убывания?

6.4-4. Покажите, что время работы алгоритма пирамидальной сортировки в наихудшем случае равно $\Omega(n \lg n)$.

⋆ 6.4-5. Покажите, что для массива, все элементы которого различны, время работы пирамидальной сортировки в наилучшем случае равно $\Omega(n \lg n)$.

6.5 Очереди с приоритетами

Пирамидальная сортировка — превосходный алгоритм, однако качественная реализация алгоритма быстрой сортировки, представленного в главе 7, на практике обычно превосходит по производительности пирамидальную сортировку. Тем не менее, структура данных, использующаяся при пирамидальной сортировке, сама по себе имеет большую ценность. В этом разделе представлено одно из наиболее популярных применений пирамид — в качестве эффективных очередей с приоритетами. Как и пирамиды, очереди с приоритетами бывают двух видов: невозрастающие и неубывающие. Мы рассмотрим процесс реализации невозрастающих очередей с приоритетами, которые основаны на невозрастающих пирамидах; в упражнении 6.5-3 требуется написать процедуры для неубывающих очередей с приоритетами.

Очередь с приоритетами (priority queue) — это структура данных, предназначенная для обслуживания множества S, с каждым элементом которого связано определенное значение, называющееся ***ключом*** (key). В ***невозрастающей очереди с приоритетами*** поддерживаются следующие операции.

- Операция INSERT(S, x) вставляет элемент x в множество S. Эту операцию можно записать как $S \leftarrow S \cup \{x\}$.
- Операция MAXIMUM(S) возвращает элемент множества S с наибольшим ключом.
- Операция EXTRACT_MAX(S) возвращает элемент с наибольшим ключом, удаляя его при этом из множества S.
- Операция INCREASE_KEY(S, x, k) увеличивает значение ключа, соответствующего элементу x, путем его замены ключом со значением k. Предполагается, что величина k не меньше текущего ключа элемента x.

Одна из областей применения невозрастающих очередей — планирование заданий на компьютере, который совместно используется несколькими пользователями. Очередь позволяет следить за заданиями, которые подлежат выполнению, и за их приоритетами. Если задание прервано или завершило свою работу, из очереди с помощью операции EXTRACT_MAX выбирается следующее задание с наибольшим приоритетом. В очередь в любое время можно добавить новое задание, воспользовавшись операцией HEAP_INSERT.

В ***неубывающей очереди с приоритетами*** поддерживаются операции INSERT, MINIMUM, EXTRACT_MIN и DECREASE_KEY. Очереди такого вида могут использоваться в моделировании систем, управляемых событиями. В роли элементов очереди в таком случае выступают моделируемые события, для каждого из которых сопоставляется время происшествия, играющее роль ключа. Элементы должны моделироваться последовательно согласно времени событий, поскольку

процесс моделирования может вызвать генерацию других событий. Моделирующая программа выбирает очередное моделируемое событие с помощью операции EXTRACT_MIN. Когда инициируются новые события, они помещаются в очередь с помощью процедуры INSERT. В главах 23 и 24 нам предстоит познакомиться и с другими случаями применения неубывающих очередей с приоритетами, когда особо важной становится роль операции DECREASE_KEY.

Не удивительно, что приоритетную очередь можно реализовать с помощью пирамиды. В каждом отдельно взятом приложении, например, в планировщике заданий, или при моделировании событий элементы очереди с приоритетами соответствуют объектам, с которыми работает это приложение. Часто возникает необходимость определить, какой из объектов приложения отвечает тому или иному элементу очереди, или наоборот. Если очередь с приоритетами реализуется с помощью пирамиды, то в каждом элементе пирамиды приходится хранить ***идентификатор*** (handle) соответствующего объекта приложения. То, каким будет конкретный вид этого идентификатора (указатель, целочисленный индекс или что-нибудь другое), — зависит от приложения. В каждом объекте приложения точно так же необходимо хранить идентификатор соответствующего элемента пирамиды. В данной книге таким идентификатором, как правило, будет индекс массива. Поскольку в ходе операций над пирамидой ее элементы изменяют свое расположение в массиве, при перемещении элемента пирамиды необходимо также обновлять значение индекса в соответствующем объекте приложения. Так как детали доступа к объектам приложения в значительной мере зависят от самого приложения и его реализации, мы не станем останавливаться на этом вопросе. Ограничимся лишь замечанием, что на практике необходима организация надлежащей обработки идентификаторов.

Теперь перейдем к реализации операций в невозрастающей очереди с приоритетами. Процедура HEAP_MAXIMUM реализует выполнение операции MAXIMUM за время $\Theta(1)$:

```
HEAP_MAXIMUM(A)
1  return A[1]
```

Процедура HEAP_EXTRACT_MAX реализует операцию EXTRACT_MAX. Она напоминает тело цикла **for** в строках 3–5 процедуры HEAPSORT:

```
HEAP_EXTRACT_MAX(A)
1  if heap_size[A] < 1
2     then error "Очередь пуста"
3  max ← A[1]
4  A[1] ← A[heap_size[A]]
5  heap_size[A] ← heap_size[A] − 1
6  MAX_HEAPIFY(A, 1)
7  return max
```

Время работы процедуры HEAP_EXTRACT_MAX равно $O(\lg n)$, поскольку перед запуском процедуры MAX_HEAPIFY, выполняющейся в течение времени $O(\lg n)$, в ней выполняется лишь определенная постоянная подготовительная работа.

Процедура HEAP_INCREASE_KEY реализует операцию INCREASE_KEY. Элемент очереди с приоритетами, ключ которого подлежит увеличению, идентифицируется в массиве с помощью индекса i. Сначала процедура обновляет ключ элемента $A[i]$. Поскольку это может нарушить свойство невозрастающих пирамид, после этого процедура проходит путь от измененного узла к корню в поисках надлежащего места для нового ключа. Эта операция напоминает реализованную в цикле процедуры INSERTION_SORT (строки 5–7) из раздела 2.1. В процессе прохода выполняется сравнение текущего элемента с родительским. Если оказывается, что ключ текущего элемента превышает значение ключа родительского элемента, то происходит обмен ключами элементов и процедура продолжает свою работу на более высоком уровне. В противном случае процедура прекращает работу, поскольку ей удалось восстановить свойство невозрастающих пирамид. (Точная формулировка инварианта цикла этого алгоритма приведена в упражнении 6.5-5.)

```
HEAP_INCREASE_KEY(A, i, key)
1  if key < A[i]
2     then error "Новый ключ меньше текущего"
3  A[i] ← key
4  while i > 1 и A[PARENT(i)] < A[i]
5        do Обменять A[i] ↔ A[PARENT(i)]
6           i ← PARENT(i)
```

На рис. 6.5 приведен пример работы процедуры HEAP_INCREASE_KEY. В части *а* этого рисунка изображена невозрастающая пирамида, в которой будет увеличен ключ узла, выделенного темно-серым цветом (кроме выделения цветом, возле этого узла указан индекс i). В части *б* рисунка показана эта же пирамида после того, как ключ выделенного узла был увеличен до 15. В части *в* обрабатываемая пирамида изображена после первой итерации цикла **while** (строки 4–6). Здесь видно, как текущий и родительский по отношению к нему узлы обменялись ключами, и индекс i перешел к родительскому узлу. В части *г* показана эта же пирамида после еще одной итерации цикла. Теперь условие невозрастающих пирамид выполняется, и процедура завершает работу. Время обработки n-элементной пирамиды с помощью этой процедуры равно $O(\lg n)$. Это объясняется тем, что длина пути от обновляемого в строке 3 элемента до корня равна $O(\lg n)$.

Процедура MAX_HEAP_INSERT реализует операцию INSERT. В качестве параметра этой процедуре передается ключ нового элемента. Сначала процедура добавляет в пирамиду новый лист и присваивает ему ключ со значением $-\infty$. Затем вызывается процедура HEAP_INCREASE_KEY, которая присваивает корректное зна-

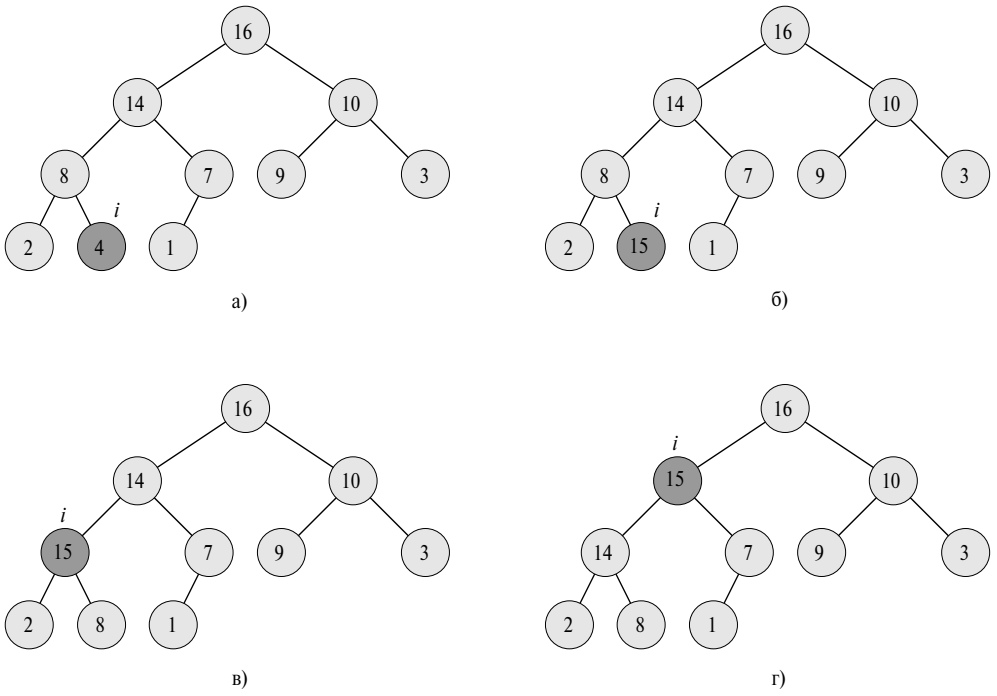


Рис. 6.5. Работа процедуры HEAP_INCREASE_KEY

чение ключу и помещает его в надлежащее место, чтобы не нарушалось свойство невозрастающих пирамид:

MAX_HEAP_INSERT(A, key)
1 $heap\_size[A] \leftarrow heap\_size[A] + 1$
2 $A[heap\_size[A]] \leftarrow -\infty$
3 HEAP_INCREASE_KEY$(A, heap_size[A], key)$

Время вставки в n-элементную пирамиду с помощью процедуры MAX_HEAP_INSERT составляет $O(\lg n)$.

Подводя итог, заметим, что в пирамиде время выполнения всех операций по обслуживанию очереди с приоритетами равно $O(\lg n)$.

Упражнения

6.5-1. Проиллюстрируйте работу процедуры HEAP_EXTRACT_MAX с пирамидой $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-2. Проиллюстрируйте работу процедуры MAX_HEAP_INSERT$(A, 10)$ с пирамидой $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$. Воспользуйтесь в качестве модели рисунком 6.5.

6.5-3. Напишите псевдокод процедур HEAP_MINIMUM, HEAP_EXTRACT_MIN, HEAP_DECREASE_KEY и MIN_HEAP_INSERT, реализующих на базе неубывающей пирамиды неубывающую очередь с приоритетами.

6.5-4. Зачем нужна такая мера предосторожности, как присвоение ключу добавляемого в пирамиду узла в строке 2 процедуры MAX_HEAP_INSERT значения $-\infty$, если уже при следующем шаге значение этого ключа увеличивается до требуемой величины?

6.5-5. Обоснуйте корректность процедуры HEAP_INCREASE_KEY с помощью следующего инварианта цикла.

> Перед каждой итерацией цикла **while** в строках 4–6, массив $A[1..\,heap\_size\,[A]]$ удовлетворяет свойству невозрастающих пирамид, за исключением одного возможного нарушения — элемент $A[i]$ может быть больше элемента $A[Parent\,(i)]$.

6.5-6. Покажите, как с помощью очереди с приоритетами реализовать очередь "первым вошел — первым вышел". Продемонстрируйте, как с помощью очереди с приоритетами реализовать стек. (Очереди и стеки определены в разделе 10.1.)

6.5-7. Процедура HEAP_DELETE(A, i) удаляет из пирамиды A узел i. Разработайте реализацию этой процедуры, которой требуется время $O\,(\lg n)$ для удаления узла из n-элементной невозрастающей пирамиды.

6.5-8. Разработайте алгоритм, объединяющий k отсортированных списков в один список за время $O\,(n \lg k)$, где n — общее количество элементов во всех входных списках. (*Указание*: для слияния списков воспользуйтесь неубывающей пирамидой.)

Задачи

6-1. **Создание пирамиды с помощью вставок**

Описанную в разделе 6.3 процедуру BUILD_MAX_HEAP можно реализовать путем многократного использования процедуры MAX_HEAP_INSERT для вставки элементов в пирамиду. Рассмотрим следующую реализацию:

```
BUILD_MAX_HEAP′(A)
1  heap_size[A] ← 1
2  for i ← 2 to length[A]
3        do MAX_HEAP_INSERT(A, A[i])
```

а) Всегда ли процедуры BUILD_MAX_HEAP и BUILD_MAX_HEAP′ для одного и того же входного массива создают одинаковые пирамиды? Докажите, что это так, или приведите контрпример.

б) Покажите, что в наихудшем случае для создания n-элементной пирамиды процедуре BUILD_MAX_HEAP′ требуется время $\Theta(n \lg n)$.

6-2. **Анализ пирамид, отличных от бинарных**

***d*-арные пирамиды** (d-ary heap) похожи на бинарные, с тем исключением, что узлы, отличные от листьев, имеют не по 2, а по d дочерних элементов.

а) Как бы вы представили d-арную пирамиду в виде массива?

б) Как выражается высота d-арной n-элементной пирамиды через n и d?

в) Разработайте эффективную реализацию процедуры EXTRACT_MAX, предназначенную для работы с d-арной невозрастающей пирамидой. Проанализируйте время работы этой процедуры и выразите его в терминах d и n.

г) Разработайте эффективную реализацию процедуры INSERT, предназначенную для работы с d-арной невозрастающей пирамидой. Проанализируйте время работы этой процедуры и выразите его в терминах d и n.

д) Разработайте эффективную реализацию процедуры INCREASE_KEY(A, i, k), в которой сначала выполняется присвоение $A[i] \leftarrow \max(A[i], k)$, а затем — обновление d-арной невозрастающей пирамиды. Проанализируйте время работы этой процедуры и выразите его в терминах d и n.

6-3. **Таблицы Юнга**

Таблица Юнга (Young tableau) $m \times n$ — это матрица $m \times n$, элементы которой в каждой строке отсортированы слева направо, а в каждом столбце — сверху вниз. Некоторые элементы таблицы Юнга могут быть равны ∞, что трактуется как отсутствие элемента. Таким образом, в таблице Юнга можно разместить $r \leqslant mn$ конечных чисел.

а) Начертите таблицу Юнга 4×4, содержащую элементы $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

б) Докажите, что таблица Юнга Y размером $m \times n$ пустая, если $Y[1, 1] = \infty$. Докажите, что таблица Y полностью заполнена (т.е. она содержит mn элементов), если $Y[m, n] < \infty$.

в) Разработайте алгоритм, реализующий процедуру EXTRACT_MIN для непустой таблицы Юнга $m \times n$ за время $O(m + n)$. В алгоритме следует использовать рекурсивную подпрограмму, которая решает задачу размером $m \times n$ путем рекурсивного сведения к задачам $(m - 1) \times n$ или $m \times (n - 1)$. (*Указание*: вспомните о процедуре

MAX_HEAPIFY.) Обозначим максимальное время обработки произвольной таблицы Юнга $m \times n$ с помощью процедуры EXTRACT_MIN как $T(p)$, где $p = m + n$. Запишите и решите рекуррентное соотношение, которое дает границу для $T(p)$, равную $O(m + n)$.

г) Покажите, как вставить новый элемент в незаполненную таблицу Юнга размером $m \times n$ за время $O(m + n)$.

д) Покажите, как с помощью таблицы Юнга $n \times n$ выполнить сортировку n^2 чисел за время $O(n^3)$, не используя при этом никаких других подпрограмм сортировки.

е) Разработайте алгоритм, позволяющий за время $O(m + n)$ определить, содержится ли в таблице Юнга размером $m \times n$ заданное число.

Заключительные замечания

Алгоритм пирамидальной сортировки был предложен Вильямсом (Williams) [316], который также описал, каким образом с помощью пирамиды можно реализовать очередь с приоритетами. Процедура BUILD_MAX_HEAP разработана Флойдом (Floyd) [90].

В главах 16, 23 и 24 будут реализованы очереди с приоритетами с помощью неубывающих пирамид. В главах 19 и 20 вы познакомитесь с реализацией некоторых операций с улучшенными временны́ми характеристиками.

Для целочисленных данных можно реализовать очереди с приоритетами, работающие быстрее, чем те, в которых не делаются предварительные предположения о типе данных. В структуре данных, предложенной Боасом (van Emde Boas) [301], поддерживаются операции MINIMUM, MAXIMUM, INSERT, DELETE, SEARCH, EXTRACT_MIN, EXTRACT_MAX, PREDECESSOR и SUCCESSOR. При условии, что ключи выбираются из множества $\{1, 2, \ldots, C\}$, время работы этих операций в наихудшем случае оценивается как $O(\lg \lg C)$. Фредман (Fredman) и Виллард (Willard) [99] показали, как реализовать операцию MINIMUM со временем работы $O(1)$ и операции INSERT и EXTRACT_MIN со временем работы $O\left(\sqrt{\lg n}\right)$, если данные представляют собой b-битовые целые числа, а память компьютера состоит из адресуемых b-битовых слов. В работе Торупа (Thorup) [299] граница $O\left(\sqrt{\lg n}\right)$ была улучшена до $O(\lg \lg n)$. При этом используемая память не ограничена величиной n, однако такого линейного ограничения используемой памяти можно достичь с помощью рандомизированного хеширования.

Важный частный случай очередей с приоритетами имеет место, когда последовательность операций EXTRACT_MIN является ***монотонной***, т.е. возвращаемые этой операцией последовательные значения образуют монотонную возрастающую

последовательность. Такая ситуация встречается во многих важных приложениях, например, в алгоритме поиска кратчайшего пути Дейкстры (Dijkstra), который рассматривается в главе 24, и при моделировании дискретных событий. Для алгоритма Дейкстры особенно важна эффективность операции DECREASE_KEY. Для частного случая монотонности для целочисленных данных из диапазона 1, 2, ..., C, Ахуйя (Ahuja), Мельхорн (Mehlhorn), Орлин (Orlin) и Таржан (Tarjan) [8] описали, как с помощью структуры данных под названием позиционная пирамида (radix heap) реализовать операции EXTRACT_MIN и INSERT со временем выполнения $O(\lg C)$ (более подробные сведения на эту тему можно найти в главе 17), и операцию DECREASE_KEY со временем выполнения $O(1)$. Граница $O(\lg C)$ может быть улучшена до $O\left(\sqrt{\lg C}\right)$ путем совместного использования пирамид Фибоначчи (см. главу 20) и позиционных пирамид. Дальнейшее улучшение этой границы до $O\left(\lg^{1/3+\varepsilon} C\right)$ было осуществлено Черкасским (Cherkassky), Гольдбергом (Goldberg) и Сильверстайном (Silverstein) [58], которые объединили многоуровневую группирующую структуру (multi-level bucketing structure) Денардо (Denardo) и Фокса (Fox) [72] с уже упоминавшейся пирамидой Торупа. Раману (Raman) [256] удалось еще больше улучшить эти результаты и получить границу, которая равна $O\left(\min\left(\lg^{1/4+\varepsilon} C, \lg^{1/3+\varepsilon} n\right)\right)$ для произвольной фиксированной величины $\varepsilon > 0$. Более подробное обсуждение этих результатов можно найти в работах Рамана [256] и Торупа [299].

ГЛАВА 7

Быстрая сортировка

Быстрая сортировка — это алгоритм сортировки, время работы которого для входного массива из n чисел в наихудшем случае равно $\Theta\left(n^2\right)$. Несмотря на такую медленную работу в наихудшем случае, этот алгоритм на практике зачастую оказывается оптимальным благодаря тому, что в среднем время его работы намного лучше: $\Theta\left(n \lg n\right)$. Кроме того, постоянные множители, не учтенные в выражении $\Theta\left(n \lg n\right)$, достаточно малы по величине. Алгоритм обладает также тем преимуществом, что сортировка в нем выполняется без использования дополнительной памяти, поэтому он хорошо работает даже в средах с виртуальной памятью.

В разделе 7.1 описан сам алгоритм и важная подпрограмма, использующаяся в нем. Поскольку поведение алгоритма быстрой сортировки достаточно сложное, мы начнем с нестрогого, интуитивного обсуждения производительности этого алгоритма в разделе 7.2, а строгий анализ отложим до конца данной главы. В разделе 7.3 представлена версия быстрой сортировки, в которой используется случайная выборка. У этого алгоритма хорошее среднее время работы, при этом никакие конкретные входные данные не могут ухудшить его производительность до уровня наихудшего случая. Этот рандомизированный алгоритм анализируется в разделе 7.4, где показано, что время его работы в наихудшем случае равно $\Theta\left(n^2\right)$, а среднее время работы в предположении, что все элементы различны, составляет $O\left(n \lg n\right)$.

7.1 Описание быстрой сортировки

Быстрая сортировка, подобно сортировке слиянием, основана на парадигме "разделяй и властвуй", представленной в разделе 2.3.1. Ниже описан процесс сортировки подмассива $A[p..r]$, состоящий, как и все алгоритмы с использованием декомпозиции, из трех этапов.

Разделение. Массив $A[p..r]$ разбивается (путем переупорядочения его элементов) на два (возможно, пустых) подмассива $A[p..q-1]$ и $A[q+1..r]$. Каждый элемент подмассива $A[p..q-1]$ не превышает элемент $A[q]$, а каждый элемент подмассива $A[q+1..r]$ не меньше элемента $A[q]$. Индекс q вычисляется в ходе процедуры разбиения.

Покорение. Подмассивы $A[p..q-1]$ и $A[q+1..r]$ сортируются путем рекурсивного вызова процедуры быстрой сортировки.

Комбинирование. Поскольку подмассивы сортируются на месте, для их объединения не нужны никакие действия: весь массив $A[p..r]$ оказывается отсортирован.

Алгоритм сортировки реализуется представленной ниже процедурой:

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

Чтобы выполнить сортировку всего массива A, вызов процедуры должен иметь вид QUICKSORT$(A, 1, lehgth\,[A])$.

Разбиение массива

Ключевой частью рассматриваемого алгоритма сортировки является процедура PARTITION, изменяющая порядок элементов подмассива $A[p..r]$ без привлечения дополнительной памяти:

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4     do if A[j] ⩽ x
5           then i ← i + 1
6                Обменять A[i] ↔ A[j]
7   Обменять A[i + 1] ↔ A[r]
8  return i + 1
```

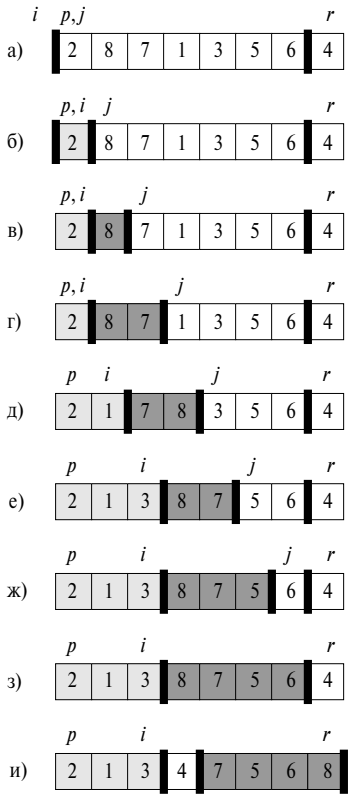


Рис. 7.1. Пример действия на массив процедуры PARTITION

На рис. 7.1 приведен пример работы процедуры PARTITION с 8-элементным массивом. Эта процедура всегда выбирает элемент $x = A[r]$ в качестве ***опорного*** (pivot) элемента. Разбиение подмассива $A[p..r]$ будет выполняться относительно этого элемента. В начале выполнения процедуры массив разделяется на четыре области (они могут быть пустыми). В начале каждой итерации цикла **for** в строках 3–6 каждая область удовлетворяет определенным свойствам, которые можно сформулировать в виде следующего инварианта цикла.

> В начале каждой итерации цикла в строках 3–6 для любого индекса k массива справедливо следующее:
>
> 1) если $p \leqslant k \leqslant i$, то $A[k] \leqslant x$;
> 2) если $i + 1 \leqslant k \leqslant j - 1$, то $A[k] > x$;
> 3) если $k = r$, то $A[k] = x$.

Соответствующая структура показана на рис. 7.2. Индексы между j и $r-1$ не подпадают ни под один из трех перечисленных случаев, и значения соответствующих им элементов никак не связаны с опорным элементом x.

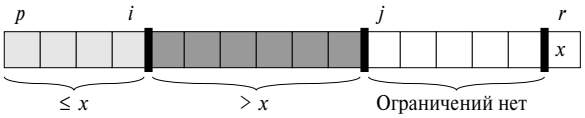


Рис. 7.2. Четыре области, поддерживаемые процедурой PARTITION в подмассиве $A[p..r]$; все элементы подмассива $A[p..i]$ меньше либо равны x, все элементы подмассива $A[i+1..j-1]$ больше x и $A[r] = x$, а все элементы подмассива $A[j..r-1]$ могут иметь любые значения

Перед тем как приступить к работе со сформулированным инвариантом цикла, давайте еще раз рассмотрим рис. 7.1. Светло-серым цветом обозначены элементы массива, которые попали в первую часть разбиения; значения всех этих элементов не превышают величину x. Элементы темно-серого цвета образуют вторую часть массива; их величина больше x. Незакрашенные элементы — это те, что не попали ни в одну из первых двух частей; последний незакрашенный элемент является опорным. В части *а* рисунка показано начальное состояние массива и значения переменных. Ни один элемент не помещен ни в одну из первых двух частей. В части *б* рисунка элемент со значением 2 "переставлен сам с собой" и помещен в ту часть, элементы которой не превышают x. В частях *в* и *г* элементы со значениями 8 и 7 добавлены во вторую часть массива (которая до этого момента была пустой). В части *д* элементы 1 и 8 поменялись местами, в результате чего количество элементов в первой части возросло. В части *е* меняются местами элементы 3 и 7, в результате чего количество элементов в первой части возрастает. В частях *ж* и *з* вторая часть увеличивается за счет включения в нее элементов 5 и 6, после чего цикл завершается. В части *и*, иллюстрирующей действие строк 7 и 8, опорный элемент меняется местами с тем, который находится на границе раздела двух областей.

Теперь нам необходимо показать, что сформулированный выше инвариант цикла справедлив перед первой итерацией, что каждая итерация цикла сохраняет этот инвариант и что инвариант позволяет продемонстрировать корректность алгоритма по завершении цикла.

Инициализация. Перед первой итерацией цикла $i = p - 1$ и $j = p$. Между элементами с индексами p и i нет никаких элементов, как нет их и между элементами с индексами $i + 1$ и $j - 1$, поэтому первые два условия инварианта цикла выполняются. Присваивание, которое выполняется в строке 1 процедуры, приводит к тому, что становится справедливым третье условие.

Сохранение. Как видно из рис. 7.3, нужно рассмотреть два случая, выбор одного из которых определяется проверкой в строке 4 алгоритма. На рис. 7.3*а* показано, что происходит, если $A[j] > x$; единственное действие, которое

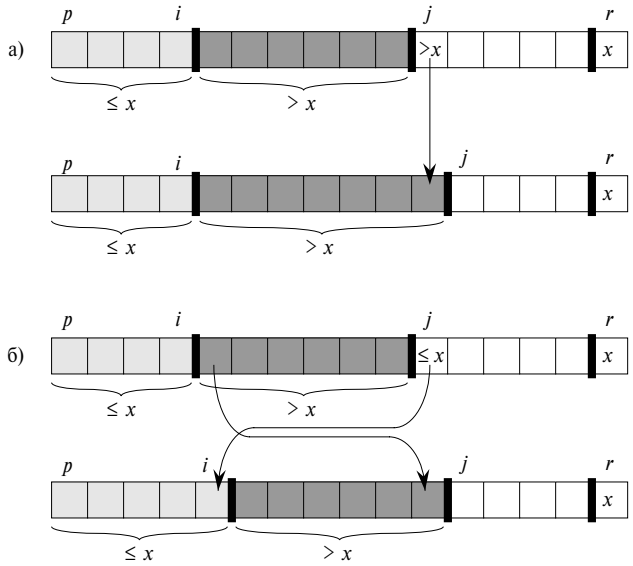


Рис. 7.3. Два варианта итерации процедуры PARTITION

выполняется в этом случае в цикле — увеличение на единицу значения j. При увеличении значения j для элемента $A[j-1]$ выполняется условие 2, а все остальные элементы остаются неизменными. На рис. 7.3*б* показано, что происходит, если $A[j] \leqslant x$; в этом случае увеличивается значение i, элементы $A[i]$ и $A[j]$ меняются местами, после чего на единицу увеличивается значение j. В результате перестановки $A[i] \leqslant x$, и условие 1 выполняется. Аналогично получаем $A[j-1] > x$, поскольку элемент, который был переставлен с элементом $A[j-1]$, согласно инварианту цикла, больше x.

Завершение. По завершении работы алгоритма $j = r$. Поэтому каждый элемент массива является членом одного из трех множеств, описанных в инварианте цикла. Таким образом, все элементы массива разбиты на три множества: величина которых не превышает x, превышающие x, и одноэлементное множество, состоящее из элемента x.

В последних двух строках процедуры PARTITION опорный элемент перемещается на свое место в средину массива при помощи его перестановки с крайним левым элементом, превышающим величину x. Теперь вывод процедуры PARTITION удовлетворяет требованиям, накладываемым на этап разбиения.

Время обработки процедурой PARTITION подмассива $A[p..r]$ равно $\Theta(n)$, где $n = r - p + 1$ (см. упражнение 7.3-1).

Упражнения

7.1-1. Используя в качестве модели рис. 7.1, проиллюстрируйте действие процедуры PARTITION на массив $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

7.1-2. Какое значение q возвращает процедура PARTITION, если все элементы массива $A[p..r]$ одинаковы? Модифицируйте эту процедуру так, чтобы в случае, когда все элементы массива $A[p..r]$ одинаковы, q определялось следующим образом: $q = \lfloor (p + r)/2 \rfloor$.

7.1-3. Приведите краткое обоснование утверждения, что время обработки процедурой PARTITION подмассива, состоящего из n элементов, равно $\Theta(n)$.

7.1-4. Как бы вы изменили процедуру QUICKSORT для сортировки в невозрастающем порядке?

7.2 Производительность быстрой сортировки

Время работы алгоритма быстрой сортировки зависит от степени сбалансированности, которой характеризуется разбиение. Сбалансированность, в свою очередь, зависит от того, какой элемент выбран в качестве опорного. Если разбиение сбалансированное, асимптотически алгоритм работает так же быстро, как и сортировка слиянием. В противном случае асимптотическое поведение этого алгоритма столь же медленное, как и у сортировки вставкой. В данном разделе будет проведено неформальное исследование поведения быстрой сортировки при условии сбалансированного и несбалансированного разбиения.

Наихудшее разбиение

Наихудшее поведение алгоритма быстрой сортировки имеет место в том случае, когда подпрограмма, выполняющая разбиение, порождает одну подзадачу с $n - 1$ элементов, а вторую — с 0 элементов. (Это будет доказано в разделе 7.4.1.) Предположим, что такое несбалансированное разбиение возникает при каждом рекурсивном вызове. Для выполнения разбиения требуется время $\Theta(n)$. Поскольку рекурсивный вызов процедуры разбиения, на вход которой подается массив размера 0, приводит к возврату из этой процедуры без выполнения каких-либо операций, $T(0) = \Theta(1)$. Итак, рекуррентное соотношение, описывающее время работы этой процедуры, записывается следующим образом:

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n).$$

Интуитивно понятно, что при суммировании промежутков времени, затрачиваемых на каждый уровень рекурсии, получается арифметическая прогрессия (уравнение (A.2)), что дает нам в результате $\Theta\left(n^2\right)$. В самом деле, с помощью метода

подстановок легко доказать, что $T(n) = \Theta\left(n^2\right)$ является решением рекуррентного соотношения $T(n) = T(n-1) + \Theta(n)$ (см. упражнение 7.2-1).

Таким образом, если на каждом уровне рекурсии алгоритма разбиение максимально несбалансированное, то время работы алгоритма равно $\Theta\left(n^2\right)$. Следовательно, производительность быстрой сортировки в наихудшем случае не превышает производительности сортировки вставкой. Более того, за такое же время алгоритм быстрой сортировки обрабатывает массив, который уже полностью отсортирован, — часто встречающаяся ситуация, в которой время работы алгоритма сортировки вставкой равно $O(n)$.

Наилучшее разбиение

В самом благоприятном случае процедура PARTITION делит задачу размером n на две подзадачи, размер каждой из которых не превышает $n/2$, поскольку размер одной из них равен $\lfloor n/2 \rfloor$, а второй — $\lceil n/2 \rceil - 1$. В такой ситуации быстрая сортировка работает намного производительнее, и время ее работы описывается следующим рекуррентным соотношением:

$$T(n) \leqslant 2T(n/2) + \Theta(n).$$

Это рекуррентное соотношение подпадает под случай 2 основной теоремы (теоремы 4.1), так что его решение — $T(n) = O(n \lg n)$. Таким образом, разбиение на равные части приводит к асимптотически более быстрому алгоритму.

Сбалансированное разбиение

Как станет ясно из анализа, проведенного в разделе 7.4, в асимптотическом пределе поведение алгоритма быстрой сортировки в среднем случае намного ближе к его поведению в наилучшем случае, чем в наихудшем. Чтобы стало ясно, почему это так, нужно понять, как баланс разбиения отражается на рекуррентном соотношении, описывающем время работы алгоритма.

Предположим, что разбиение происходит в соотношении один к девяти, что на первый взгляд весьма далеко от сбалансированности. В этом случае для времени работы алгоритма быстрой сортировки мы получим следующее рекуррентное соотношение:

$$T(n) \leqslant T(9n/10) + T(n/10) + cn,$$

в которое явным образом входит константа c, скрытая в члене $\Theta(n)$. На рис. 7.4 показано рекурсивное дерево, отвечающее этому рекуррентному соотношению. В узлах дерева приведены размеры соответствующих подзадач, а справа от каждого уровня — его время работы. Время работы уровней явным образом содержит константу c, скрытую в члене $\Theta(n)$. Обратите внимание, что время работы

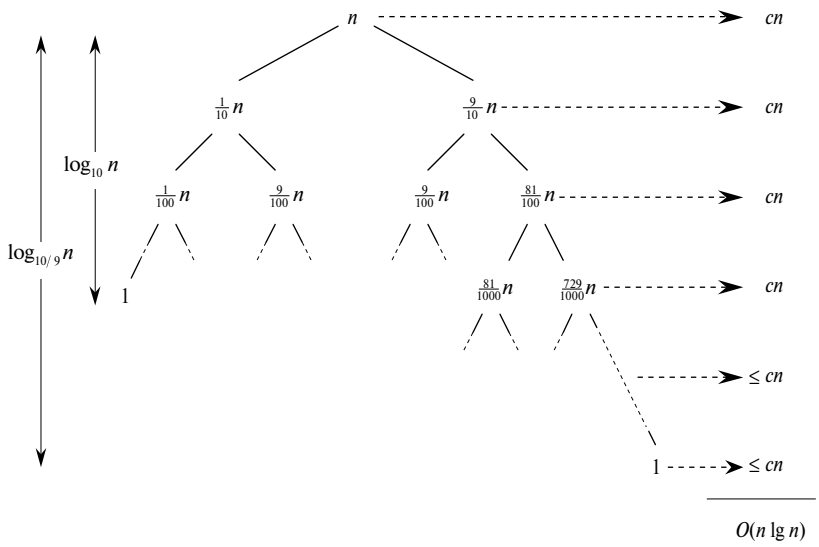


Рис. 7.4. Рекурсивное дерево, соответствующее делению задачи процедурой PARTITION в соотношении 1:9

каждого уровня этого дерева равна cn. Так происходит до тех пор, пока не будет достигнута глубина $\log_{10} n = \Theta(\lg n)$. Время работы более глубоких уровней не превышает величину cn. Рекурсия прекращается на глубине $\log_{10/9} n = \Theta(\lg n)$; таким образом, полное время работы алгоритма быстрой сортировки равно $O(n \lg n)$. Итак, если на каждом уровне рекурсии разбиение производится в соотношении один к девяти, время работы алгоритма быстрой сортировки равно $O(n \lg n)$. Другими словами, несмотря на то, что такое разбиение выглядит довольно несбалансированным, в асимптотическом пределе алгоритм ведет себя так же, как и при делении задачи на две одинаковые подзадачи. Фактически, даже разбиение в соотношении девяносто девять к одному приводит к тому, что время выполнения этого алгоритма будет равно $O(n \lg n)$. Причина в том, что любое разбиение, характеризующееся конечной *константой* пропорциональности, приводит к образованию рекурсивного дерева высотой $\Theta(\lg n)$ и временем работы каждого уровня $O(n)$. Таким образом, при любой постоянной величине пропорции полное время выполнения составляет $O(n \lg n)$.

Интуитивные рассуждения для среднего случая

Чтобы получить представление о работе алгоритма быстрой сортировки в среднем случае, необходимо сделать предположение о том, как часто ожидается появление тех или иных входных данных. Поведение рассматриваемого алгоритма определяется относительным расположением элементов данного входного массива,

а не их конкретной величиной. Как и при вероятностном анализе задачи о найме сотрудника, проведенном в разделе 5.2, пока что предположим, что все перестановки входных чисел равновероятны.

Если алгоритм быстрой сортировки обрабатывает случайным образом выбранный входной массив, то маловероятно, чтобы разбиение на каждом уровне происходило в одном и том же соотношении, как это было в ходе проведенного выше неформального анализа. Предположим, что некоторые разбиения сбалансированы довольно хорошо, в то время как другие сбалансированы плохо. Например, в упражнении 7.2-6 нужно показать, что соотношение размеров подзадач на выходе процедуры PARTITION примерно в 80% случаев сбалансировано лучше, чем девять к одному, а примерно в 20% случаев — хуже.

В среднем случае процедура PARTITION производит и "хорошие" и "плохие" деления. В рекурсивном дереве, соответствующем среднему случаю, хорошие и плохие разбиения равномерно распределены по всему дереву. Для упрощения интуитивных рассуждений предположим, что уровни дерева с плохими и хорошими разбиениями чередуются, и что хорошие разбиения получаются такими, как в наилучшем случае, а плохие — такими, как в наихудшем. На рис. 7.5*а* показаны разбиения на двух соседних уровнях такого рекурсивного дерева. В корне дерева время разбиения пропорционально n, а размеры полученных подмассивов равны $n-1$ и 0, что соответствует наихудшему случаю. На следующем уровне подмассив размера $n-1$ делится оптимальным образом на подмассивы размеров $(n-1)/2-1$ и $(n-1)/2$. Предположим, что для подмассива размером 0 время работы равно 1.

Чередование таких хороших и плохих разбиений приводит к образованию трех подмассивов с размерами 0, $(n-1)/2-1$ и $(n-1)/2$. При этом суммарное время разбиения равно $\Theta(n)+\Theta(n-1)=\Theta(n)$. Эта ситуация определенно не хуже показанной на рис. 7.5*б*, где изображен один уровень разбиения, в результате которого в течение времени $\Theta(n)$ образуются два подмассива с размерами $(n-1)/2$. Однако в этом случае разбиение получается сбалансированным! Интуитивно понятно, что время $\Theta(n-1)$, которое требуется на плохое разбиение, поглощается временем $\Theta(n)$, которое требуется на хорошее разбиение, а полученное в результате разбиение оказывается хорошим. Таким образом, время работы алгоритма быстрой сортировки, при которой на последовательных уровнях чередуются плохие и хорошие разбиения, ведет себя подобно времени работы алгоритма быстрой сортировки, при которой выполняются только хорошие разбиения. Оно также равно $O(n\lg n)$, просто константа, скрытая в O-обозначении, в этом случае несколько больше. Строгий анализ рандомизированной версии алгоритма быстрой сортировки в среднем случае содержится в разделе 7.4.2.

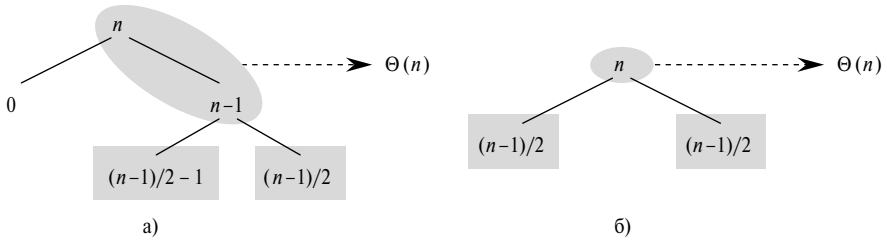


Рис. 7.5. а) Два уровня рекурсивного дерева быстрой сортировки. б) Хорошо сбалансированный уровень рекурсивного дерева. В обоих случаях время выполнения подзадач в заштрихованных эллипсах равно $\Theta(n)$

Упражнения

7.2-1. С помощью метода подстановки докажите, что решение рекуррентного соотношения $T(n) = T(n-1) + \Theta(n)$ имеет вид $T(n) = \Theta\left(n^2\right)$, как утверждалось в начале раздела 7.2.

7.2-2. Чему равно время работы процедуры QUICKSORT в случае, когда все элементы массива A одинаковы по величине?

7.2-3. Покажите, что если все элементы массива A различаются по величине и расположены в убывающем порядке, то время работы процедуры QUICKSORT равно $\Theta\left(n^2\right)$.

7.2-4. Сведения о банковских операциях часто записываются в хронологическом порядке, но многие предпочитают получать отчеты о состоянии своих банковских счетов в порядке нумерации чеков. Чеки чаще всего выписываются в порядке возрастания их номеров, а торговцы обычно обналичивают их с небольшой задержкой. Таким образом, задача преобразования упорядочения по времени операций в упорядочение по номерам чеков — это задача сортировки почти упорядоченных массивов. Обоснуйте утверждение, что при решении этой задачи процедура INSERTION_SORT превзойдет по производительности процедуру QUICKSORT.

7.2-5. Предположим, что в ходе быстрой сортировки на каждом уровне происходит разбиение в пропорции $1-\alpha$ к α, где $0 < \alpha \leqslant 1/2$ — константа. Покажите, что минимальная глубина, на которой расположен лист рекурсивного дерева, приблизительно равна $-\lg n/\lg a$, а максимальная глубина — приблизительно $-\lg n/\lg(1-\alpha)$. (Округление до целых чисел во внимание не принимается.)

⋆ 7.2-6. Докажите, что для любой константы $0 < \alpha \leqslant 1/2$ вероятность того, что процедура PARTITION поделит случайно выбранный входной массив в более сбалансированной пропорции, чем $1-\alpha$ к α, приблизительно равна $1-2\alpha$.

7.3 Рандомизированная версия быстрой сортировки

Исследуя поведение алгоритма быстрой сортировки в среднем случае, мы сделали предположение, что все перестановки входных чисел встречаются с равной вероятностью. Однако на практике это далеко не всегда так (см. упражнение 7.2-4). Иногда путем добавления в алгоритм этапа рандомизации (аналогично тому, как это было сделано в разделе 5.3) удается получить среднюю производительность во всех случаях. Многие считают такую рандомизированную версию алгоритма быстрой сортировки оптимальным выбором для обработки достаточно больших массивов.

В разделе 5.3 рандомизация алгоритма проводилась путем явной перестановки его входных элементов. В алгоритме быстрой сортировки можно было бы поступить точно так же, однако анализ упростится, если применить другой метод рандомизации, получивший название ***случайной выборки*** (random sampling). Вместо того чтобы в качестве опорного элемента всегда использовать $A[r]$, такой элемент будет выбираться в массиве $A[p..r]$ случайным образом. Подобная модификация, при которой опорный элемент выбирается случайным образом среди элементов с индексами от p до r, обеспечивает равную вероятность оказаться опорным любому из $r-p+1$ элементов подмассива. Благодаря случайному выбору опорного элемента можно ожидать, что разбиение входного массива в среднем окажется довольно хорошо сбалансированным.

Изменения, которые нужно внести в процедуры PARTITION и QUICKSORT, незначительны. В новой версии процедуры PARTITION непосредственно перед разбиением достаточно реализовать перестановку:

```
RANDOMIZED_PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  Обменять A[r] ↔ A[i]
3  return PARTITION(A, p, r)
```

В новой процедуре быстрой сортировки вместо процедуры PARTITION вызывается процедура RANDOMIZED_PARTITION:

```
RANDOMIZED_QUICKSORT(A, p, r)
1  if p < r
2     then q ← RANDOMIZED_PARTITION(A, p, r)
3          RANDOMIZED_QUICKSORT(A, p, q − 1)
4          RANDOMIZED_QUICKSORT(A, q + 1, r)
```

Этот алгоритм будет проанализирован в следующем разделе.

Упражнения

7.3-1. Почему производительность рандомизированного алгоритма анализируется в среднем, а не в наихудшем случае?

7.3-2. Сколько раз в ходе выполнения процедуры RANDOMIZED_QUICKSORT в наихудшем случае вызывается генератор случайных чисел RANDOM?

7.4 Анализ быстрой сортировки

В разделе 7.2 были приведены некоторые интуитивные рассуждения по поводу поведения алгоритма быстрой сортировки в наихудшем случае, и обосновывалось, почему следует ожидать достаточно высокой производительности его работы. В данном разделе проведен более строгий анализ поведения этого алгоритма. Начнем этот анализ с наихудшего случая. Подобный анализ применим как к процедуре QUICKSORT, так и к процедуре RANDOMIZED_QUICKSORT. В конце раздела анализируется работа процедуры RANDOMIZED_QUICKSORT в среднем случае.

7.4.1 Анализ в наихудшем случае

В разделе 7.2 было показано, что при самом неудачном разбиении на каждом уровне рекурсии время работы алгоритма быстрой сортировки равно $\Theta\left(n^2\right)$. Интуитивно понятно, что это наихудшее время работы рассматриваемого алгоритма. Докажем это утверждение.

С помощью метода подстановки (см. раздел 4.1) можно показать, что время работы алгоритма быстрой сортировки равно $O\left(n^2\right)$. Пусть $T(n)$ — наихудшее время обработки процедурой QUICKSORT входных данных размером n. Тогда мы получаем следующее рекуррентное соотношение:

$$T(n) = \max_{0 \leqslant q \leqslant n-1} \left(T(q) + T(n-q-1)\right) + \Theta(n), \tag{7.1}$$

где параметр q изменяется в интервале от 0 до $n-1$, поскольку на выходе процедуры PARTITION мы получаем две подзадачи, полный размер которых равен $n-1$. Мы предполагаем, что $T(n) \leqslant cn^2$ для некоторой константы c. Подставляя это неравенство в рекуррентное соотношение (7.1), получим

$$\begin{aligned} T(n) &\leqslant \max_{0 \leqslant q \leqslant n-1} \left(cq^2 + c(n-q-1)^2\right) + \Theta(n) = \\ &= c \cdot \max_{0 \leqslant q \leqslant n-1} \left(q^2 + (n-1-q)^2\right) + \Theta(n). \end{aligned}$$

Выражение $q^2 + (n-q-1)^2$ достигает максимума на обоих концах интервала $0 \leqslant q \leqslant n-1$, что подтверждается тем, что вторая производная от него по

q положительна (см. упражнение 7.4-3). Это наблюдение позволяет нам сделать следующую оценку:

$$\max_{0 \leqslant q \leqslant n-1} \left(q^2 + (n-q-1)^2 \right) \leqslant (n-1)^2 = n^2 - 2n + 1.$$

В результате получаем следующее ограничение для $T(n)$:

$$T(n) \leqslant cn^2 - c(2n-1) + \Theta(n) \leqslant cn^2,$$

поскольку константу c можно выбрать настолько большой, чтобы слагаемое $c(2n-1)$ доминировало над слагаемым $\Theta(n)$. Таким образом, $T(n) = O\left(n^2\right)$. В разделе 7.2 мы встречались с частным случаем быстрой сортировки, при котором требовалось время $\Omega\left(n^2\right)$ — это случай несбалансированного разбиения. В упражнении 7.4-1 нужно показать, что рекуррентное соотношение (7.1) имеет решение $T(n) = \Omega\left(n^2\right)$. Таким образом, время работы алгоритма быстрой сортировки (в наихудшем случае) равно $\Theta\left(n^2\right)$.

7.4.2 Математическое ожидание времени работы

Мы уже приводили интуитивный аргумент в пользу того, что время работы процедуры RANDOMIZED_QUICKSORT в среднем случае равно $O(n \lg n)$: если на каждом уровне рекурсии в разделении, производимом процедурой RANDOMIZED_PARTITION, в одну часть массива помещается произвольная фиксированная доля элементов, то высота рекурсивного дерева равна $\Theta(\lg n)$, а время работы каждого его уровня — $O(n)$. Даже после добавления новых уровней с наименее сбалансированным разбиением время работы останется равным $O(n \lg n)$. Можно провести точный анализ математического ожидания времени работы процедуры RANDOMIZED_QUICKSORT. Для этого сначала нужно понять, как работает процедура разбиения, а затем — получить для математического ожидания времени работы оценку $O(n \lg n)$ в предположении, что значения всех элементов различны. Эта верхняя граница математического ожидания времени работы в сочетании с полученной в разделе 7.2 оценкой для наилучшего случая, равной $\Theta(n \lg n)$, позволяют сделать вывод о том, что математическое ожидание времени работы равно $\Theta(n \lg n)$.

Время работы и сравнения

Время работы процедуры QUICKSORT определяется преимущественно временем работы, затраченным на выполнение процедуры PARTITION. При каждом выполнении последней происходит выбор опорного элемента, который впоследствии не принимает участия ни в одном рекурсивном вызове процедур QUICKSORT и PARTITION. Таким образом, на протяжении всего времени выполнения алгоритма

быстрой сортировки процедура PARTITION вызывается не более n раз. Один вызов процедуры PARTITION выполняется в течение времени $O(1)$, к которому нужно прибавить время, пропорциональное количеству итераций цикла **for** в строках 3–6. В каждой итерации цикла **for** в строке 4 опорный элемент сравнивается с другими элементами массива A. Поэтому, если известно количество выполнений строки 4, можно оценить полное время, которое затрачивается на выполнение цикла **for** в процессе работы процедуры QUICKSORT.

Лемма 7.1. Пусть X — количество сравнений, которое выполняется в строке 4 процедуры PARTITION в течение полной обработки n-элементного массива процедурой QUICKSORT. Тогда время работы процедуры QUICKSORT равно $O(n + X)$.

Доказательство. Как следует из приведенных выше рассуждений, процедура PARTITION вызывается n раз, и при этом производится определенный фиксированный объем работы. Затем определенное число раз запускается цикл **for**, при каждой итерации которого выполняется строка 4. ■

Таким образом, наша цель — вычислить величину X, т.е. полное количество сравнений, выполняемых при всех вызовах процедуры PARTITION. Не будем пытаться проанализировать, сколько сравнений производится при *каждом* вызове этой процедуры. Вместо этого получим общую оценку полного количества сравнений. Для этого необходимо понять, в каких случаях в алгоритме производится сравнение двух элементов массива, а в каких — нет. Для упрощения анализа переименуем элементы массива A как $z_1, z_2, \ldots, z_n$, где z_i — i-й наименьший по порядку элемент. Кроме того, определим множество $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$, которое содержит элементы, расположенные между элементами z_i и z_j включительно.

В каких случаях в алгоритме производится сравнение элементов z_i и z_j? Прежде чем ответить на этот вопрос, заметим, что сравнение каждой пары элементов производится не более одного раза. Почему? Дело в том, что элементы сравниваются с опорным, который никогда не используется в двух разных вызовах процедуры PARTITION. Таким образом, после определенного вызова этой процедуры используемый в качестве опорного элемент впоследствии больше не будет сравниваться с другими элементами.

Воспользуемся в нашем анализе индикаторными случайными величинами (см. раздел 5.2). Определим величину

$$X_{ij} = \mathrm{I}\{z_i \text{ сравнивается с } z_j\},$$

с помощью которой учитывается, произошло ли сравнение в течение работы алгоритма (но не в течение определенной итерации или определенного вызова процедуры PARTITION). Поскольку каждая пара элементов сравнивается не более одного

раза, полное количество сравнений, выполняемых на протяжении работы алгоритма, легко выразить следующим образом:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

Применяя к обеим частям этого выражения операцию вычисления математического ожидания и используя свойство ее линейности и лемму 5.1, получим:

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] = \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ сравнивается с } z_j\}. \end{aligned} \tag{7.2}$$

Осталось вычислить величину $\Pr\{z_i$ сравнивается с $z_j\}$. Предполагается, что опорные элементы выбираются случайным образом, причем независимо друг от друга.

Полезно поразмышлять о том, когда два элемента *не* сравниваются. Рассмотрим в качестве входных данных для алгоритма быстрой сортировки множество, состоящее из целых чисел от 1 до 10 (в произвольном порядке), и предположим, что в качестве первого опорного элемента выбрано число 7. Тогда в результате первого вызова процедуры PARTITION все числа разбиваются на два множества: $\{1, 2, 3, 4, 5, 6\}$ и $\{8, 9, 10\}$. При этом элемент 7 сравнивается со всеми остальными. Очевидно, что ни одно из чисел, попавшее в первое подмножество (например, 2), больше не будет сравниваться ни с одним элементом второго подмножества (например, с 9).

В общем случае ситуация такая. Поскольку предполагается, что значения всех элементов различаются, то при выборе x в качестве опорного элемента впоследствии не будут сравниваться никакие z_i и z_j, для которых $z_i < x < z_j$. С другой стороны, если в качестве опорного выбран элемент z_i, то он будет сравниваться с каждым элементом множества Z_{ij}, кроме себя самого. Аналогичное утверждение можно сделать по поводу элемента z_j. В рассматриваемом примере значения 7 и 9 сравниваются, поскольку элемент 7 — первый из множества $Z_{7,9}$, выбранный в роли опорного. По той же причине (поскольку элемент 7 — первый из множества $Z_{2,9}$, выбранный в роли опорного) элементы 2 и 9 сравниваться не будут. Таким образом, элементы z_i и z_j сравниваются тогда и только тогда, когда первым в роли опорного в множестве Z_{ij} выбран один из них.

Теперь вычислим вероятность этого события. Перед тем как в множестве Z_{ij} будет выбран опорный элемент, все это множество является не разделенным, и любой его элемент с одинаковой вероятностью может стать опорным. Поскольку всего в этом множестве $j-i+1$ элемент, а опорные элементы выбираются случайным образом и независимо друг от друга, вероятность того, что какой-либо фиксированный элемент первым будет выбран в качестве опорного, равна $1/(j-i+1)$. Таким образом, выполняется следующее соотношение:

$$\begin{aligned}\Pr\{z_i \text{ сравнивается с } z_j\} &= \\ &= \Pr\{\text{Первым опорным элементом выбран } z_i \text{ или } z_j\} = \\ &= \Pr\{\text{Первым опорным элементом выбран } z_i\} + \\ &\quad + \Pr\{\text{Первым опорным элементом выбран } z_j\} = \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}.\end{aligned} \tag{7.3}$$

Сумма вероятностей в приведенной выше цепочке равенств следует из того, что рассматриваемые события взаимоисключающие. Сопоставляя уравнения (7.2) и (7.3), получаем:

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}.$$

Эту сумму можно оценить, воспользовавшись заменой переменных ($k = j-i$) и границей для гармонического ряда (уравнение (A.7)):

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \\ &< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n).\end{aligned} \tag{7.4}$$

Таким образом, можно сделать вывод, что при использовании процедуры RANDOMIZED_PARTITION математическое ожидание времени работы алгоритма быстрой сортировки различающихся по величине элементов равно $O(n \lg n)$.

Упражнения

7.4-1. Покажите, что в рекуррентном соотношении

$$\begin{aligned}T(n) &= \max_{0 \leqslant q \leqslant n-1} (T(q) + T(n-q-1)) + \Theta(n), \\ T(n) &= \Omega\left(n^2\right).\end{aligned}$$

7.4-2. Покажите, что в самом благоприятном случае время работы алгоритма быстрой сортировки равно $\Omega(n \lg n)$.

7.4-3. Покажите, что функция $q^2 + (n - q - 1)^2$ на множестве $q = 0, 1, \ldots, n-1$ достигает максимума при $q = 0$ или $q = n - 1$.

7.4-4. Покажите, что математическое ожидание времени работы процедуры RANDOMIZED_QUICKSORT равно $\Omega(n \lg n)$.

7.4-5. На практике время работы алгоритма быстрой сортировки можно улучшить, воспользовавшись тем, что алгоритм сортировки по методу вставок быстро работает для "почти" отсортированных входных последовательностей. Для этого можно поступить следующим образом. Когда процедура быстрой сортировки начнет рекурсивно вызываться для обработки подмассивов, содержащих менее k элементов, в ней не будет производиться никаких действий, кроме выхода из процедуры. После возврата из процедуры быстрой сортировки, вызванной на самом высоком уровне, запускается алгоритм сортировки по методу вставок, на вход которого подается весь обрабатываемый массив. На этом процесс сортировки завершается. Докажите, что математическое ожидание времени работы такого алгоритма сортировки равно $O(nk + n \lg(n/k))$. Как следует выбирать значение k, исходя из теоретических и практических соображений?

⋆ 7.4-6. Рассмотрим модификацию процедуры PARTITION путем случайного выбора трех элементов массива A и разбиения массива по медиане выбранных элементов (т.е. по среднему из этих трех элементов). Найдите приближенную величину вероятности того, что в худшем случае разбиение будет произведено в отношении α к $(1 - \alpha)$, как функцию от α в диапазоне $0 < \alpha < 1$.

Задачи

7-1. **Корректность разбиения по Хоару**

Представленная в этой главе версия процедуры PARTITION не является реализацией первоначально предложенного алгоритма. Ниже приведен исходный алгоритм разбиения, разработанный Хоаром (C.A.R. Hoare):

```
HOARE_PARTITION(A, p, r)
 1  x ← A[p]
 2  i ← p − 1
 3  j ← r + 1
 4  while TRUE
 5       do repeat j ← j − 1
```

```
6            until A[j] ⩽ x
7          repeat i ← i + 1
8            until A[i] ⩾ x
9          if i < j
10           then Обменять A[i] ↔ A[j]
11           else return j
```

а) Продемонстрируйте, как работает процедура HOARE_PARTITION с входным массивом $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$. Для этого укажите, чему будут равны значения элементов массива и значения вспомогательных переменных после каждой итерации цикла **while** в строках 4–11.

В следующих трех заданиях предлагается привести аргументы в пользу корректности процедуры HOARE_PARTITION. Докажите следующие утверждения.

б) Индексы i и j принимают такие значения, что обращение к элементам массива A, находящимся за пределами подмассива $A[p..r]$, никогда не произойдет.

в) По завершении процедуры HOARE_PARTITION возвращается значение j, такое что $p \leqslant j < r$.

г) По завершении процедуры HOARE_PARTITION каждый элемент подмассива $A[p..j]$ не превышает значений каждого элемента подмассива $A[j+1..r]$.

В описанной в разделе 7.1 процедуре PARTITION опорный элемент (которым изначально является элемент $A[r]$) отделяется от двух образованных с его помощью подмассивов. В процедуре же HOARE_PARTITION, напротив, опорный элемент (которым изначально является элемент $A[p]$) всегда помещается в один из двух полученных подмассивов: $A[p..j]$ или $A[j+1..r]$. Поскольку $p \leqslant j < r$, это разбиение всегда нетривиально.

д) Перепишите процедуру QUICKSORT таким образом, чтобы в ней использовалась процедура HOARE_PARTITION.

7-2. Альтернативный анализ быстрой сортировки

Можно предложить альтернативный анализ рандомизированной быстрой сортировки, в ходе которого внимание сосредотачивается на математическом ожидании времени, которое требуется для выполнения каждого рекурсивного вызова процедуры QUICKSORT, а не на количестве производимых сравнений.

а) Докажите, что вероятность того, что какой-либо заданный элемент n-элементного массива будет выбран в качестве опорного, равна $1/n$. Опираясь на этот факт, определите индикаторную случайную величину

$$X_i = \mathrm{I}\{i\text{-й в порядке возрастания элемент выбран опорным}\}.$$

Какой смысл имеет величина $\mathrm{E}[X_i]$?

б) Пусть $T(n)$ — случайная величина, обозначающая время работы алгоритма быстрой сортировки n-элементного массива. Докажите, что

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q \left(T(q-1) + T(n-q) + \Theta(n)\right)\right]. \qquad (7.5)$$

в) Покажите, что уравнение (7.5) можно представить в виде

$$\mathrm{E}[T(n)] = \frac{2}{n}\sum_{q=2}^{n-1} \mathrm{E}[T(q)] + \Theta(n). \qquad (7.6)$$

г) Покажите, что

$$\sum_{k=2}^{n-1} k \lg k \leqslant \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2. \qquad (7.7)$$

(*Указание*: разбейте сумму на две части, в одной из которых суммирование проводится по индексам $k = 2, 3, \ldots, \lceil n/2 \rceil - 1$, а в другой — по индексам $k = \lceil n/2 \rceil, \ldots, n-1$.)

д) Покажите с помощью неравенства (7.7), что решение рекуррентного соотношения (7.6) имеет вид $\mathrm{E}[T(n)] = \Theta(n \lg n)$. (*Указание*: с помощью подстановки покажите, что $\mathrm{E}[T(n)] \leqslant an \lg n$ для достаточно больших n и некоторой положительной константы a.)

7-3. **Блуждающая сортировка**

Два профессора предложили “элегантный” алгоритм сортировки, который выглядит следующим образом:

STOOGE_SORT(A, i, j)

```
1  if A[i] > A[j]
2      then Обменять A[i] ↔ A[j]
3  if i + 1 ≥ j
4      then return
5  k ← ⌊(j − i + 1)/3⌋          ▷ Округление в меньшую сторону
6  STOOGE_SORT(A, i, j − k)     ▷ Первые две трети
7  STOOGE_SORT(A, i + k, j)     ▷ Последние две трети
8  STOOGE_SORT(A, i, j − k)     ▷ Снова первые две трети
```

а) Докажите, что если $n = length\,[A]$, то процедура STOOGE_SORT$(A, 1, length\,[A])$ корректно сортирует входной массив $A\,[1..n]$.

б) Запишите рекуррентное соотношение для времени работы процедуры STOOGE_SORT и дайте точную асимптотическую границу (в Θ-обозначениях) этой величины.

в) Сравните время работы процедуры STOOGE_SORT в наихудшем случае со временем сортировки вставкой, сортировки слиянием, пирамидальной сортировки и быстрой сортировки. Достойны ли эти профессора своих званий?

7-4. **Глубина стека для быстрой сортировки**

В описанном в разделе 7.1 алгоритме QUICKSORT содержатся два рекурсивных вызова этого же алгоритма. После вызова процедуры PARTITION полученные в результате правый и левый подмассивы рекурсивно сортируются. Второй рекурсивный вызов процедуры QUICKSORT на самом деле не является необходимым; его можно избежать с помощью итеративной управляющей структуры. Этот метод, получивший название ***оконечной рекурсии*** (tail recursion), автоматически обеспечивается хорошими компиляторами. Рассмотрите приведенную ниже версию быстрой сортировки, в которой имитируется оконечная рекурсия:

```
QUICKSORT′(A, p, r)
1  while p < r
2        do   ▷ Разбиение и сортировка левого подмассива
3           q ← PARTITION(A, p, r)
4           QUICKSORT′(A, p, q − 1)
5           p ← q + 1
```

а) Докажите, что процедура QUICKSORT$'(A, 1, length\,[A])$ корректно сортирует входной массив A.

Обычно компиляторы выполняют рекурсивные процедуры с помощью ***стека***, содержащего необходимую информацию, в том числе и значения параметров, применяющихся для каждого рекурсивного вызова. Информация о самом последнем вызове находится на верху стека, а информация о начальном вызове — на его дне. При вызове процедуры информация ***записывается в стек*** (pushed), а при ее завершении — ***выталкивается*** (popped) из него. Поскольку предполагается, что массивы-параметры представлены указателями, при каждом обращении процедуры к стеку передается информация объемом $O\,(1)$. ***Глубина стека*** (stack depth) — это максимальная величина стекового пространства, которая используется в ходе вычисления.

б) Опишите сценарий, в котором глубина стека, необходимая для обработки процедурой QUICKSORT′ n-элементного массива, равна $\Theta(n)$.

в) Модифицируйте код процедуры QUICKSORT′ так, чтобы необходимая для ее работы глубина стека в наихудшем случае была равна $\Theta(\lg n)$. Математическое ожидание времени работы алгоритма $O(n \lg n)$ должно при этом остаться неизменным.

7-5. **Разбиение по медиане трех элементов**

Один из способов улучшения процедуры RANDOMIZED_QUICKSORT заключается в том, чтобы производить разбиение по опорному элементу, определенному аккуратнее, чем путем случайного выбора. Один из распространенных подходов — метод ***медианы трех элементов*** (median-of-3). Согласно этому методу, в качестве опорного элемента выбирается медиана (средний элемент) подмножества, составленного из трех выбранных в подмассиве элементов (см. упражнение 7.4-6). В этой задаче предполагается, что все элементы входного подмассива $A[1..n]$ различны и что $n \geqslant 3$. Обозначим сохраняемый выходной массив через $A'[1..n]$. Используя опорный элемент x с помощью метода медианы трех элементов, определим $p_i = \Pr\{x = A'[i]\}$.

а) Приведите точную формулу для величины p_i как функции от n и i для $i = 2, 3, \ldots, n-1$. (Заметим, что $p_1 = p_n = 0$.)

б) На сколько увеличится вероятность выбора в массиве $A[1..n]$ в качестве опорного элемента $x = A'[\lfloor (n+1)/2 \rfloor]$, если используется не обычный способ, а метод медианы? Чему равны предельные значения этих вероятностей при $n \to \infty$?

в) Будем считать выбор опорного элемента $x = A'[i]$ “удачным”, если $n/3 \leqslant i \leqslant 2n/3$. На сколько возрастает вероятность удачного выбора в методе медианы по сравнению с обычной реализацией? (*Указание*: используйте приближение суммы интегралом.)

г) Докажите, что при использовании метода медианы трех элементов время работы алгоритма быстрой сортировки, равное $\Omega(n \lg n)$, изменится только на постоянный множитель.

7-6. **Нечеткая сортировка по интервалам**

Рассмотрим задачу сортировки, в которой отсутствуют точные сведения о значениях чисел. Вместо них для каждого числа на действительной числовой оси задается интервал, которому оно принадлежит. Таким образом, в нашем распоряжении имеется n закрытых интервалов, заданных в виде $[a_i, b_i]$, где $a_i \leqslant b_i$. Задача в том, чтобы произвести ***нечеткую сортировку*** (fuzzy-sort) этих интервалов, т.е. получить такую

перестановку $(i_1, i_2, \ldots, i_n)$ интервалов, чтобы в каждом интервале можно было найти значения $c_j \in \left[a_{i_j}, b_{i_j}\right]$, удовлетворяющие неравенствам $c_1 \leqslant c_2 \leqslant \cdots \leqslant c_n$.

а) Разработайте алгоритм нечеткой сортировки n интервалов. Общая структура предложенного алгоритма должна совпадать со структурой алгоритма, выполняющего быструю сортировку по левым границам интервалов (т.е. по значениям a_i). Однако время работы нового алгоритма должно быть улучшено за счет возможностей, предоставляемых перекрытием интервалов. (По мере увеличения степени перекрытия интервалов, задача их нечеткой сортировки все больше упрощается. В представленном алгоритме следует воспользоваться этим преимуществом, насколько это возможно.)

б) Докажите, что в общем случае математическое ожидание времени работы рассматриваемого алгоритма равно $\Theta(n \lg n)$, но если все интервалы перекрываются (т.е. если существует такое значение x, что для всех i $x \in [a_i, b_i]$), то оно равно $\Theta(n)$. В предложенном алгоритме этот факт не следует проверять явным образом; его производительность должна улучшаться естественным образом по мере увеличения степени перекрывания.

Заключительные замечания

Процедуру быстрой сортировки впервые разработал Хоар (Hoare) [147]; предложенная им версия описана в задаче 7-1. Представленная в разделе 7.1 процедура PARTITION появилась благодаря Ломуто (N. Lomuto). Содержащийся в разделе 7.4 анализ выполнен Авримом Блюмом (Avrim Blum). Хороший обзор литературы, посвященной описанию особенностей реализации алгоритмов и их влияния на производительность, можно найти у Седжвика (Sedgewick) [268] и Бентли (Bentley) [40].

Мак-Илрой (McIlroy) [216] показал, как создать конструкцию, позволяющую получить массив, при обработке которого почти каждая реализация быстрой сортировки будет выполняться в течение времени $\Theta\left(n^2\right)$. Если реализация рандомизированная, то данная конструкция может создать данный массив на основании информации о том, каким образом в рандомизированном алгоритме быстрой сортировки осуществляется случайный выбор.

ГЛАВА 8

Сортировка за линейное время

Мы уже познакомились с несколькими алгоритмами, позволяющими выполнить сортировку n чисел за время $O(n \lg n)$. В алгоритмах сортировки по методу слияния и пирамидальной сортировки эта верхняя граница достигалась в наихудшем случае; в алгоритме быстрой сортировки она достигалась в среднем случае. Кроме того, для каждого из этих алгоритмов можно создать такую последовательность из n входных чисел, для которой алгоритм будет работать в течение времени $\Omega(n \lg n)$.

Все упомянутые алгоритмы обладают одним общим свойством: *при сортировке используется только сравнение входных элементов*. Назовем такие алгоритмы сортировки ***сортировкой сравнением*** (comparison sorts). Все описанные до настоящего момента алгоритмы сортировки принадлежат к данному типу.

В разделе 8.1 будет доказано, что при любой сортировке путем сравнения для обработки n элементов в наихудшем случае нужно произвести не менее $\Omega(n \lg n)$ сравнений. Таким образом, алгоритмы сортировки слиянием и пирамидальной сортировки асимптотически оптимальны, и не существует алгоритмов этого класса, которые бы работали быстрее и время выполнения которых отличалось бы больше, чем на постоянный множитель.

В разделах 8.2, 8.3 и 8.4 рассматриваются три алгоритма сортировки: сортировка подсчетом (counting sort), поразрядная сортировка (radix sort) и карманная сортировка (bucket sort). Все эти алгоритмы выполняются в течение времени, линейно зависящего от количества элементов. Вряд ли стоит говорить о том, что в этих алгоритмах для определения правильного порядка элементов применяются операции, отличные от сравнений, а следовательно, нижняя граница $\Omega(n \lg n)$ к ним не применима.

8.1 Нижние оценки алгоритмов сортировки

В алгоритмах сортировки сравнением для получения информации о расположении элементов входной последовательности $\langle a_1, a_2, \ldots, a_n \rangle$ используются только попарные сравнения элементов. Другими словами, для определения взаимного порядка двух элементов a_i и a_j выполняется одна из проверок $a_i < a_j$, $a_i \leqslant a_j$, $a_i = a_j$, $a_i \geqslant a_j$ или $a_i > a_j$. Значения самих элементов или иная информация о них не доступна.

В этом разделе без потери общности предполагается, что все входные элементы различны. При этом операция сравнения $a_i = a_j$ становится бесполезной, и можно предположить, что никаких сравнений этого вида не производится. Заметим также, что операции $a_i < a_j$, $a_i \leqslant a_j$, $a_i = a_j$, $a_i \geqslant a_j$ и $a_i > a_j$ эквивалентны в том смысле, что они дают одну и ту же информацию о взаимном расположении элементов a_i и a_j. Таким образом, можно считать, что все сравнения имеют вид $a_i \leqslant a_j$.

Модель дерева решений

Абстрактное рассмотрение алгоритмов сортировки сравнением можно производить с помощью ***деревьев решений*** (decision trees). Дерево решений — это полное бинарное дерево, в котором представлены операции сравнения элементов, производящиеся тем или иным алгоритмом сортировки, который обрабатывает входные данные заданного размера. Управляющие операции, перемещение данных и все другие аспекты алгоритма игнорируются. На рис. 8.1 показано дерево решений, которое соответствует представленному в разделе 2.1 алгоритму сортировки вставкой, обрабатывающему входную последовательность из трех элементов.

В дереве решений каждый внутренний узел обозначен двухиндексной меткой $i : j$, где индексы i и j принадлежат интервалу $1 \leqslant i, j \leqslant n$, а n — это количество элементов во входной последовательности. Метка указывает на

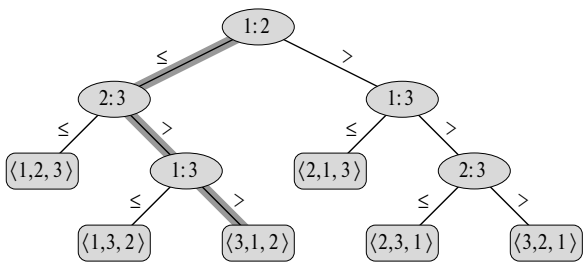


Рис. 8.1. Дерево решений для алгоритма сортировки вставкой, обрабатывающего три элемента

то, что сравниваются элементы a_i и a_j. Каждый лист помечен перестановкой $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$, которая представляет окончательное упорядочение элементов $\langle a_{\pi(1)} \leqslant a_{\pi(2)} \leqslant \cdots \leqslant a_{\pi(n)} \rangle$. (Детальный материал по перестановкам содержится в разделе B.1.) Выполнение алгоритма сортировки соответствует прохождению пути от корня дерева до одного из его листьев. Толстой серой линией на рис. 8.1 выделена последовательность решений, принимаемых алгоритмом при сортировке входной последовательности $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; показанная в листе дерева перестановка $\langle 3, 1, 2 \rangle$ указывает на то, что порядок сортировки определяется соотношениями $a_3 = 5 \leqslant a_1 = 6 \leqslant a_2 = 8$. В данном случае всего имеется $3! = 6$ возможных перестановок входных элементов, поэтому дерево решений должно иметь не менее 6 листьев. В общем случае в каждом внутреннем узле производится сравнение $a_i \leqslant a_j$. Левым поддеревом предписываются дальнейшие сравнения, которые нужно выполнить при $a_i \leqslant a_j$, а правым — сравнения, которые нужно выполнить при $a_i > a_j$. Дойдя до какого-нибудь из листьев, алгоритм сортировки устанавливает соответствующее этому листу упорядочение элементов $\langle a_{\pi(1)} \leqslant a_{\pi(2)} \leqslant \cdots \leqslant a_{\pi(n)} \rangle$. Поскольку каждый корректный алгоритм сортировки должен быть способен произвести любую перестановку входных элементов, необходимое условие корректности сортировки сравнением заключается в том, что в листьях дерева решений должны размещаться все $n!$ перестановок n элементов, и что из корня дерева к каждому его листу можно проложить путь, соответствующий одному из реальных вариантов работы сортировки сравнением. (Назовем такие листы “достижимыми”.) Таким образом, мы будем рассматривать только такие деревья решений, в которых каждая из перестановок представлена в виде достижимого листа.

Нижняя оценка для наихудшего случая

Величина самого длинного пути от корня дерева решений к любому из его достижимых листов соответствует количеству сравнений, которые выполняются в рассматриваемом алгоритме сортировки в наихудшем случае. Следовательно, количество сравнений, выполняемых в том или ином алгоритме сортировки сравнением в наихудшем случае, равно высоте его дерева решений. Поэтому нижняя оценка высот для всех деревьев, в которых все перестановки представлены достижимыми листьями, является нижней оценкой времени работы для любого алгоритма сортировки сравнением. Эту оценку дает приведенная ниже теорема.

Теорема 8.1. В наихудшем случае в ходе выполнения любого алгоритма сортировки сравнением выполняется $\Omega(n \lg n)$ сравнений.

Доказательство. Из приведенных выше рассуждений становится понятно, что для доказательства теоремы достаточно определить высоту дерева, в котором каждая перестановка представлена достижимым листом. Рассмотрим дерево решений

высотой h с l достижимыми листьями, которое соответствует сортировке сравнением n элементов. Поскольку каждая из $n!$ перестановок входных элементов сопоставляется с одним из листьев, $n! \leqslant l$. Так как бинарное дерево высоты h имеет не более 2^h листьев, получаем:

$$n! \leqslant l \leqslant 2^h,$$

откуда после логарифмирования в силу монотонности логарифма и уравнения (3.18) следует:

$$h \geqslant \lg(n!) = \Omega(n \lg n).$$

■

Следствие 8.2. Пирамидальная сортировка и сортировка слиянием — асимптотически оптимальные алгоритмы сортировки.

Доказательство. Верхние границы $O(n \lg n)$ времени работы пирамидальной сортировки и сортировки слиянием, совпадают с нижней границей $\Omega(n \lg n)$ для наихудшего случая из теоремы 8.1. ■

Упражнения

8.1-1. Чему равна наименьшая допустимая глубина, на которой находится лист дерева решений сортировки сравнением?

8.1-2. Получите асимптотически точные границы для величины $\lg(n!)$, не используя приближение Стирлинга. Вместо этого воспользуйтесь для оценки суммы $\sum_{k=1}^{n} \lg k$ методом, описанным в разделе А.2.

8.1-3. Покажите, что не существует алгоритмов сортировки сравнением, время работы которых линейно по крайней мере для половины из $n!$ вариантов входных данных длины n. Что можно сказать по поводу $1/n$-й части всех вариантов входных данных? По поводу $1/2^n$-й части?

8.1-4. Производится сортировка последовательности, состоящей из n элементов. Входная последовательность состоит из n/k подпоследовательностей, в каждой из которых k элементов. Все элементы данной подпоследовательности меньше элементов следующей подпоследовательности и больше элементов предыдущей подпоследовательности. Таким образом, для сортировки всей n-элементной последовательности достаточно отсортировать k элементов в каждой из n/k подпоследовательностей. Покажите, что нижняя граница количества сравнений, необходимых для решения этой разновидности задачи сортировки, равна $\Omega(n \lg k)$. (*Указание*: просто скомбинировать нижние границы для отдельных подпоследовательностей недостаточно.)

8.2 Сортировка подсчетом

В ***сортировке подсчетом*** (counting sort) предполагается, что все n входных элементов — целые числа, принадлежащие интервалу от 0 до k, где k — некоторая целая константа. Если $k = O(n)$, то время работы алгоритма сортировки подсчетом равно $\Theta(n)$.

Основная идея сортировки подсчетом заключается в том, чтобы для каждого входного элемента x определить количество элементов, которые меньше x. С помощью этой информации элемент x можно разместить на той позиции выходного массива, где он должен находиться. Например, если всего имеется 17 элементов, которые меньше x, то в выходной последовательности элемент x должен занимать 18-ю позицию. Если допускается ситуация, когда несколько элементов имеют одно и то же значение, эту схему придется слегка модифицировать, поскольку мы не можем разместить все такие элементы в одной и той же позиции.

При составлении кода для этого алгоритма предполагается, что на вход подается массив $A[1..n]$, так что $length[A] = n$. Потребуются еще два массива: в массиве $B[1..n]$ будет содержаться отсортированная выходная последовательность, а массив $C[0..k]$ служит временным рабочим хранилищем:

```
COUNTING_SORT(A, B, k)
 1  for i ← 0 to k
 2       do C[i] ← 0
 3  for j ← 1 to length[A]
 4       do C[A[j]] ← C[A[j]] + 1
 5  ▷ В C[i] хранится количество элементов, равных i.
 6  for i ← 1 to k
 7       do C[i] ← C[i] + C[i − 1]
 8  ▷ В C[i] — количество элементов, не превышающих i.
 9  for j ← length[A] downto 1
10       do B[C[A[j]]] ← A[j]
11          C[A[j]] ← C[A[j]] − 1
```

Работа алгоритма сортировки подсчетом проиллюстрирована на рис. 8.2. В части *а* рисунка показан массив A и вспомогательный массив C после выполнения строки 4. В части *б* рисунка массив C приведен после выполнения строки 7. В частях *в–д* показано состояние выходного массива B и вспомогательного массива C после, соответственно, одной, двух и трех итераций цикла в строках 9–11. Заполненными являются только светло-серые элементы массива B. Конечный отсортированный массив B изображен в части *е* рисунка.

После инициализации в цикле **for** в строках 1–2, в цикле **for** в строках 3–4 выполняется проверка каждого входного элемента. Если его значение равно i, то к величине $C[i]$ прибавляется единица. Таким образом, после выполнения стро-

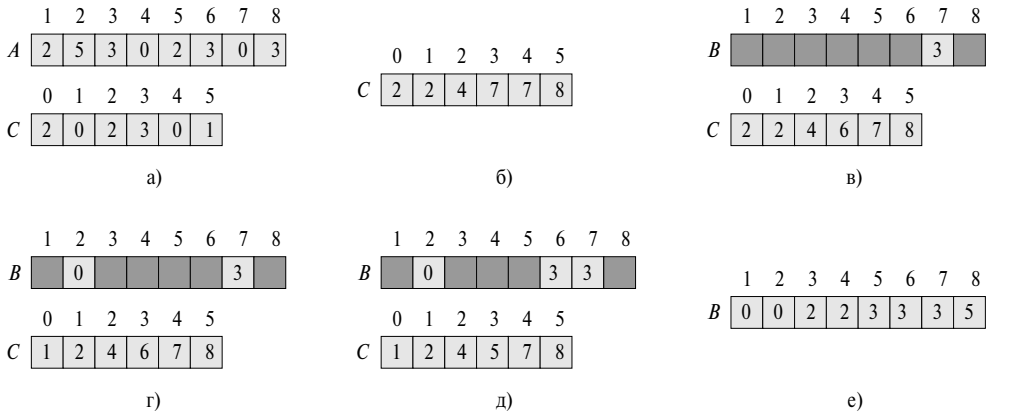


Рис. 8.2. Обработка алгоритмом COUNTING_SORT входного массива $A[1..8]$, каждый элемент которого — неотрицательное целое число, не превышающее 5

ки 4 для каждого $i = 0, 1, \ldots, k$ в переменной $C[i]$ хранится количество входных элементов, равных i. В строках 6–7 для каждого $i = 0, 1, \ldots, k$ определяется число входных элементов, не превышающих i.

Наконец, в цикле **for** в строках 9–11 каждый элемент $A[j]$ помещается в надлежащую позицию выходного массива B. Если все n элементов различны, то при первом переходе к строке 9 для каждого элемента $A[j]$ в переменной $C[A[j]]$ хранится корректный индекс конечного положения этого элемента в выходном массиве, поскольку имеется $C[A[j]]$ элементов, меньших или равных $A[j]$. Поскольку разные элементы могут иметь одни и те же значения, помещая значение $A[j]$ в массив B, мы каждый раз уменьшаем $C[A[j]]$ на единицу. Благодаря этому следующий входной элемент, значение которого равно $A[j]$ (если таковой имеется), в выходном массиве размещается непосредственно перед элементом $A[j]$.

Сколько времени требуется для сортировки методом подсчета? На выполнение цикла **for** в строках 1–2 затрачивается время $\Theta(k)$, на выполнение цикла **for** в строках 3–4 — время $\Theta(n)$, цикл в строках 6–7 требует $\Theta(k)$ времени, а цикл в строках 9–11 — $\Theta(n)$. Таким образом, полное время можно записать как $\Theta(k + n)$. На практике сортировка подсчетом применяется, когда $k = O(n)$, а в этом случае время работы алгоритма равно $\Theta(n)$.

В алгоритме сортировки подсчетом нижняя граница $\Omega(n \lg n)$, о которой шла речь в разделе 8.1, оказывается превзойденной, поскольку описанный алгоритм не основан на сравнениях. Фактически нигде в коде не производится сравнение входных элементов — вместо этого непосредственно используются их значения, с помощью которых элементам сопоставляются конкретные индексы. Нижняя же граница $\Omega(n \lg n)$ справедлива только при выполнении сортировки сравнением.

Важное свойство алгоритма сортировки подсчетом заключается в том, что он ***устойчив*** (stable): элементы с одним и тем же значением находятся в выходном массиве в том же порядке, что и во входном. Обычно свойство устойчивости важно только в ситуации, когда вместе сортируемые элементы имеют сопутствующие данные. Устойчивость, присущая сортировке подсчетом, важна еще и по другой причине: этот алгоритм часто используется в качестве подпрограммы при поразрядной сортировке. Как вы увидите в следующем разделе, устойчивость сортировки подсчетом критична для корректной работы поразрядной сортировки.

Упражнения

8.2-1. Используя в качестве модели рис. 8.2, проиллюстрируйте обработку массива $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$ процедурой COUNTING_SORT.

8.2-2. Докажите, что процедура COUNTING_SORT устойчива.

8.2-3. Предположим, что заголовок цикла **for** (строка 9) в процедуре COUNTING_SORT переписан в таком виде:

9 **for** $j \leftarrow 1$ **to** $length[A]$

Покажите, что алгоритм по-прежнему работает корректно. Устойчив ли модифицированный таким образом алгоритм?

8.2-4. Опишите алгоритм, в котором производится предварительная обработка n элементов, принадлежащих интервалу от 0 до k, после чего в течение времени $O(1)$ можно получить ответ на запрос о том, сколько входных элементов принадлежат отрезку $[a..b]$. На предварительную обработку должно использоваться $\Theta(n + k)$ времени.

8.3 Поразрядная сортировка

Поразрядная сортировка (radix sort) — это алгоритм, который использовался в машинах, предназначенных для сортировки перфокарт. Такие машины теперь можно найти разве что в музеях вычислительной техники. Перфокарты были разбиты на 80 столбцов, в каждом из которых на одной из 12 позиций можно было сделать отверстие. Сортировщик можно было механически “запрограммировать” таким образом, чтобы он проверял заданный столбец в каждой перфокарте, которая находится в колоде, и распределял перфокарты по 12 приемникам в зависимости от того, в какой позиции есть отверстие. После этого оператор получал возможность извлечь перфокарты из всех приемников и разместить их так, чтобы сверху находились перфокарты с пробитым первым разрядом, за ними — перфокарты с пробитым вторым разрядом и т.д.

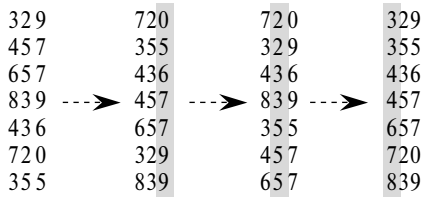


Рис. 8.3. Обработка процедурой поразрядной сортировки списка из семи трехзначных чисел

При записи десятичных цифр в каждом столбце используются лишь первые 10 разрядов (остальные два разряда нужны для кодирования символов, отличных от цифр). Таким образом, число, состоящее из d цифр, занимает поле из d столбцов. Поскольку сортировщик за один раз может обработать только один столбец, для решения задачи о сортировке n перфокарт в порядке возрастания записанных на них d-значных чисел требуется разработать некий алгоритм сортировки.

Интуиция подсказывает способ сортировки, при котором выполняется сортировка по *старшей* цифре, а затем полученные стопки рекурсивно сортируются по следующим в порядке старшинства цифрам. К сожалению, в этом случае возникает большое количество промежуточных стопок перфокарт (после первой же сортировки по старшей цифре — 10 стопок), за которыми нужно следить (см. упражнение 8.3-5.).

В алгоритме поразрядной сортировки поставленная задача решается способом, противоположным тому, что подсказывает интуиция. Сначала производится сортировка по *младшей* цифре, после чего перфокарты снова объединяются в одну колоду, в которой сначала идут перфокарты из нулевого приемника, затем — из первого приемника, затем — из второго и т.д. После этого вся колода снова сортируется по предпоследней цифре, и перфокарты вновь собираются в одну стопку тем же образом. Процесс продолжается до тех пор, пока перфокарты не окажутся отсортированными по всем d цифрам. После этого перфокарты оказываются полностью отсортированы в порядке возрастания d-значных чисел. Таким образом, для сортировки перфокарт требуется лишь d проходов колоды. На рис. 8.3 показано, как поразрядная сортировка обрабатывает “колоду” из семи трехзначных чисел. В крайнем левом столбце показаны входные числа, а в последующих столбцах — последовательные состояния списка после его сортировки по цифрам, начиная с младшей. Серым цветом выделен текущий разряд, по которому производится сортировка, в результате чего получается следующий (расположенный справа) столбец.

Важно, чтобы сортировка по цифрам того или иного разряда в этом алгоритме обладала устойчивостью. Сортировка, которая производится сортировщиком пер-

фокарт, устойчива, но оператор должен также следить за тем, чтобы не перепутать порядок перфокарт после извлечения их из приемника. Это важно, несмотря на то, что на всех перфокартах из одного и того же приемника в столбце, номер которого соответствует этапу обработки, стоит одна и та же цифра.

В типичных компьютерах, представляющих собой машины с произвольным доступом к памяти, поразрядная сортировка иногда применяется для приведения в порядок записей, ключи которых разбиты на несколько полей. Например, пусть нужно выполнить сортировку дат по трем ключам: год, месяц и день. Для этого можно было бы запустить алгоритм сортировки с функцией сравнения, в котором в двух заданных датах сначала бы сравнивались годы, при их совпадении сравнивались месяцы, а при совпадении и тех, и других сравнивались бы дни. Можно поступить и по-другому, т.е. выполнить трехкратную сортировку с помощью устойчивой процедуры: сначала по дням, потом по месяцам и наконец по годам.

Разработать код поразрядной сортировки не составляет труда. В приведенной ниже процедуре предполагается, что каждый из n элементов массива A — это число, в котором всего d цифр, причем первая цифра стоит в самом младшем разряде, а цифра под номером d — в самом старшем разряде:

```
RADIX_SORT(A, d)
1  for i ← 1 to d
2       do Устойчивая сортировка массива A по i-ой цифре
```

Лемма 8.3. Пусть имеется n d-значных чисел, в которых каждая цифра принимает одно из k возможных значений. Тогда алгоритм RADIX_SORT позволяет выполнить корректную сортировку этих чисел за время $\Theta(d(n+k))$, если устойчивая сортировка, используемая данным алгоритмом, имеет время работы $\Theta(n+k)$.

Доказательство. Корректность поразрядной сортировки доказывается методом математической индукции по столбцам, по которым производится сортировка (см. упражнение 8.3-3). Анализ времени работы рассматриваемого алгоритма зависит от того, какой из методов устойчивой сортировки используется в качестве промежуточного алгоритма сортировки. Если каждая цифра принадлежит интервалу от 0 до $k-1$ (и, таким образом, может принимать k возможных значений), и k не слишком большое, то поразрядная сортировка — оптимальный выбор. Для обработки каждой из d цифр n чисел понадобится время $\Theta(n+k)$, а все цифры будут обработаны алгоритмом поразрядной сортировки в течение времени $\Theta(d(n+k))$. ■

Если d — константа, а $k = O(n)$, то время работы алгоритма поразрядной сортировки линейно зависит от количества входных элементов. В общем случае мы получаем определенную степень свободы в выборе разбиения ключей на цифры.

Лемма 8.4. Пусть имеется n b-битовых чисел и натуральное число $r \leqslant b$. Алгоритм RADIX_SORT позволяет выполнить корректную сортировку этих чисел за время $Q\left((b/r)\left(n+2^r\right)\right)$.

Доказательство. Для $r \leqslant b$ каждый ключ можно рассматривать как число, состоящее из $d = \lceil b/r \rceil$ цифр по r битов каждая. Все цифры представляют собой целые числа в интервале от 0 до $2^r - 1$, поэтому можно воспользоваться алгоритмом сортировки подсчетом, в котором $k = 2^r - 1$ (например, 32-битовое слово можно рассматривать как число, состоящее из четырех 8-битовых цифр, так что $b = 32$, $r = 8$, $k = 2^r - 1 = 255$, а $d = b/r = 4$). Каждый проход сортировки подсчетом занимает время $\Theta\left(n+k\right) = \Theta\left(n+2^r\right)$, а всего выполняется d проходов, так что полное время выполнения алгоритма равно $\Theta\left(d\left(n+2^r\right)\right) = \Theta\left((b/r)\left(n+2^r\right)\right)$. ■

Выберем для двух заданных значений n и b такую величину $r \leqslant b$, которая бы сводила к минимуму выражение $(b/r)\left(n+2^r\right)$. Если $b < \lfloor \lg n \rfloor$, то для любого значения $r \leqslant b$ имеем $\left(n+2^r\right) = \Theta\left(n\right)$. Таким образом, выбор $r = b$ приводит к асимптотически оптимальному времени работы $(b/b)\left(n+2^b\right) = \Theta\left(n\right)$. Если же $b \geqslant \lfloor \lg n \rfloor$, то наилучшее время с точностью до постоянного множителя можно получить, выбрав $r = \lfloor \lg n \rfloor$. Это можно понять из следующих рассуждений. Если выбрать $r = \lfloor \lg n \rfloor$, то время выполнения алгоритма будет равно $\Theta\left(bn/\lg n\right)$. Если r увеличивается и превышает значение $\lfloor \lg n \rfloor$, то член 2^r в числителе возрастает быстрее, чем член r в знаменателе, поэтому увеличение r приводит ко времени работы алгоритма, равному $\Omega\left(bn/\lg n\right)$. Если же величина r уменьшается и становится меньше $\lfloor \lg n \rfloor$, то член b/r возрастает, а множитель $n + 2^r$ остается величиной порядка $\Theta\left(n\right)$.

Является ли поразрядная сортировка более предпочтительной, чем сортировка сравнением, например, быстрая сортировка? Если $b = O\left(\lg n\right)$, как это часто бывает, и мы выбираем $r \approx \lg n$, то время работы алгоритма поразрядной сортировки равно $\Theta\left(n\right)$, что выглядит предпочтительнее среднего времени выполнения быстрой сортировки $\Theta\left(n \lg n\right)$. Однако в этих выражениях, записанных в Θ-обозначениях, разные постоянные множители. Несмотря на то, что для поразрядной сортировки n ключей может понадобиться меньше проходов, чем для их быстрой сортировки, каждый проход при поразрядной сортировке может длиться существенно дольше. Выбор подходящего алгоритма сортировки зависит от характеристик реализации алгоритма, от машины, на которой производится сортировка (например, при быстрой сортировке аппаратный кэш часто используется эффективнее, чем при поразрядной сортировке), а также от входных данных. Кроме того, в той версии поразрядной сортировки, в которой в качестве промежуточного этапа используется устойчивая сортировка подсчетом, обработка элементов производится с привлечением дополнительной памяти, которая не нужна во многих алгоритмах сортировки сравнением, время работы которых равно $\Theta\left(n \lg n\right)$.

Таким образом, если в первую очередь нужно учитывать объем расходуемой памяти, может оказаться более предпочтительным использование, например, алгоритма быстрой сортировки, в котором элементы обрабатываются на месте.

Упражнения

8.3-1. Используя в качестве модели рис. 8.3, проиллюстрируйте, как с помощью процедуры RADIX_SORT сортируется следующий список английских слов: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

8.3-2. Какие из перечисленных ниже алгоритмов сортировки устойчивы: сортировка вставкой, сортировка слиянием, пирамидальная сортировка, быстрая сортировка? Приведите пример схемы, в результате применения которой любой алгоритм сортировки становится устойчивым. Оцените количество дополнительного времени и объем памяти, необходимые для реализации этой схемы.

8.3-3. Докажите методом математической индукции корректность работы поразрядной сортировки. Где в доказательстве используется предположение об устойчивости промежуточной сортировки?

8.3-4. Покажите, как выполнить сортировку n чисел, принадлежащих интервалу от 0 до $n^2 - 1$, за время $O(n)$.

⋆ 8.3-5. Определите точное количество проходов, которые понадобятся для сортировки d-значных десятичных чисел в наихудшем случае, если используется сортировка перфокарт, начинающаяся со старшей цифры. За каким количеством стопок перфокарт нужно будет следить оператору в наихудшем случае?

8.4 Карманная сортировка

Если входные элементы подчиняются равномерному закону распределения, то ожидаемое время работы алгоритма ***карманной сортировки*** (bucket sort) линейно зависит от количества входных элементов. Карманная сортировка, как и сортировка подсчетом, работает быстрее, чем алгоритмы сортировки сравнением. Это происходит благодаря определенным предположениям о входных данных. Если при сортировке методом подсчета предполагается, что входные данные состоят из целых чисел, принадлежащих небольшому интервалу, то при карманной сортировке предполагается, что входные числа генерируются случайным процессом и равномерно распределены в интервале $[0, 1)$ (определение равномерного распределения можно найти в разделе B.2).

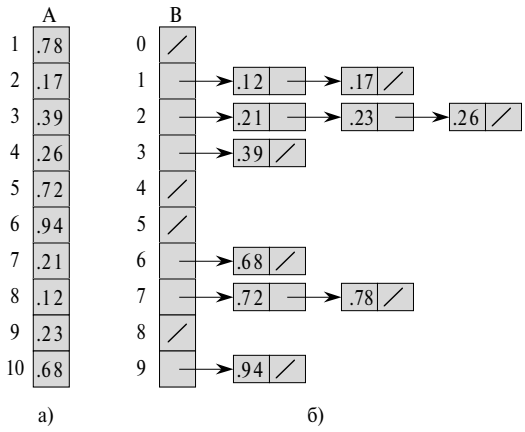


Рис. 8.4. Процедура BUCKET_SORT в действии

Идея, лежащая в основе карманной сортировки, заключается в том, чтобы разбить интервал $[0, 1)$ на n одинаковых интервалов, или ***карманов*** (buckets), а затем распределить по этим карманам n входных величин. Поскольку входные числа равномерно распределены в интервале $[0, 1)$, мы предполагаем, что в каждый из карманов попадет *не* много элементов. Чтобы получить выходную последовательность, нужно просто выполнить сортировку чисел в каждом кармане, а затем последовательно перечислить элементы каждого кармана.

При составлении кода карманной сортировки предполагается, что на вход подается массив A, состоящий из n элементов, и что величина каждого принадлежащего массиву элемента $A[i]$ удовлетворяет неравенству $0 \leqslant A[i] < 1$. Для работы нам понадобится вспомогательный массив связанных списков (карманов) $B[0..n-1]$; предполагается, что в нашем распоряжении имеется механизм поддержки таких списков. (Реализация основных операций при работе со связанным списком описана в разделе 10.2.)

BUCKET_SORT(A)
1 $n \leftarrow length[A]$
2 **for** $i \leftarrow 1$ **to** n
3 **do** Вставить элемент $A[i]$ в список $B[\lfloor nA[i] \rfloor]$
4 **for** $i \leftarrow 0$ **to** $n-1$
5 **do** Сортировка вставкой списка $B[i]$
6 Объединение списков $B[0], B[1], \ldots, B[n-1]$

Чтобы понять, как работает этот алгоритм, рассмотрим два элемента $A[i]$ и $A[j]$. Без потери общности можно предположить, что $A[i] \leqslant A[j]$. Поскольку $\lfloor nA[i] \rfloor \leqslant \lfloor nA[j] \rfloor$, то элемент $A[i]$ помещается либо в тот же карман, что и элемент $A[j]$, либо в карман с меньшим индексом. В первом случае элементы

$A[i]$ и $A[j]$ располагаются в нужном порядке благодаря циклу **for** в строках 4–5. Если же эти элементы попадут в разные карманы, то они разместятся в правильном порядке после выполнения строки 6. Таким образом, карманная сортировка работает корректно.

На рис. 8.4 показано, как с помощью карманной сортировки обрабатывается входной массив, состоящий из 10 чисел. В части *а* этого рисунка показан входной массив $A[1..10]$. В части *б* изображен массив $B[0..9]$, состоящий из отсортированных списков после выполнения строки 5 алгоритма. В i-м кармане содержатся величины, принадлежащие полуоткрытому интервалу $[i/10, (i+1)/10)$. Отсортированная выходная последовательность представляет собой объединение списков $B[0], B[1], \ldots, B[9]$.

Чтобы оценить время работы алгоритма, заметим, что в наихудшем случае для выполнения всех его строк, кроме строки 5, требуется время $O(n)$. Остается просуммировать полное время, которое потребуется для n вызовов алгоритма сортировки методом вставок (строка 5).

Чтобы оценить стоимость этих вызовов, введем случайную величину n_i, обозначающую количество элементов, попавших в карман $B[i]$. Поскольку время работы алгоритма сортировки вставкой является квадратичной функцией от количества входных элементов (см. раздел 2.2), время работы алгоритма карманной сортировки равно

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2).$$

Вычисляя математическое ожидание обеих частей этого уравнения и воспользовавшись его линейностью, с учетом уравнения (B.21) получаем:

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] = \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] = \\ &= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right) \end{aligned} \tag{8.1}$$

Мы утверждаем, что для всех $i = 0, 1, \ldots, n-1$

$$\mathrm{E}\left[n_i^2\right] = 2 - 1/n. \tag{8.2}$$

Не удивительно, что каждому i-му карману соответствует одна и та же величина $\mathrm{E}\left[n_i^2\right]$, поскольку все элементы входного массива могут попасть в любой карман

с равной вероятностью. Чтобы доказать уравнение (8.2), определим для каждого $i = 0, 1, \ldots, n-1$ и $j = 1, 2, \ldots, n$ индикаторную случайную величину

$$X_{ij} = \mathrm{I}\{A[j] \text{ попадает в } i\text{-ый карман}\}.$$

Следовательно,

$$n_i = \sum_{j=1}^{n} X_{ij}.$$

Чтобы вычислить величину $\mathrm{E}\left[n_i^2\right]$, раскроем квадрат и перегруппируем слагаемые:

$$\begin{aligned}
\mathrm{E}\left[n_i^2\right] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] = \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] = \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\leqslant j\leqslant n}\ \sum_{\substack{1\leqslant k\leqslant n\\ k\neq j}} X_{ij}X_{ik}\right] = \\
&= \sum_{j=1}^{n} \mathrm{E}\left[X_{ij}^2\right] + \sum_{1\leqslant j\leqslant n}\ \sum_{\substack{1\leqslant k\leqslant n\\ k\neq j}} \mathrm{E}\left[X_{ij}X_{ik}\right]
\end{aligned} \tag{8.3}$$

В приведенной выше цепочке уравнений последнее равенство следует из линейности математического ожидания. Отдельно оценим обе суммы. Индикаторная случайная величина X_{ij} равна 1 с вероятностью $1/n$ и 0 в противном случае, поэтому получаем:

$$\mathrm{E}\left[X_{ij}^2\right] = 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) = \frac{1}{n}.$$

Когда $k \neq j$, величины X_{ij} и X_{ik} независимы, поэтому можно записать:

$$\mathrm{E}\left[X_{ij}X_{ik}\right] = \mathrm{E}\left[X_{ij}\right]\mathrm{E}\left[X_{ik}\right] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}.$$

Подставив эти величины в уравнение (8.3), получим:

$$\mathrm{E}\left[n_i^2\right] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1\leqslant j\leqslant n}\ \sum_{\substack{1\leqslant k\leqslant n\\ k\neq j}} \frac{1}{n^2} = n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} = 1 + \frac{n-1}{n} = 2 - \frac{1}{n},$$

что и доказывает уравнение (8.2).

Используя это выражение в уравнении (8.1), приходим к выводу, что математическое ожидание времени работы алгоритма карманной сортировки равно

$\Theta(n)+n\cdot O(2-1/n)=\Theta(n)$. Таким образом, математическое ожидание времени работы алгоритма карманной сортировки в целом линейно зависит от количества входных элементов.

Такая зависимость может наблюдаться даже в том случае, когда входные элементы не подчиняются закону равномерного распределения. Если входные элементы обладают тем свойством, что сумма возведенных в квадрат размеров карманов линейно зависит от количества входных элементов, уравнение (8.1) утверждает, что карманная сортировка этих данных выполняется в течение времени, линейно зависящего от количества данных.

Упражнения

8.4-1. Используя в качестве модели рис. 8.4, проиллюстрируйте обработку алгоритмом BUCKET_SORT массива $A=\langle 0.79, 0.13, 0.16, 0.64, 0.39, 0.20, 0.89, 0.53, 0.71, 0.42\rangle$.

8.4-2. Чему равно время работы алгоритма карманной сортировки в наихудшем случае? Какое простое изменение следует внести в этот алгоритм, чтобы его ожидаемое время работы осталось линейным, а время работы в наихудшем случае стало равным $O(n\lg n)$?

⋆ 8.4-3. ***Функция распределения вероятности*** $P(x)$ случайной величины X определяется с помощью соотношения $P(x)=\Pr\{X\leqslant x\}$. Предположим, что на вход алгоритма поступает последовательность из n случайных величин $X_1, X_2, \ldots, X_n$ с непрерывной функцией распределения P, значение которой вычисляется в течение времени $O(1)$. Покажите, каким образом выполнить сортировку этих величин, чтобы математическое ожидание времени выполнения процедуры линейно зависело от их количества.

Задачи

8-1. **Нижние оценки для сортировки сравнением в среднем случае**

В этой задаче мы докажем, что нижняя граница математического ожидания времени работы любого детерминистического или рандомизированного алгоритма сортировки сравнением при обработке n различающихся входных элементов равна $\Omega(n\lg n)$. Начнем с того, что рассмотрим детерминистическую сортировку сравнением A, которой соответствует дерево решений T_A. Предполагается, что все перестановки входных элементов A равновероятны.

а) Предположим, что каждый лист дерева T_A помечен вероятностью его достижения при заданном случайном наборе входных данных.

Докажите, что ровно $n!$ листьям соответствует вероятность $1/n!$, а остальным — вероятность 0.

б) Пусть $D(T)$ — длина внешнего пути дерева решений T; другими словами, это сумма глубин всех листьев этого дерева. Пусть T — дерево решений с $k > 1$ листьями, а LT и RT — его левое и правое поддеревья. Покажите, что $D(T) = D(LT) + D(RT) + k$.

в) Пусть $d(k)$ — минимальная величина $D(T)$ среди всех деревьев решений T с $k > 1$ листьями. Покажите, что $d(k) = \min_{1 \leqslant i \leqslant k-1}\{d(i) + d(k-i) + k\}$. (*Указание*: рассмотрите дерево решений T с k листьями, на котором достигается минимум. Обозначьте количество листьев в LT через i_0, а количество листьев в RT — через $k - i_0$.)

г) Докажите, что для данного $k > 1$ и i из диапазона $1 \leqslant i \leqslant k-1$ функция $i \lg i + (k-i) \lg (k-i)$ достигает минимума при $i = k/2$. Выведите отсюда, что $d(k) = \Omega(k \lg k)$.

д) Докажите, что справедливо соотношение $D(T_A) = \Omega(n! \lg(n!))$, и выведите отсюда, что математическое ожидание времени сортировки n элементов равно $\Omega(n \lg n)$.

А теперь рассмотрим *рандомизированную* сортировку B. Модель дерева решений можно обобщить таким образом, чтобы с ее помощью можно было рассматривать рандомизированные алгоритмы. Для этого в нее нужно включить узлы двух видов: обычные узлы сравнения и узлы "рандомизации", которые моделируют случайный выбор вида RANDOM$(1, r)$ в алгоритме B; такой узел имеет r дочерних узлов, каждый из которых в процессе выполнения алгоритма может быть выбран с равной вероятностью.

е) Покажите, что для любой рандомизированной сортировки сравнением B существует детерминистическая сортировка сравнением A, в которой в среднем производится не больше сравнений, чем в сортировке B.

8-2. **Сортировка на месте за линейное время**

Предположим, что у нас имеется массив, содержащий n записей с сортируемыми данными, и что ключ каждой записи принимает значения 0 или 1. Алгоритм, предназначенный для сортировки такого набора записей, должен обладать некоторыми из трех перечисленных ниже характеристик.

1) Время работы алгоритма равно $O(n)$.

2) Алгоритм обладает свойством устойчивости.

3) Сортировка производится на месте, т.е. кроме исходного массива используется дополнительное пространство, не превышающее некоторой постоянной величины.

   а) Разработайте алгоритм, удовлетворяющий критериям 1 и 2.

   б) Разработайте алгоритм, удовлетворяющий критериям 1 и 3.

   в) Разработайте алгоритм, удовлетворяющий критериям 2 и 3.

   г) Может ли какой-либо из представленных в частях *а*)–*в*) алгоритмов обеспечить поразрядную сортировку n записей с b-битовыми ключами за время $O(bn)$? Поясните, почему.

   д) Предположим, что n записей обладают ключами, значения которых находятся в интервале от 1 до k. Покажите, как можно модифицировать алгоритм сортировки подсчетом, чтобы обеспечить сортировку этих записей на месте в течение времени $O(n + k)$. В дополнение к входному массиву, можно использовать дополнительную память объемом $O(k)$. Устойчив ли этот алгоритм? (*Указание*: подумайте, как можно решить задачу для $k = 3$.)

8-3. **Сортировка элементов переменной длины**

   а) Имеется массив целых чисел, причем различные элементы этого массива могут иметь разные количества цифр; однако общее количество цифр во *всех* числах равно n. Покажите, как выполнить сортировку этого массива за время $O(n)$.

   б) Имеется массив строк, в котором различные строки могут иметь разную длину; однако общее количество символов во *всех* строках равно n. Покажите, как выполнить сортировку этого массива за время $O(n)$.

   (Порядок сортировки в этой задаче определяется обычным алфавитным порядком; например, $a < ab < b$.)

8-4. **Кувшины для воды**

Предположим, имеется n красных и n синих кувшинов для воды, которые различаются формой и объемом. Все красные кувшины могут вместить разное количество воды; то же относится и к синим кувшинам. Кроме того, каждому красному кувшину соответствует синий кувшин того же объема и наоборот.

Задача заключается в том, чтобы разбить кувшины на пары, в каждой из которых будут красный и синий кувшин одинакового объема. Для этого можно использовать такую операцию: сформировать пару кувшинов, в которых один будет синим, а второй — красным, наполнить красный

кувшин водой, а затем перелить ее в синий кувшин. Эта операция позволит узнать, какой из кувшинов больше или является ли их объем одинаковым. Предположим, что для выполнения такой операции требуется одна единица времени. Необходимо сформулировать алгоритм, в котором для разбиения кувшинов на пары производилось бы минимальное количество сравнений. Напомним, что непосредственно сравнивать два красных или два синих кувшина нельзя.

а) Опишите детерминистический алгоритм, в котором разбиение кувшинов на пары производилось бы с помощью $\Theta\left(n^2\right)$ сравнений.

б) Докажите, что нижняя граница количества сравнений, которые должен выполнить алгоритм, предназначенный для решения этой задачи, равна $\Omega\left(n \lg n\right)$.

в) Разработайте рандомизированный алгоритм, математическое ожидание количества сравнений в котором было бы равно $O\left(n \lg n\right)$, и докажите корректность этой границы. Чему равно количество сравнений в этом алгоритме в наихудшем случае?

8-5. **Сортировка в среднем**

Предположим, что вместо сортировки массива нам нужно, чтобы его элементы возрастали в среднем. Точнее говоря, n-элементный массив A называется ***k-отсортированным***, если для всех $i = 1, 2, \ldots, n - k$ выполняется такое соотношение:

$$\frac{\sum_{j=i}^{i+k-1} A\left[j\right]}{k} \leqslant \frac{\sum_{j=i+1}^{i+k} A\left[j\right]}{k}.$$

а) Что из себя представляет 1-отсортированный массив?

б) Приведите пример перестановки чисел 1, 2, ..., 10, являющейся 2-отсортированной, но *не* отсортированной.

в) Докажите, что n-элементный массив k-отсортирован тогда и только тогда, когда для всех $i = 1, 2, \ldots, n - k$ справедливо соотношение $A\left[i\right] \leqslant A\left[i + k\right]$.

г) Разработайте алгоритм, который выполняет k-сортировку n-элементного массива за время $O\left(n \lg\left(n/k\right)\right)$.

Можно также найти нижнюю границу для времени, необходимого для получения k-отсортированного массива, если k — константа.

д) Покажите, что k-сортировку массива длины n можно выполнить за время $O\left(n \lg k\right)$. (*Указание*: воспользуйтесь решением упражнения 6.5-8.)

е) Покажите, что если k — константа, то для k-сортировки n-элементного массива потребуется время $\Omega(n \lg n)$. (*Указание*: воспользуйтесь решением предыдущего задания вместе с нижней границей для алгоритмов сортировки сравнением.)

8-6. **Нижняя граница для объединения отсортированных списков**

Часто возникает задача объединения двух отсортированных списков. Эта процедура используется в качестве подпрограммы в процедуре MERGE_SORT; она описана в разделе 2.3.1, где получила имя MERGE. В настоящей задаче мы покажем, что для количества сравнений, необходимых для объединения двух n-элементных отсортированных списков в наихудшем случае, существует нижняя граница, равная $2n - 1$ сравнениям.

Сначала с помощью дерева решений покажем, что нижняя граница количества сравнений равна $2n - o(n)$.

а) Покажите, что всего имеется $\binom{2n}{n}$ способов разбиения $2n$ чисел на два отсортированных списка, в каждом из которых n чисел.

б) Покажите с помощью дерева решений, что в любом алгоритме, корректно объединяющем два отсортированных списка, выполняется не менее $2n - o(n)$ сравнений.

Теперь мы докажем наличие несколько более точной границы $2n - 1$.

в) Покажите, что если два элемента, которые в объединенном списке будут расположены последовательно один за другим, принадлежат разным подлежащим объединению спискам, то в ходе объединения эти элементы обязательно придется сравнить.

г) С помощью ответа на предыдущий пункт задачи покажите, что нижняя граница для количества сравнений, которые производятся при объединении двух отсортированных списков, равна $2n - 1$.

Заключительные замечания

Использовать модель дерева решений при изучении алгоритмов сортировки сравнением впервые предложили Форд (Ford) и Джонсон (Johnson) [94]. В томе *Искусства программирования* Кнута (Knuth), посвященном сортировке [185], описываются различные разновидности задачи сортировки, включая приведенную здесь теоретико-информационную нижнюю границу для сложности сортировки. Полное исследование нижних границ сортировки с помощью обобщений модели дерева решений было проведено Бен-Ором (Ben-Or) [36].

Согласно Кнуту, сортировка подсчетом впервые была предложена Севардом (H.H. Seward) в 1954 году; ему также принадлежит идея объединения сортировки

подсчетом и поразрядной сортировки. Оказывается, что поразрядная сортировка, начинающаяся с самой младшей значащей цифры, — по сути "народный" алгоритм, широко применявшийся операторами механических машин, предназначенных для сортировки перфокарт. Как утверждает Кнут, первая ссылка на этот метод появилась в документе, составленном Комри (L.J. Comrie) и опубликованном в 1929 году, где описывается счетно-перфорационное оборудование. Карманная сортировка используется с 1956 года, с того времени, когда Исаак (E.J. Isaac) и Синглтон (R.C. Singleton) предложили основную идею этого метода.

Мунро (Munro) и Раман (Raman) [229] предложили устойчивый алгоритм сортировки, который в наихудшем случае выполняет $O\left(n^{1+\varepsilon}\right)$ сравнений, где $0 < \varepsilon \leqslant 1$ — произвольная константа. Несмотря на то, что в алгоритмах, время выполнения которых равно $O(n \lg n)$, выполняется меньше сравнений, в алгоритме Мунро и Рамана данные перемещаются $O(n)$ раз, и он выполняет сортировку на месте.

Случай сортировки n b-битовых чисел за время $o(n \lg n)$ рассматривался многими исследователями. Было получено несколько положительных результатов, в каждом из которых незначительно менялись предположения о вычислительной модели, а также накладываемые на алгоритм ограничения. Во всех случаях предполагалось, что память компьютера разделена на адресуемые b-битовые слова. Фредман (Fredman) и Виллард (Willard) [99] первыми предложили использовать дерево слияний (fusion tree) и выполнять с его помощью сортировку n целых чисел за время $O(n \lg n / \lg \lg n)$. Впоследствии эта граница была улучшена Андерссоном (Andersson) [16] до $O\left(n\sqrt{\lg n}\right)$. В этих алгоритмах применяется операция умножения, а также некоторые предварительно вычисляемые константы. Андерссон, Хейгерап (Hagerup), Нильсон (Nilsson) и Раман [17] показали, как выполнить сортировку n чисел за время $O(n \lg \lg n)$, не используя при этом умножение, однако этот метод требует дополнительной памяти, объем которой увеличивается с ростом n. С помощью мультипликативного хеширования объем этого пространства можно уменьшить до величины $O(n)$, но при этом граница для наихудшего случая $O(n \lg \lg n)$ становится границей для математического ожидания времени работы. Обобщив экспоненциальные деревья поиска, Андерссон [16] и Торуп (Thorup) [297] сформулировали алгоритм сортировки, выполняющийся в течение времени $O\left(n (\lg \lg n)^2\right)$. В этом алгоритме не используется ни умножение, ни рандомизация, а объем необходимой дополнительной памяти линейно зависит от количества элементов. Дополнив эти методы новыми идеями, Хан (Han) [137] улучшил границу времени работы до $O(n \lg \lg n \lg \lg \lg n)$. Несмотря на то, что упомянутые алгоритмы стали важным теоретическим достижением, все они чрезвычайно сложные, и в данный момент представляется маловероятным, чтобы они могли практически составить конкуренцию существующим алгоритмам сортировки.

ГЛАВА 9

Медианы и порядковые статистики

Будем называть i-й ***порядковой статистикой*** (order statistic) множества, состоящего из n элементов, i-й элемент в порядке возрастания. Например, ***минимум*** такого множества — это первая порядковая статистика ($i = 1$), а его ***максимум*** — это n-я порядковая статистика ($i = n$). ***Медиана*** (median) неформально обозначает середину множества. Если n нечетное, то медиана единственная, и ее индекс равен $i = (n + 1)/2$; если же n четное, то медианы две, и их индексы равны $i = n/2$ и $i = n/2 + 1$. Таким образом, независимо от четности n, медианы располагаются при $i = \lfloor (n + 1)/2 \rfloor$ (***нижняя медиана*** (lower median)) и $i = \lceil (n + 1)/2 \rceil$ (***верхняя медиана*** (upper median)). Однако в этой книге для простоты используется выражение "медиана", которое относится к нижней медиане.

Данная глава посвящена проблеме выбора i-й порядковой статистики в множестве, состоящем из n различных чисел. Для удобства предположим, что все числа в множестве различны, хотя почти все, что мы будем делать, обобщается на случай, когда некоторые значения в множестве повторяются. Формально ***задачу выбора*** (selection problem) можно определить следующим образом.

Вход: множество A, состоящее из n (различных) чисел, и число $1 \leqslant i \leqslant n$.

Выход: элемент $x \in A$, превышающий по величине ровно $i - 1$ других элементов множества A.

Задачу выбора можно решить за время $O(n \lg n)$. Для этого достаточно выполнить сортировку элементов с помощью пирамидальной сортировки или сортировки слиянием, а затем просто извлечь элемент выходного массива с индексом i. Однако есть и более быстрые алгоритмы.

В разделе 9.1 рассматривается задача о выборе минимального и максимального элементов множества. Больший интерес представляет общая задача выбора,

которая исследуется в двух последующих разделах. В разделе 9.2 анализируется применяющийся на практике алгоритм, время работы которого в среднем составляет $O(n)$ (предполагается, что все элементы различны). В разделе 9.3 приведен алгоритм, представляющий больший практический интерес, время работы которого достигает величины $O(n)$ в наихудшем случае.

9.1 Минимум и максимум

Сколько сравнений необходимо для того, чтобы найти минимальный элемент в n-элементном множестве? Для этой величины легко найти верхнюю границу, равную $n-1$ сравнениям: мы по очереди проверяем каждый элемент множества и следим за тем, какой из них является минимальным на данный момент. В представленной ниже процедуре предполагается, что исследуется множество, состоящее из элементов массива A, где $length[A] = n$:

```
MINIMUM(A)
1  min ← A[1]
2  for i ← 2 to length[A]
3        do if min > A[i]
4              then min ← A[i]
5  return min
```

Очевидно, что для поиска максимального элемента также понадобится не более $n-1$ сравнений.

Является ли представленный выше алгоритм оптимальным? Да, поскольку можно доказать, что нижняя граница для задачи определения минимума также равна $n-1$ сравнений. Любой алгоритм, предназначенный для определения минимального элемента множества, можно представить в виде турнира, в котором принимают участие все элементы. Каждое сравнение — это поединок между двумя элементами, в котором побеждает элемент с меньшей величиной. Важное наблюдение заключается в том, что каждый элемент, кроме минимального, должен потерпеть поражение хотя бы в одном поединке. Таким образом, для определения минимума понадобится $n-1$ сравнений, и алгоритм MINIMUM является оптимальным по отношению к количеству производимых в нем сравнений.

Одновременный поиск минимума и максимума

В некоторых приложениях возникает необходимость найти как минимальный, так и максимальный элементы множества. Например, графической программе может понадобиться выполнить масштабирование множества координат (x, y) таким образом, чтобы они совпали по размеру с прямоугольной областью экрана или

другого устройства вывода. Для этого сначала надо определить максимальную и минимальную координаты.

Несложно разработать алгоритм для поиска минимума и максимума в n-элементном множестве, производя при этом $\Theta(n)$ сравнений; при этом алгоритм будет асимптотически оптимальным. Достаточно просто выполнить независимый поиск минимального и максимального элементов. Для выполнения каждой подзадачи понадобится $n-1$ сравнений, что в сумме составит $2n-2$ сравнений.

Однако на самом деле для одновременного определения минимума и максимума достаточно не более $3\lfloor n/2 \rfloor$ сравнений. Для этого необходимо следить за тем, какой из проверенных на данный момент элементов минимальный, а какой — максимальный. Вместо того, чтобы отдельно сравнивать каждый входной элемент с текущим минимумом и максимумом (для чего пришлось бы на каждый элемент израсходовать по два сравнения), мы будем обрабатывать пары элементов. Образовав пару входных элементов, сначала сравним их *один с другим*, а затем меньший элемент пары будем сравнивать с текущим минимумом, а больший — с текущим максимумом. Таким образом, для каждой пары элементов понадобится по 3 сравнения.

Способ начального выбора текущего минимума и максимума зависит от четности количества элементов в множестве n. Если n нечетно, мы выбираем из множества один из элементов и считаем его значение одновременно и минимумом, и максимумом; остальные элементы обрабатываем парами. Если же n четно, то выбираем два первых элемента и путем сравнения определяем, значение какого из них будет минимумом, а какого — максимумом. Остальные элементы обрабатываем парами, как и в предыдущем случае.

Проанализируем, чему равно полное число сравнений. Если n нечетно, то нужно будет выполнить $3\lfloor n/2 \rfloor$ сравнений. Если же n четно, то выполняется одно начальное сравнение, а затем — еще $3(n-2)/2$ сравнений, что в сумме дает $3n/2-2$ сравнений. Таким образом, в обоих случаях полное количество сравнений не превышает $3\lfloor n/2 \rfloor$.

Упражнения

9.1-1. Покажите, что для поиска второго в порядке возрастания элемента в наихудшем случае достаточно $n + \lceil \lg n \rceil - 2$ сравнений. (*Указание*: найдите заодно и наименьший элемент.)

9.1-2. Покажите, что в наихудшем случае для поиска максимального и минимального среди n чисел необходимо выполнить $\lceil 3n/2 \rceil - 2$ сравнений. (*Указание*: рассмотрите вопрос о том, сколько чисел являются потенциальными кандидатами на роль максимума или минимума, и определите, как на это количество влияет каждое сравнение.)

9.2 Выбор в течение линейного ожидаемого времени

Общая задача выбора оказывается более сложной, чем простая задача поиска минимума. Однако, как это ни странно, время решения обеих задач в асимптотическом пределе ведет себя одинаково — как $\Theta(n)$. В данном разделе вниманию читателя представляется алгоритм типа "разделяй и властвуй" RANDOMIZED_SELECT, предназначенный для решения задачи выбора. Этот алгоритм разработан по аналогии с алгоритмом быстрой сортировки, который рассматривался в главе 7. Как и в алгоритме быстрой сортировки, в алгоритме RANDOMIZED_SELECT используется идея рекурсивного разбиения входного массива. Однако в отличие от алгоритма быстрой сортировки, в котором рекурсивно обрабатываются обе части разбиения, алгоритм RANDOMIZED_SELECT работает лишь с одной частью. Это различие проявляется в результатах анализа обоих алгоритмов: если математическое ожидание времени работы алгоритма быстрой сортировки равно $\Theta(n \lg n)$, то ожидаемое время работы алгоритма RANDOMIZED_SELECT равно $\Theta(n)$, в предположении, что все элементы входных множеств различны.

В алгоритме RANDOMIZED_SELECT используется процедура RANDOMIZED_PARTITION, впервые представленная в разделе 7.3. Таким образом, подобно процедуре RANDOMIZED_QUICKSORT, RANDOMIZED_SELECT — это рандомизированный алгоритм, поскольку его поведение частично определяется выводом генератора случайных чисел. Приведенный ниже код процедуры RANDOMIZED_SELECT возвращает i-й в порядке возрастания элемент массива $A[p..r]$:

```
RANDOMIZED_SELECT(A, p, r, i)
1  if p = r
2     then return A[p]
3  q ← RANDOMIZED_PARTITION(A, p, r)
4  k ← q − p + 1
5  if i = k            ▷ Опорное значение — это ответ
6     then return A[q]
7  elseif i < k
8     then return RANDOMIZED_SELECT(A, p, q − 1, i)
9  else return RANDOMIZED_SELECT(A, q + 1, r, i − k)
```

После выполнения процедуры RANDOMIZED_PARTITION, которая вызывается в строке 3 представленного выше алгоритма, массив $A[p..r]$ оказывается разбитым на два (возможно, пустых) подмассива $A[p..q-1]$ и $A[q+1..r]$. При этом величина каждого элемента $A[p..q-1]$ не превышает $A[q]$, а величина каждого элемента $A[q+1..r]$ больше $A[q]$. Как и в алгоритме быстрой сортировки, элемент $A[q]$ будем называть ***опорным*** (pivot). В строке 4 процедуры RANDOM-

IZED_SELECT вычисляется количество элементов k подмассива $A[p..q]$, т.е. количество элементов, попадающих в нижнюю часть разбиения плюс один опорный элемент. Затем в строке 5 проверяется, является ли элемент $A[q]$ i-м в порядке возрастания элементом. Если это так, то возвращается элемент $A[q]$. В противном случае в алгоритме определяется, в каком из двух подмассивов содержится i-й в порядке возрастания элемент: в подмассиве $A[p..q-1]$ или в подмассиве $A[q+1..r]$. Если $i < k$, то нужный элемент находится в нижней части разбиения, и он рекурсивно выбирается из соответствующего подмассива в строке 8. Если же $i > k$, то нужный элемент находится в верхней части разбиения. Поскольку нам уже известны k значений, которые меньше i-го в порядке возрастания элемента массива $A[p..r]$ (это элементы подмассива $A[p..q]$), нужный элемент является $(i-k)$-м в порядке возрастания элементом подмассива $A[q+1..r]$, который рекурсивно находится в строке 9. Создается впечатление, что в представленном коде допускаются рекурсивные обращения к подмассивам, содержащим 0 элементов, но в упражнении 9.2-1 предлагается показать, что это невозможно.

Время работы алгоритма RANDOMIZED_SELECT в наихудшем случае равно $\Theta(n^2)$, причем даже для поиска минимума. Дело в том, что фортуна может от нас отвернуться, и разбиение всегда будет производиться относительно наибольшего из оставшихся элементов, а само разбиение занимает время $\Theta(n)$. Однако в среднем алгоритм работает хорошо, и поскольку он рандомизированный, никакие отдельно выбранные входные данные не могут гарантированно привести к наихудшему поведению алгоритма.

Время, необходимое для выполнения алгоритма RANDOMIZED_SELECT с входным массивом $A[p..r]$, состоящим из n элементов, — это случайная величина. Обозначим ее как $T(n)$ и получим верхнюю границу $\mathrm{E}[T(n)]$ следующим образом. Процедура RANDOMIZED_PARTITION с одинаковой вероятностью возвращает в качестве опорного любой элемент. Поэтому для каждого $1 \leqslant k \leqslant n$ в подмассиве $A[p..q]$ с вероятностью $1/n$ содержится k элементов (все они не превышают опорный элемент). Определим для $k = 1, 2, \ldots, n$ индикаторную случайную величину X_k, где

$$X_k = \mathrm{I}\{\text{В подмассиве } A[p..q] \text{ содержится ровно } k \text{ элементов}\}.$$

В предположении, что все элементы различны, имеем:

$$\mathrm{E}[X_k] = 1/n. \tag{9.1}$$

В момент вызова процедуры RANDOMIZED_SELECT и выбора опорного элемента $A[q]$ заранее неизвестно, будет ли получен правильный ответ, после чего работа алгоритма сразу же прекратится, или произойдет рекурсивное обращение к подмассиву $A[p..q-1]$ или подмассиву $A[q+1..r]$. Это будет зависеть от того,

где будет расположен искомый элемент относительно элемента $A[q]$. В предположении монотонного возрастания функции $T(n)$ необходимое для рекурсивного вызова процедуры RANDOMIZED_SELECT время можно ограничить сверху временем, необходимым для вызова этой процедуры с входным массивом максимально возможного размера. Другими словами, для получения верхней границы предполагается, что искомый элемент всегда попадает в ту часть разбиения, где больше элементов. При конкретном вызове процедуры RANDOMIZED_SELECT индикаторная случайная величина X_k принимает значение 1 только для одного значения k, а при всех других k эта величина равна 0. Если $X_k = 1$, то размеры двух подмассивов, к которым может произойти рекурсивное обращение, равны $k-1$ и $n-k$. Таким образом, получаем следующее рекуррентное соотношение:

$$T(n) \leqslant \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) = \\ = \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n).$$

Вычисляя математическое ожидание, получаем:

$$\begin{aligned}
&\mathrm{E}[T(n)] \leqslant \\
&\leqslant \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] = \\
&= \sum_{k=1}^{n} \mathrm{E}[X_k \cdot T(\max(k-1, n-k))] + O(n) = && \text{(в силу линейности математического ожидания)} \\
&= \sum_{k=1}^{n} \mathrm{E}[X_k] \cdot E[T(\max(k-1, n-k))] + O(n) = && \text{(в соответствии с уравнением (B.23))} \\
&= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) && \text{(в соответствии с уравнением (9.1))}
\end{aligned}$$

Применяя уравнение (B.23), мы основывались на независимости случайных величин X_k и $T(\max(k-1, n-k))$. В упражнении 9.2-2 предлагается доказать это предположение.

Рассмотрим выражение $\max(k-1, n-k)$:

$$\max(k-1, n-k) = \begin{cases} k-1, & \text{если } k > \lceil n/2 \rceil, \\ n-k, & \text{если } k \leqslant \lceil n/2 \rceil. \end{cases}$$

Если n четное, то каждое слагаемое от $T(\lceil n/2 \rceil)$ до $T(n-1)$ появляется в сумме ровно дважды. Если же n нечетное, то дважды появляются все слагаемые, кроме

$T(\lfloor n/2 \rfloor)$, которое добавляется лишь один раз. Таким образом, имеем следующее соотношение:

$$\mathrm{E}[T(n)] \leqslant \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}[T(k)] + O(n).$$

Найдем решение этого рекуррентного соотношения методом подстановок. Предположим, что $\mathrm{E}[T(n)] \leqslant cn$ для некоторой константы c, удовлетворяющей начальным условиям рекуррентного соотношения. Кроме того, предположим, что для n, меньших какой-то константы, справедливо $T(n) = O(1)$; эта константа будет найдена позже. Выберем также константу a, такую чтобы функция, соответствующая слагаемому $O(n)$ (и описывающая нерекурсивную составляющую времени работы данного алгоритма), была ограничена сверху величиной an для всех $n > 0$. С помощью этой гипотезы индукции получаем:

$$\begin{aligned}
\mathrm{E}[T(n)] &\leqslant \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an = \\
&= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an = \\
&= \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) + an \leqslant \\
&\leqslant \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + an = \\
&= \frac{2c}{n} \left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) + an = \\
&= \frac{c}{n} \left(\frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an = \\
&= c \left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an \leqslant \\
&\leqslant \frac{3cn}{4} + \frac{c}{2} + an = \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an \right).
\end{aligned}$$

Чтобы завершить доказательство, нужно показать, что для достаточно больших n последнее выражение не превышает величину cn, или (что равносильно) что выполняется неравенство $cn/4 - c/2 - an \geqslant 0$. Добавив к обеим частям этого неравенства $c/2$ и умножив их на n, получим неравенство $n(c/4 - a) \geqslant c/2$. Если константа c выбрана таким образом, что $c/4 - a > 0$, т.е. $c > 4a$, то обе части приведенного выше соотношения можно разделить на $c/4 - a$, что дает нам

следующий результат:

$$n \geqslant \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a}.$$

Таким образом, если предположить, что для всех $n < 2c/(c - 4a)$ справедливо $T(n) = O(1)$, то $\mathrm{E}[T(n)] = O(n)$. Итак, мы приходим к выводу, что для поиска порядковой статистики (в частности, медианы) требуется время, линейно зависящее от количества входных элементов (при этом предполагается, что все элементы различны по величине).

Упражнения

9.2-1. Покажите, что процедура RANDOMIZED_SELECT никогда не вызывается с массивом-параметром, содержащим нулевое количество элементов.

9.2-2. Докажите, что индикаторная случайная величина X_k и величина $T(\max(k-1, n-k))$ независимы.

9.2-3. Напишите итеративную версию процедуры RANDOMIZED_SELECT.

9.2-4. Предположим, что для выбора минимального элемента массива $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1\rangle$ используется процедура RANDOMIZED_SELECT. Опишите последовательность разделений, соответствующих наихудшей производительности этой процедуры.

9.3 Алгоритм выбора с линейным временем работы в наихудшем случае

Рассмотрим алгоритм выбора, время работы которого в наихудшем случае равно $O(n)$. Подобно алгоритму RANDOMIZED_SELECT, алгоритм SELECT находит нужный элемент путем рекурсивного разбиения входного массива. Однако в основе этого алгоритма заложена идея, которая заключается в том, чтобы *гарантировать* хорошее разбиение массива. В алгоритме SELECT используется детерминистическая процедура PARTITION, которая применяется при быстрой сортировке (см. раздел 7.1). Эта процедура модифицирована таким образом, чтобы одним из ее входных параметров был элемент, относительно которого производится разбиение.

Для определения во входном массиве (содержащем более одного элемента) i-го в порядке возрастания элемента, в алгоритме SELECT выполняются следующие шаги (если $n = 1$, то процедура SELECT просто возвращает единственное входное значение.)

1. Все n элементов входного массива разбиваются на $\lfloor n/5 \rfloor$ групп по 5 элементов в каждой и одну группу, содержащую оставшиеся $n \bmod 5$ элементов (впрочем, эта группа может оказаться пустой).

2. Сначала по методу вставок сортируется каждая из $\lceil n/5 \rceil$ групп (содержащих не более 5 элементов), а затем в каждом отсортированном списке, состоящем из элементов групп, выбирается медиана.
3. Путем рекурсивного использования процедуры SELECT определяется медиана x множества из $\lceil n/5 \rceil$ медиан, найденных на втором шаге. (Если этих медиан окажется четное количество, то, согласно принятому соглашению, переменной x будет присвоено значение нижней медианы.)
4. С помощью модифицированной версии процедуры PARTITION входной массив делится относительно медианы медиан x. Пусть число k на единицу превышает количество элементов, попавших в нижнюю часть разбиения. Тогда $x - k$-й в порядке возрастания элемент, и в верхнюю часть разбиения попадает $n - k$ элементов.
5. Если $i = k$, то возвращается значение x. В противном случае процедура SELECT вызывается рекурсивно, и с ее помощью выполняется поиск i-го в порядке возрастания элемента в нижней части, если $i < k$, или в верхней части, если $i > k$.

Чтобы проанализировать время работы процедуры SELECT, сначала определим нижнюю границу для количества элементов, превышающих по величине опорный элемент x. Разобраться в этой бухгалтерии поможет рис. 9.1. На этом рисунке n элементов представлены в виде маленьких кружков, а каждая группа — отдельной колонкой. Медианы групп обозначены белыми кружками, а медиана медиан — белым кружком, рядом с которым стоит символ x. (При определении медианы четного количества элементов используется нижняя медиана.) Стрелки проведены в направлении от бóльших элементов к меньшим. Из рисунка видно, что в каждой полной группе из 5 элементов, расположенной справа от x, содержится по 3 элемента, превышающих по величине x, а в каждой полной группе из 5 элементов, расположенной слева от x, содержится по 3 элемента, меньших по величине, чем x. Элементы, которые превышают x, выделены серым фоном. В общем случае как минимум половина медиан, найденных на втором шаге, больше или равны медианы медиан x[1]. Таким образом, как минимум $\lceil n/5 \rceil$ групп содержат по 3 элемента, превышающих величину x, за исключением группы, в которой меньше пяти элементов (такая группа существует, если n не делится на 5 нацело), и еще одной группы, содержащей сам элемент x. Не учитывая эти две группы, приходим к выводу, что количество элементов, величина которых превышает x, удовлетворяет следующему неравенству:

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil - 2\right) \geqslant \frac{3n}{10} - 6.$$

[1]Согласно предположению о том, что все числа различаются, каждая из этих медиан больше или меньше x, за исключением самой медианы x.

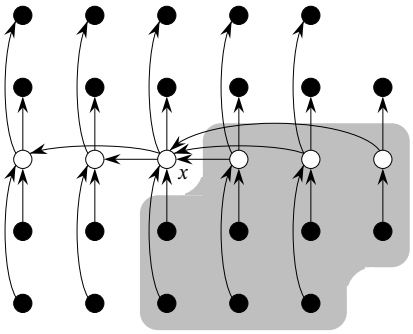


Рис. 9.1. Анализ алгоритма SELECT

Аналогично, имеется не менее $3n/10 - 6$ элементов, величина которых меньше x. Таким образом, процедура SELECT рекурсивно вызывается на пятом шаге не менее чем для $7n/10 + 6$ элементов.

Теперь можно сформулировать рекуррентное соотношение для времени работы алгоритма SELECT (обозначим его через $T(n)$) в наихудшем случае. Для выполнения первого, второго и четвертого шагов требуется время $O(n)$ (шаг 2 состоит из $O(n)$ вызовов сортировки вставкой для множеств размером $O(1)$). Выполнение третьего шага длится в течение времени $T(\lfloor n/5 \rfloor)$, а выполнение пятого шага — в течение времени, не превышающего величину $T(7n/10 + 6)$ (предполагается, что функция T монотонно возрастает). Сделаем на первый взгляд необоснованное предположение, что для обработки любого входного массива, количество элементов которого меньше 140, требуется время $O(1)$ (вскоре нам раскроется магия константы 140). Получаем рекуррентное соотношение:

$$T(n) \leqslant \begin{cases} O(1) & \text{если } n < 140, \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{если } n \geqslant 140. \end{cases}$$

Покажем методом подстановки, что время работы, описываемое этим соотношением, линейно зависит от количества входных элементов. Точнее говоря, покажем, что для некоторой достаточно большой константы c и для всех $n > 0$ выполняется неравенство $T(n) \leqslant cn$. Начнем с предположения, что это неравенство выполняется для некоторой достаточно большой константы c и для всех $n < 140$; это предположение действительно выполняется, если константа c выбрана достаточно большой. Кроме того, выберем константу a таким образом, чтобы функция, соответствующая представленному выше слагаемому $O(n)$ (которое описывает нерекурсивную составляющую времени работы алгоритма) для всех $n > 0$ была ограничена сверху величиной an. Подставив эту гипотезу индукции в правую

часть рекуррентного соотношения, получим:

$$\begin{aligned} T(n) &\leqslant c\lceil n/5\rceil + c(7n/10+6) + an \leqslant \\ &\leqslant cn/5 + c + 7cn/10 + 6c + an = \\ &= 9cn/10 + 7c + an = \\ &= cn + (-cn/10 + 7c + an). \end{aligned}$$

Это выражение не превышает величину cn, если выполняется неравенство

$$-cn/10 + 7c + an \leqslant 0. \tag{9.2}$$

При $n > 70$ неравенство (9.2) эквивалентно неравенству $c \geqslant 10a(n/(n-70))$. Поскольку мы предположили, что $n \geqslant 140$, то $n/(n-70) \leqslant 2$, а значит, если выбрать $c \geqslant 20a$, то будет удовлетворяться неравенство (9.2). (Заметим, что в константе 140 нет ничего особенного; вместо нее можно было бы взять любое другое целое число, превышающее 70, после чего соответствующим образом нужно было бы выбрать константу c.) Таким образом, в наихудшем случае время работы алгоритма SELECT линейно зависит от количества входных элементов.

Как и в алгоритмах сортировки сравнением, которые рассматривались в разделе 8.1, в алгоритмах SELECT и RANDOMIZED_SELECT информация о взаимном расположении элементов извлекается только путем их сравнения. Как было доказано в главе 8, в модели сравнений сортировка выполняется в течение времени $\Omega(n \lg n)$, даже в среднем (см. задачу 8-1). В этой же главе был рассмотрен алгоритм сортировки, время работы которого линейно зависит от количества сортируемых элементов, но в нем делаются дополнительные предположения относительно входных данных. Для работы представленных в настоящей главе алгоритмов выбора, время работы которых такое же, напротив, никаких дополнительных предположений не требуется. К этим алгоритмам не применима нижняя граница $\Omega(n \lg n)$, поскольку задача выбора в них решается без сортировки.

Таким образом, время работы рассмотренных выше алгоритмов линейно, поскольку в них не производится сортировка. Линейная зависимость от количества входных элементов не является результатом каких-то дополнительных предположений о входных данных, как в случае алгоритмов сортировки, описанных в главе 8. Для сортировки сравнением даже в среднем требуется время $\Omega(n \lg n)$, так что предложенный во введении к этой главе метод сортировки и индексирования асимптотически неэффективен.

Упражнения

9.3-1. В алгоритме SELECT все входные элементы делятся на группы по 5 элементов в каждой. Было бы время работы этого алгоритма линейным, если

бы входной массив делился на группы по 7 элементов в каждой? Докажите, что время работы алгоритма не будет линейным, если в каждой группе будет по 3 элемента.

9.3-2. Проанализируйте алгоритм SELECT и покажите, что при $n \geqslant 140$ как минимум $\lceil n/4 \rceil$ элементов превышают по величине медиану медиан x, и как минимум $\lceil n/4 \rceil$ элементов меньше x.

9.3-3. Покажите, каким образом можно выполнить быструю сортировку за время $O(n \lg n)$ в наихудшем случае, считая, что все элементы различны.

⋆ 9.3-4. Предположим, что в алгоритме, предназначенном для поиска среди n элементов i-го в порядке возрастания элемента, применяется только операция сравнения. Покажите, что с помощью этого алгоритма можно также найти $i-1$ наименьших элементов и $n-i$ наибольших элементов, не выполняя никаких дополнительных сравнений.

9.3-5. Предположим, что у нас имеется подпрограмма поиска медиан, которая представляет собой "черный ящик" и время работы которой линейно зависит от количества входных элементов. Приведите пример алгоритма с линейным временем работы, с помощью которого задача выбора решалась бы для произвольной порядковой статистики.

9.3-6. По определению k-ми ***квантилями*** (quantiles) n-элементного множества называются $k-1$ порядковых статистик, разбивающих это отсортированное множество на k одинаковых подмножеств (с точностью до одного элемента). Сформулируйте алгоритм, который бы выводил список k-х квантилей множества в течение времени $O(n \lg k)$.

9.3-7. Опишите алгоритм, который для заданного множества S, состоящего из n различных чисел, и положительной целой константы $k \leqslant n$ определял бы k ближайших соседей медианы множества S, принадлежащих к этому множеству. Время работы этого алгоритма должно быть равно $O(n)$.

9.3-8. Пусть $X[1..n]$ и $Y[1..n]$ — два массива, каждый из которых содержит по n элементов, расположенных в отсортированном порядке. Разработайте алгоритм, в котором поиск медианы всех $2n$ элементов, содержащихся в массивах X и Y, выполнялся бы за время $O(\lg n)$.

9.3-9. Профессор работает консультантом в нефтяной компании, которая планирует провести магистральный трубопровод от восточного до западного края нефтяного месторождения с n скважинами. От каждой скважины к магистральному трубопроводу кратчайшим путем проведены рукава (рис. 9.2). Каким образом профессор может выбрать оптимальное расположение трубопровода (т.е. такое, при котором общая длина всех рукавов была бы минимальной) по заданным координатам скважин (x, y)? Пока-

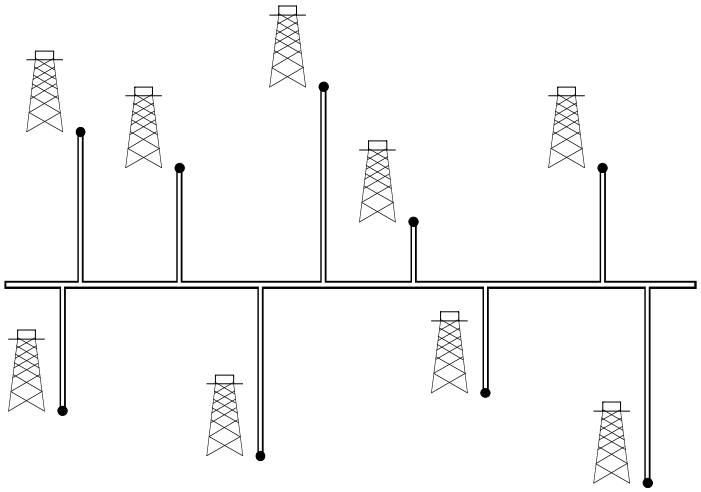


Рис. 9.2. Определение положения нефтепровода, при котором общая длина поперечных рукавов будет минимальной

жите, что это можно сделать в течение времени, линейно зависящего от количества скважин.

Задачи

9-1. **Наибольшие i элементов в порядке сортировки**

Пусть имеется n-элементное множество, в котором с помощью алгоритма, основанного на сравнениях, нужно найти i наибольших элементов, расположенных в порядке сортировки. Сформулируйте алгоритм, в котором реализовывался бы каждый из перечисленных ниже методов с наилучшим возможным асимптотическим временем работы в наихудшем случае. Проанализируйте зависимость времени работы этих алгоритмов от n и i.

а) Все числа сортируются и выводятся i наибольших.

б) Создайте из чисел невозрастающую приоритетную очередь и i раз вызовите процедуру EXTRACT_MAX.

в) Найдите с помощью алгоритма порядковой статистики i-й по порядку наибольший элемент, произведите разбиение относительно него и выполните сортировку i наибольших чисел.

9-2. **Взвешенная медиана**

Пусть имеется n различных элементов $x_1, x_2, \ldots, x_n$, для которых заданы положительные веса $w_1, w_2, \ldots, w_n$, удовлетворяющие соотношению

$\sum_{i=1}^{n} w_i = 1$. ***Взвешенной (нижней) медианой*** (weighted (lower) median) называется элемент x_k, удовлетворяющий неравенствам

$$\sum_{x_i<x_k} w_i < \frac{1}{2}$$

и

$$\sum_{x_i>x_k} w_i \leqslant \frac{1}{2}.$$

а) Докажите, что обычная медиана элементов $x_1, x_2, \ldots, x_n$ равна взвешенной медиане x_i, веса которой для всех $i = 1, 2, \ldots, n$ равны $w_i = 1/n$.

б) Покажите, как вычислить взвешенную медиану n элементов с помощью сортировки, чтобы время вычисления в наихудшем случае было равно $O(n \lg n)$.

в) Покажите, как с помощью алгоритма, в котором производится поиск медианы за линейное время (подобного описанному в разделе 9.3 алгоритму SELECT), вычислить взвешенную медиану в течение времени, которое равно $\Theta(n)$ в наихудшем случае.

Задача об оптимальном размещении почтового отделения формулируется так. Имеется n точек $p_1, p_2, \ldots, p_n$, которым отвечают веса $w_1, w_2, \ldots, w_n$. Нужно найти точку p (это не обязательно должна быть одна из входных точек), в которой минимизируется сумма $\sum_{i=1}^{n} w_i d(p, p_i)$, где $d(a, b)$ — расстояние между точками a и b.

г) Докажите, что взвешенная медиана — это наилучшее решение одномерной задачи об оптимальном размещении почтового отделения. В этой задаче точки представляют собой действительные числа, а расстояние между точками a и b определяется как $d(a, b) = |a - b|$.

д) Найдите наилучшее решение двумерной задачи об оптимальном размещении почтового отделения. В этой задаче точки задаются парами координат (x, y), а расстояние между точками $a = (x_1, y_1)$ и $b = (x_2, y_2)$ представляет собой ***манхэттенское расстояние*** (Manhattan distance), которое определяется соотношением $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

9-3. Малые порядковые статистики

Ранее было показано, что количество сравнений $T(n)$, которые производятся в алгоритме SELECT в наихудшем случае в ходе выбора i-й порядковой статистики из n элементов, удовлетворяет соотношению $T(n) = \Theta(n)$. Однако при этом скрытая в Θ-обозначении константа имеет

довольно большую величину. Если i намного меньше n, то можно реализовать другую процедуру, в которой в качестве подпрограммы используется процедура SELECT, но в которой в наихудшем случае производится меньшее количество сравнений.

а) Опишите алгоритм, в котором для поиска i-го в порядке возрастания среди n элементов требуется $U_i(n)$ сравнений, где

$$U_i(n) = \begin{cases} T(n) & \text{если } i \geqslant n/2, \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) & \text{в противном случае.} \end{cases}$$

(*Указание*: начните с $\lfloor n/2 \rfloor$ непересекающихся попарных сравнений и организуйте рекурсивную процедуру для множества, содержащего меньшие элементы из каждой пары.)

б) Покажите, что если $i < n/2$, то $U_i(n) = n + O(T(2i)\lg(n/i))$.

в) Покажите, что если i — константа, меньшая, чем $n/2$, то $U_i(n) = n + O(\lg n)$.

г) Покажите, что если $i = n/k$ для $k \geqslant 2$, то $U_i(n) = n + O(T(2n/k) \times \lg k)$.

Заключительные замечания

Алгоритм поиска медиан, время работы которого в наихудшем случае линейно зависит от количества входных элементов, был разработан Блюмом (Blum), Флойдом (Floyd), Праттом (Pratt), Ривестом (Rivest) и Таржаном (Tarjan) [43]. Версия этого алгоритма с пониженным средним временем работы появилась благодаря Хоару (Hoare) [146]. Флойд и Ривест [92] создали улучшенную версию этого алгоритма, работающую в среднем еще производительнее, в которой разбиение производится относительно элемента, рекурсивно выбираемого из небольшого подмножества.

До сих пор точно неизвестно, сколько именно сравнений необходимо для поиска медианы. Нижняя граница, равная $2n$ сравнениям, была найдена Бентом (Bent) и Джоном (John) [38], а верхняя граница, равная $3n$ сравнениям, —Шёнхагом (Schönhage), Патерсоном (Paterson) и Пиппенджером (Pippenger) [265]. Дор (Dor) и Цвик (Zwick) [79] улучшили обе эти границы; их верхняя граница немного меньше величины $2.95n$, а нижняя — несколько превышает величину $2n$. В своей работе [239] Патерсон описал все изложенные здесь результаты, а также результаты других работ, посвященных этой теме.

ЧАСТЬ III

Структуры данных

Введение

Множество — это фундаментальное понятие как в математике, так и в теории вычислительных машин. Тогда как математические множества остаются неизменными, множества, которые обрабатываются в ходе выполнения алгоритмов, могут с течением времени разрастаться, уменьшаться или подвергаться другим изменениям. Назовем такие множества ***динамическими*** (dynamic). В пяти последующих главах описываются некоторые основные методы представления конечных динамических множеств и манипуляции с ними на компьютере.

В некоторых алгоритмах, предназначенных для обработки множеств, требуется выполнять операции нескольких различных видов. Например, набор операций, используемых во многих алгоритмах, ограничивается возможностью вставлять элементы в множество, удалять их, а также проверять, принадлежит ли множеству тот или иной элемент. Динамическое множество, поддерживающее перечисленные операции, называется ***словарем*** (dictionary). В других множествах могут потребоваться более сложные операции. Например, в неубывающих очередях с приоритетами, с которыми мы ознакомились в главе 6 в контексте пирамидальной структуры данных, поддерживаются операции вставки элемента и извлечение минимального элемента. Оптимальный способ реализации динамического множества зависит от того, какие операции должны им поддерживаться.

Элементы динамического множества

В типичных реализациях динамического множества каждый его элемент представлен некоторым объектом; если в нашем распоряжении имеется указатель на объект, то можно проверять и изменять значения его полей. (В разделе 10.3 обсуждается реализация объектов и указателей в средах, где они не являются базовыми типами данных.) В динамических множествах некоторых типов предполагается, что одно из полей объекта идентифицируется как ***ключевое*** (key field). Если все ключи различны, то динамическое множество представимо в виде набора ключевых значений. Иногда объекты содержат ***сопутствующие данные*** (satellite data), которые находятся в других его полях, но не используются реализацией множества. Кроме того, объект может содержать поля, доступные для манипуляции во время выполнения операций над множеством; иногда в этих полях хранятся данные или указатели на другие объекты множества.

В некоторых динамических множествах предполагается, что их ключи являются членами полностью упорядоченного множества, например, множества действительных чисел или множества всех слов, которые могут быть расположены в алфавитном порядке. (Полностью упорядоченные множества удовлетворяют свойству трихотомии, определение которого дано в разделе 3.1.) Полное упоря-

дочение, например, позволяет определить минимальный элемент множества или говорить о ближайшем элементе множества, превышающем заданный.

Операции в динамических множествах

Операции динамического множества можно разбить на две категории: ***запросы*** (queries), которые просто возвращают информацию о множестве, и ***модифицирующие операции*** (modifying operations), изменяющие множество. Ниже приведен список типичных операций. В каждом конкретном приложении требуется, чтобы были реализованы лишь некоторые из них.

SEARCH(S, k)

Запрос, который возвращает указатель на элемент x заданного множества S, для которого $key\,[x] = k$, или значение NIL, если в множестве S такой элемент отсутствует.

INSERT(S, x)

Модифицирующая операция, которая пополняет заданное множество S одним элементом, на который указывает x. Обычно предполагается, что выполнена предварительная инициализация всех полей элемента x, необходимых для реализации множества.

DELETE(S, x)

Модифицирующая операция, удаляющая из заданного множества S элемент, на который указывает x. (Обратите внимание, что в этой операции используется указатель на элемент, а не его ключевое значение.)

MINIMUM(S)

Запрос к полностью упорядоченному множеству S, который возвращает указатель на элемент этого множества с наименьшим ключом.

MAXIMUM(S)

Запрос к полностью упорядоченному множеству S, который возвращает указатель на элемент этого множества с наибольшим ключом.

SUCCESSOR(S, x)

Запрос к полностью упорядоченному множеству S, который возвращает указатель на элемент множества S, ключ которого является ближайшим соседом ключа элемента x и превышает его. Если же x — максимальный элемент множества S, то возвращается значение NIL.

PREDECESSOR(S, x)

Запрос к полностью упорядоченному множеству S, который возвращает указатель на элемент множества S, ключ которого является ближайшим меньшим по значению соседом ключа элемента x. Если же x — минимальный элемент множества S, то возвращается значение NIL.

Запросы SUCCESSOR и PREDECESSOR часто обобщаются на множества, в которых не все ключи различаются. Для множества, состоящего из n элементов, обычно принимается допущение, что вызов операции MINIMUM, после которой $n - 1$ раз вызывается операция SUCCESSOR, позволяет пронумеровать элементы множества в порядке сортировки.

Время, необходимое для выполнения операций множества, обычно измеряется в единицах, связанных с размером множества, который указывается в качестве одного из аргументов. Например, в главе 13 описывается структура данных, способная поддерживать все перечисленные выше операции, причем время их выполнения на множестве размера n выражается как $O(\lg n)$.

Обзор третьей части

В главах 10–14 описываются несколько структур данных, с помощью которых можно реализовать динамические множества. Многие из этих структур данных будут использоваться впоследствии при разработке эффективных алгоритмов, позволяющих решать разнообразные задачи. Еще одна важная структура данных, пирамида, уже была рассмотрена в главе 6.

В главе 10 представлены основные приемы работы с такими простыми структурами данных, как стеки, очереди, связанные списки и корневые деревья. В ней также показано, как можно реализовать в средах программирования объекты и указатели, в которых они не поддерживаются в качестве примитивов. Бóльшая часть материала этой главы должна быть знакома всем, кто освоил вводный курс программирования.

Глава 11 знакомит читателя с хеш-таблицами, поддерживающими такие присущие словарям операции, как INSERT, DELETE и SEARCH. В наихудшем случае операция поиска в хеш-таблицах выполняется в течение времени $\Theta(n)$, однако математическое ожидание времени выполнения подобных операций равно $O(1)$. Анализ хеширования основывается на теории вероятности, однако для понимания большей части материала этой главы не требуются предварительные знания в этой области.

Описанные в главе 12 бинарные деревья поиска поддерживают все перечисленные выше операции динамических множеств. В наихудшем случае для выполнения каждой такой операции на n-элементном множестве требуется время $\Theta(n)$, однако при случайном построении бинарных деревьев поиска математическое ожидание времени работы каждой операции равно $O(\lg n)$. Бинарные деревья поиска служат основой для многих других структур данных.

С красно-черными деревьями, представляющими собой разновидность бинарных деревьев поиска, нас знакомит глава 13. В отличие от обычных бинарных деревьев поиска, красно-черные деревья всегда работают хорошо: в наихудшем случае операции над ними выполняются в течение времени $O(\lg n)$. Красно-

черное дерево представляет собой сбалансированное дерево поиска; в главе 18 представлено сбалансированное дерево поиска другого вида, получившее название B-дерево. Несмотря на то, что механизмы работы красно-черных деревьев несколько запутаны, из этой главы можно получить подробное представление об их свойствах без подробного изучения этих механизмов. Тем не менее, будет довольно поучительно, если вы внимательно ознакомитесь со всем представленным в главе материалом.

В главе 14 показано, как расширить красно-черные деревья для поддержки операций, отличных от перечисленных выше базовых. Сначала мы рассмотрим расширение красно-черных деревьев, обеспечивающее динамическую поддержку порядковых статистик для множества ключей, а затем — для поддержки интервалов действительных чисел.

ГЛАВА 10

Элементарные структуры данных

В этой главе рассматривается представление динамических множеств простыми структурами данных, в которых используются указатели. Несмотря на то, что с помощью указателей можно сформировать многие сложные структуры данных, здесь будут представлены лишь простейшие из них: стеки, очереди, связанные списки и деревья. Мы также рассмотрим метод, позволяющий синтезировать из массивов объекты и указатели.

10.1 Стеки и очереди

Стеки и очереди — это динамические множества, элементы из которых удаляются с помощью предварительно определенной операции DELETE. Первым из ***стека*** (stack) удаляется элемент, который был помещен туда последним: в стеке реализуется стратегия "***последним вошел — первым вышел***" (last-in, first-out — LIFO). Аналогично, в ***очереди*** (queue) всегда удаляется элемент, который содержится в множестве дольше других: в очереди реализуется стратегия "***первым вошел — первым вышел***" (first-in, first-out — FIFO). Существует несколько эффективных способов реализации стеков и очередей в компьютере. В данном разделе будет показано, как реализовать обе эти структуры данных с помощью обычного массива.

Стеки

Операция вставки применительно к стекам часто называется PUSH (запись в стек), а операция удаления — POP (снятие со стека).

Как видно из рис. 10.1, стек, способный вместить не более n элементов, можно реализовать с помощью массива $S[1..n]$. Этот массив обладает атрибутом $top[S]$,

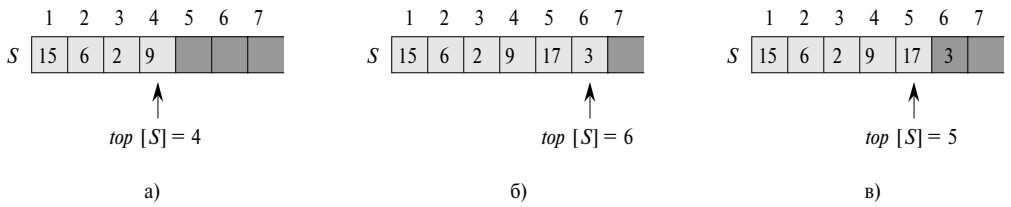


Рис. 10.1. Реализация стека S в виде массива

представляющим собой индекс последнего помещенного в стек элемента. Стек состоит из элементов $S[1..top[S]]$, где $S[1]$ — элемент на дне стека, а $S[top[S]]$ — элемент на его вершине.

Если $top[S] = 0$, то стек не содержит ни одного элемента и является ***пустым*** (empty). Протестировать стек на наличие в нем элементов можно с помощью операции-запроса STACK_EMPTY. Если элемент снимается с пустого стека, говорят, что он ***опустошается*** (underflow), что обычно приводит к ошибке. Если значение $top[S]$ больше n, то стек ***переполняется*** (overflow). (В представленном ниже псевдокоде возможное переполнение во внимание не принимается.)

Каждую операцию над стеком можно легко реализовать несколькими строками кода:

```
STACK_EMPTY(S)
1  if top[S] = 0
2      then return TRUE
3      else return FALSE

PUSH(S, x)
1  top[S] ← top[S] + 1
2  S[top[S]] ← x

POP(S)
1  if STACK_EMPTY(S)
2      then error "underflow"
3      else top[S] ← top[S] − 1
4           return S[top[S] + 1]
```

Из рис. 10.1 видно, какое воздействие на стек оказывают модифицирующие операции PUSH и POP. Элементы стека находятся только в тех позициях массива, которые отмечены светло-серым цветом. В части *а* рисунка изображен стек S, состоящий из 4 элементов. На вершине стека находится элемент 9. В части *б* представлен этот же стек после вызова процедур $\text{PUSH}(S, 17)$ и $\text{PUSH}(S, 3)$, а в части *в* — после вызова процедуры $\text{POP}(S)$, которая возвращает помещенное в стек последним значение 3. Несмотря на то, что элемент 3 все еще показан в массиве,

он больше не принадлежит стеку; теперь на вершине стека — 17. Любая из трех описанных операций со стеком выполняется в течение времени $O(1)$.

Очереди

Применительно к очередям операция вставки называется ENQUEUE (поместить в очередь), а операция удаления — DEQUEUE (вывести из очереди). Подобно стековой операции POP, операция DEQUEUE не требует передачи элемента массива в виде аргумента. Благодаря свойству FIFO очередь подобна, например, живой очереди к врачу в поликлинике. У нее имеется ***голова*** (head) и ***хвост*** (tail). Когда элемент ставится в очередь, он занимает место в ее хвосте, точно так же, как человек занимает очередь последним, чтобы попасть на прием к врачу. Из очереди всегда выводится элемент, который находится в ее головной части аналогично тому, как в кабинет врача всегда заходит больной, который ждал дольше всех.

На рис. 10.2 показан один из способов, который позволяет с помощью массива $Q[1..n]$ реализовать очередь, состоящую не более чем из $n-1$ элементов. Эта очередь обладает атрибутом $head[Q]$, который является индексом головного элемента или указателем на него; атрибут $tail[Q]$ индексирует местоположение, куда будет добавлен новый элемент. Элементы очереди расположены в ячейках $head[Q]$, $head[Q]+1, \ldots, tail[Q]-1$, которые циклически замкнуты в том смысле, что ячейка 1 следует сразу же после ячейки n в циклическом порядке. При выполнении условия $head[Q] = tail[Q]$ очередь пуста. Изначально выполняется соотношение $head[Q] = tail[Q] = 1$. Если очередь пустая, то при попытке удалить из нее элемент происходит ошибка опустошения. Если $head[Q] = tail[Q]+1$, то очередь заполнена, и попытка добавить в нее элемент приводит к ее переполнению.

В наших процедурах ENQUEUE и DEQUEUE проверка ошибок опустошения и переполнения не производится. (В упражнении 10.1-4 предлагается добавить в процедуры соответствующий код.)

```
ENQUEUE(Q, x)
1  Q[tail[Q]] ← x
2  if tail[Q] = length[Q]
3       then tail[Q] ← 1
4       else  tail[Q] ← tail[Q] + 1
```

```
DEQUEUE(Q)
1  x ← Q[head[Q]]
2  if head[Q] = length[Q]
3       then head[Q] ← 1
4       else  head[Q] ← head[Q] + 1
5  return x
```

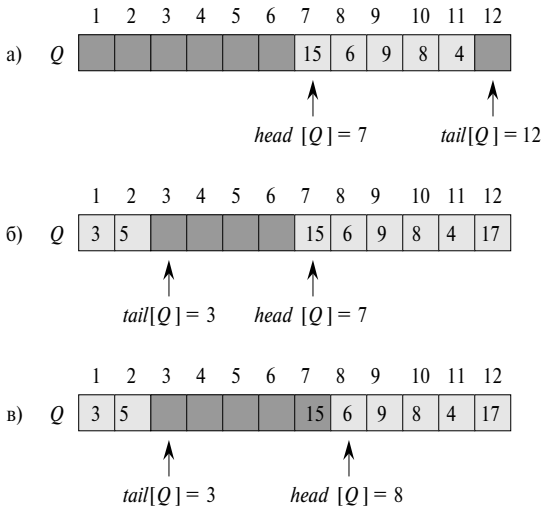


Рис. 10.2. Очередь, реализованная с помощью массива $Q\,[1..12]$

На рис. 10.2 показана работа процедур ENQUEUE и DEQUEUE. Элементы очереди содержатся только в светло-серых ячейках. В части *а* рисунка изображена очередь, состоящая из пяти элементов, расположенных в ячейках $Q\,[7..11]$. В части *б* показана эта же очередь после вызова процедур ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$ и ENQUEUE$(Q, 5)$, а в части *в* — конфигурация очереди после вызова процедуры DEQUEUE(Q), возвращающей значение ключа 15, которое до этого находилось в голове очереди. Значение ключа новой головы очереди равно 6. Каждая операция выполняется в течение времени $O\,(1)$.

Упражнения

10.1-1. Используя в качестве модели рис. 10.1, проиллюстрируйте результат воздействия на изначально пустой стек S, хранящийся в массиве $S\,[1..6]$, операций PUSH$(S, 4)$, PUSH$(S, 1)$, PUSH$(S, 3)$, POP(S), PUSH$(S, 8)$ и POP(S).

10.1-2. Объясните, как с помощью одного массива $A\,[1..n]$ можно реализовать два стека таким образом, чтобы ни один из них не переполнялся, пока суммарное количество элементов в обоих стеках не достигнет n. Операции PUSH и POP должны выполняться в течение времени $O\,(1)$.

10.1-3. Используя в качестве модели рис. 10.2, проиллюстрируйте результат воздействия на изначально пустую очередь Q, хранящуюся в массиве $Q\,[1..6]$, операций ENQUEUE$(Q, 4)$, ENQUEUE$(Q, 1)$, ENQUEUE$(Q, 3)$, DEQUEUE(Q), ENQUEUE$(Q, 8)$ и DEQUEUE(Q).

10.1-4. Перепишите код процедур ENQUEUE и DEQUEUE таким образом, чтобы они обнаруживали ошибки опустошения и переполнения.

10.1-5. При работе со стеком элементы можно добавлять в него и извлекать из него только с одного конца. Очередь позволяет добавлять элементы с одного конца, а извлекать — с другого. ***Очередь с двусторонним доступом***, или ***дек*** (deque), предоставляет возможность производить вставку и удаление с обоих концов. Напишите четыре процедуры, выполняющиеся в течение времени $O(1)$ и позволяющие вставлять и удалять элементы с обоих концов дека, реализованного с помощью массива.

10.1-6. Покажите, как реализовать очередь с помощью двух стеков. Проанализируйте время работы операций, которые выполняются с ее элементами.

10.1-7. Покажите, как реализовать стек с помощью двух очередей. Проанализируйте время работы операций, которые выполняются с его элементами.

10.2 Связанные списки

Связанный список (linked list) — это структура данных, в которой объекты расположены в линейном порядке. Однако, в отличие от массива, в котором этот порядок определяется индексами, порядок в связанном списке определяется указателями на каждый объект. Связанные списки обеспечивают простое и гибкое представление динамических множеств и поддерживают все операции (возможно, недостаточно эффективно), перечисленные на стр. 257.

Как показано на рис. 10.3, каждый элемент ***дважды связанного списка*** (doubly linked list) L — это объект с одним полем ключа key и двумя полями-указателями: $next$ (следующий) и $prev$ (предыдущий). Этот объект может также содержать другие сопутствующие данные. Для заданного элемента списка x указатель $next[x]$ указывает на следующий элемент связанного списка, а указатель $prev[x]$ — на предыдущий. Если $prev[x] = \text{NIL}$, у элемента x нет предшественника, и, следовательно, он является первым, т.е. ***головным*** в списке. Если $next[x] = \text{NIL}$, то у элемента x нет последующего, поэтому он является последним, т.е. ***хвостовым*** в списке. Атрибут $head[L]$ указывает на первый элемент списка. Если $head[L] = \text{NIL}$, то список пуст.

Списки могут быть разных видов. Список может быть однократно или дважды связанным, отсортированным или неотсортированным, кольцевым или некольцевым. Если список ***однократно связанный*** (***однонаправленный***) (singly linked), то указатель $prev$ в его элементах отсутствует. Если список ***отсортирован*** (sorted), то его линейный порядок соответствует линейному порядку его ключей; в этом случае минимальный элемент находится в голове списка, а максимальный — в его хвосте. Если же список не отсортирован, то его элементы могут располагаться в произвольном порядке. Если ***список кольцевой*** (circular list), то указатель

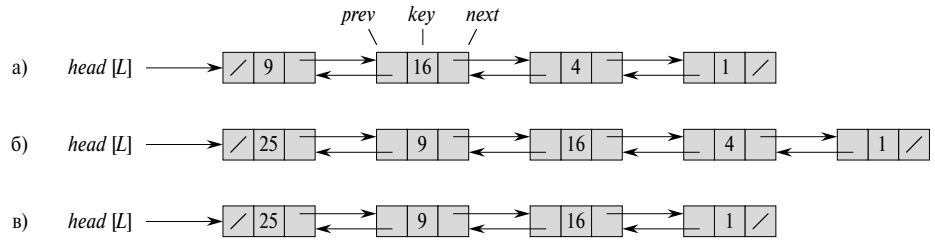


Рис. 10.3. Вставка в дважды связанный список и удаление из него

$prev$ его головного элемента указывает на его хвост, а указатель $next$ хвостового элемента — на головной элемент. Такой список можно рассматривать как замкнутый в виде кольца набор элементов. В оставшейся части раздела предполагается, что списки, с которыми нам придется работать, — неотсортированные и дважды связанные.

Поиск в связанном списке

Процедура LIST_SEARCH(L, k) позволяет найти в списке L первый элемент с ключом k путем линейного поиска, и возвращает указатель на найденный элемент. Если элемент с ключом k в списке отсутствует, возвращается значение NIL. Процедура LIST_SEARCH$(L, 4)$, вызванная для связанного списка, изображенного на рис. 10.3*а*, возвращает указатель на третий элемент, а процедура LIST_SEARCH$(L, 7)$ — значение NIL:

```
LIST_SEARCH(L, k)
1  x ← head[L]
2  while x ≠ NIL и key[x] ≠ k
3        do x ← next[x]
4  return x
```

Поиск с помощью процедуры LIST_SEARCH в списке, состоящем из n элементов, в наихудшем случае выполняется в течение времени $\Theta(n)$, поскольку может понадобиться просмотреть весь список.

Вставка в связанный список

Если имеется элемент x, полю key которого предварительно присвоено значение, то процедура LIST_INSERT вставляет элемент x в переднюю часть списка (рис. 10.3*б*):

```
LIST_INSERT(L, x)
1  next[x] ← head[L]
2  if head[L] ≠ NIL
3      then prev[head[L]] ← x
4  head[L] ← x
5  prev[x] ← NIL
```

Время работы процедуры LIST_INSERT равно $O(1)$.

Удаление из связанного списка

Представленная ниже процедура LIST_DELETE удаляет элемент x из связанного списка L. В процедуру необходимо передать указатель на элемент x, после чего она удаляет x из списка путем обновления указателей. Чтобы удалить элемент с заданным ключом, необходимо сначала вызвать процедуру LIST_SEARCH для получения указателя на элемент:

```
LIST_DELETE(L, x)
1  if prev[x] ≠ NIL
2      then next[prev[x]] ← next[x]
3      else  head[L] ← next[x]
4  if next[x] ≠ NIL
5      then prev[next[x]] ← prev[x]
```

Удаление элемента из связанного списка проиллюстрировано на рис. 10.3*в*. Время работы процедуры LIST_DELETE равно $O(1)$, но если нужно удалить элемент с заданным ключом, то в наихудшем случае понадобится время $\Theta(n)$, поскольку сначала необходимо вызвать процедуру LIST_SEARCH.

Ограничители

Код процедуры LIST_DELETE мог бы быть проще, если бы можно было игнорировать граничные условия в голове и хвосте списка:

```
LIST_DELETE′(L, x)
1  next[prev[x]] ← next[x]
2  prev[next[x]] ← prev[x]
```

Ограничитель (sentinel) — это фиктивный объект, упрощающий учет граничных условий. Например, предположим, что в списке L предусмотрен объект $nil[L]$, представляющий значение NIL, но при этом содержащий все поля, которые имеются у других элементов. Когда в коде происходит обращение к значению NIL, оно заменяется обращением к ограничителю $nil[L]$. Из рис. 10.4 видно, что наличие ограничителя преобразует обычный дважды связанный список в ***циклический***

дважды связанный список с ограничителем (circular, doubly linked list with a sentinel). В таком списке ограничитель $nil\,[L]$ расположен между головой и хвостом. Поле $next\,[nil[L]]$ указывает на голову списка, а поле $prev\,[nil[L]]$ — на его хвост. Аналогично, поле $next$ хвостового элемента и поле $prev$ головного элемента указывают на элемент $nil\,[L]$. Поскольку поле $next\,[nil[L]]$ указывает на голову списка, можно упразднить атрибут $head\,[L]$, заменив ссылки на него ссылками на поле $next\,[nil[L]]$. Пустой список содержит только ограничитель, поскольку и полю $next\,[nil[L]]$, и полю $prev\,[nil[L]]$ можно присвоить значение $nil\,[L]$.

Код процедуры LIST_SEARCH остается тем же, что и раньше, однако ссылки на значения NIL и $head\,[L]$ заменяются так, как описано выше:

```
LIST_SEARCH′(L, k)
1  x ← next[nil[L]]
2  while x ≠ nil[L] и key[x] ≠ k
3        do x ← next[x]
4  return x
```

Удаление элемента из списка производится с помощью уже описанной процедуры LIST_DELETE′, состоящей всего из двух строк. Для вставки элемента в список используется приведенная ниже процедура:

```
LIST_INSERT′(L, x)
1  next[x] ← next[nil[L]]
2  prev[next[nil[L]]] ← x
3  next[nil[L]] ← x
4  prev[x] ← nil[L]
```

Действие процедур LIST_INSERT′ и LIST_DELETE′ проиллюстрировано на рис. 10.4. В части *а* этого рисунка приведен пустой список. В части *б* изображен связанный список, уже знакомый нам по рис. 10.3*а*, в голове которого находится ключ 9, а в хвосте — ключ 1. В части *в* изображен этот же список после выполнения процедуры LIST_INSERT′(L, x), где $key\,[x] = 25$. Новый объект становится в голову списка. В части *г* рассматриваемый список представлен после удаления в нем объекта с ключом 1. Новым хвостовым объектом становится объект с ключом 4.

Применение ограничителей редко приводит к улучшению асимптотической зависимости времени обработки структур данных, однако оно может уменьшить величину постоянных множителей. Благодаря использованию ограничителей в циклах обычно не столько увеличивается скорость работы кода, сколько повышается его ясность. Например, представленный выше код, предназначенный для обработки связанного списка, упрощается в результате применения ограничителей, но сэкономленное в процедурах LIST_INSERT′ и LIST_DELETE′ время равно всего лишь $O\,(1)$. Однако в других ситуациях применение ограничителей позволяет сделать код в циклах компактнее, а иногда даже уменьшить время работы в n или n^2 раз.

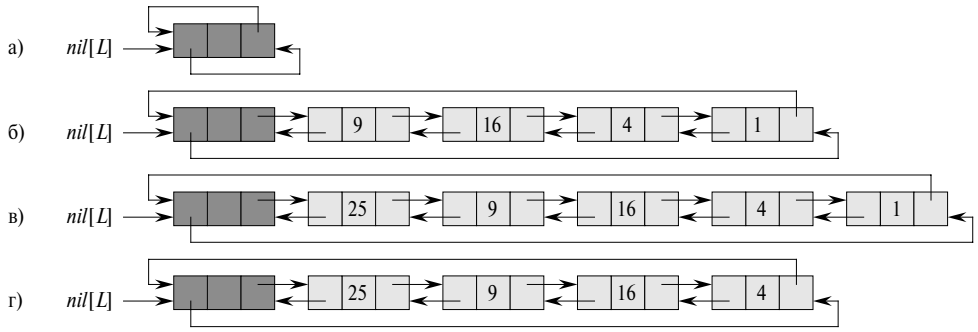


Рис. 10.4. Циклический дважды связанный список с ограничителями

Ограничители не стоит применять необдуманно. Если в программе обрабатывается большое количество маленьких списков, то дополнительное пространство, занимаемое ограничителями, может стать причиной значительного перерасхода памяти. В этой книге ограничители применяются лишь тогда, когда они действительно упрощают код.

Упражнения

10.2-1. Можно ли реализовать операцию INSERT над динамическим множеством в однократно связанном списке так, чтобы время ее работы было равно $O(1)$? А операцию DELETE?

10.2-2. Реализуйте стек с помощью однократно связанного списка L. Операции PUSH и POP должны по-прежнему выполняться в течение времени $O(1)$.

10.2-3. Реализуйте очередь с помощью однократно связанного списка L. Операции ENQUEUE и DEQUEUE должны выполняться в течение времени $O(1)$.

10.2-4. В каждой итерации цикла, входящего в состав процедуры LIST_SEARCH′, необходимо выполнить две проверки: $x \neq nil[L]$ и $key[x] \neq k$. Покажите, как избежать проверки $x \neq nil[L]$ в каждой итерации.

10.2-5. Реализуйте базовые словарные операции INSERT, DELETE и SEARCH с помощью однократно связанного циклического списка. Определите время работы этих процедур.

10.2-6. Операция UNION (объединение) над динамическим множеством принимает в качестве входных данных два отдельных множества S_1 и S_2 и возвращает множество $S = S_1 \cup S_2$, состоящее из всех элементов множеств S_1 и S_2. В результате выполнения этой операции множества S_1 и S_2 обычно разрушаются. Покажите, как организовать поддержку операции UNION со временем работы $O(1)$ с помощью подходящей списочной структуры данных.

10.2-7. Разработайте нерекурсивную процедуру со временем работы $\Theta(n)$, обращающую порядок расположения элементов в однократно связанном списке. Процедура должна использовать некоторый постоянный объем памяти помимо памяти, необходимой для хранения самого списка.

⋆ 10.2-8. Объясните, как можно реализовать дважды связанный список, используя при этом всего лишь один указатель $np[x]$ в каждом элементе вместо обычных двух ($next$ и $prev$). Предполагается, что значения всех указателей могут рассматриваться как k-битовые целые чисел, а величина $np[x]$ определяется как $np[x] = next[x]$ XOR $prev[x]$, т.е. как k-битовое "исключающее или" значений $next[x]$ и $prev[x]$. (Значение NIL представляется нулем.) Не забудьте указать, какая информация нужна для доступа к голове списка. Покажите, как реализовать операции SEARCH, INSERT и DELETE в таком списке. Покажите также, как можно обратить порядок элементов в таком списке за время $O(1)$.

10.3 Реализация указателей и объектов

Как реализовать указатели и объекты в таких языках, как Fortran, где их просто нет? В данном разделе мы ознакомимся с двумя путями реализации связанных структур данных, в которых такой тип данных, как указатель, в явном виде не используется. Объекты и указатели будут созданы на основе массивов и индексов.

Представление объектов с помощью нескольких массивов

Набор объектов с одинаковыми полями можно представить с помощью массивов. Каждому полю в таком представлении будет соответствовать свой массив. В качестве примера рассмотрим рис. 10.5, где проиллюстрирована реализация с помощью трех массивов связанного списка, представленного на рис. 10.3*a*. В каждом столбце элементов на рисунке представлен отдельный объект. В массиве key содержатся значения ключей элементов, входящих в данный момент в динамическое множество, а указатели хранятся в массивах $next$ и $prev$. Для заданного индекса массива x элементы $key[x]$, $next[x]$ и $prev[x]$ представляют объект в связанном списке. В такой интерпретации указатель x — это просто общий индекс в массивах key, $next$ и $prev$.

Указатели при таком способе хранения представляют собой просто индексы объектов, на которые они указывают. На рис. 10.3*a* объект с ключом 4 в связанном списке следует после объекта с ключом 16. Соответственно, на рис. 10.5 ключ 4 является значением элемента $key[2]$, а ключ 16 — значением элемента $key[5]$, поэтому $next[5] = 2$ и $prev[2] = 5$. В качестве значения NIL обычно используется целое число (например, 0 или -1), которое не может быть индексом массива. В переменной L содержится индекс головного элемента списка.

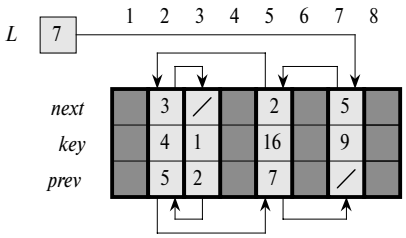


Рис. 10.5. Связанный список, представленный массивами key, $next$ и $prev$

В нашем псевдокоде квадратные скобки используются как для обозначения индекса массива, так и для выбора поля (атрибута) объекта. В любом случае значение выражений $key\,[x]$, $next\,[x]$ и $prev\,[x]$ согласуется с практической реализацией.

Представление объектов с помощью одного массива

Обычно слова в памяти компьютера адресуются с помощью целых чисел от 0 до $M-1$, где M — это достаточно большое целое число. Во многих языках программирования объект занимает непрерывную область памяти компьютера. Указатель — это просто адрес первой ячейки памяти, где находится начало объекта. Другие ячейки памяти, занимаемые объектом, можно индексировать путем добавления к указателю величины соответствующего смещения.

Этой же стратегией можно воспользоваться при реализации объектов в средах программирования, в которых отсутствуют указатели. Например, на рис. 10.6 показано, как можно реализовать связанный список, знакомый нам из рис. 10.3a и 10.5, с помощью одного массива A. Объект занимает подмассив смежных элементов $A\,[j..k]$. Каждое поле объекта соответствует смещению, величина которого принимает значения от 0 до $k-j$, а указателем на объект является индекс j. Каждый элемент списка — это объект, занимающий по три расположенных рядом элемента массива. Указатель на объект — это индекс его первого элемента. Объекты, в которых содержатся элементы списка, на рисунке отмечены светло-серым цветом. На рис. 10.6 сдвиги, отвечающие полям key, $next$ и $prev$, равны 0, 1 и 2 соответственно. Чтобы считать значение поля $prev\,[i]$ для данного указателя i, к значению указателя добавляется величина сдвига 2, в результате чего получается $A\,[i+2]$.

Представление в виде единственного массива можно считать более гибким в том плане, что с его помощью в одном и том же массиве можно хранить объекты различной длины. Задача управления такими неоднородными наборами объектов сложнее, чем аналогичная задача для однородного набора объектов, где все объекты состоят из одинаковых полей. Поскольку большинство структур данных,

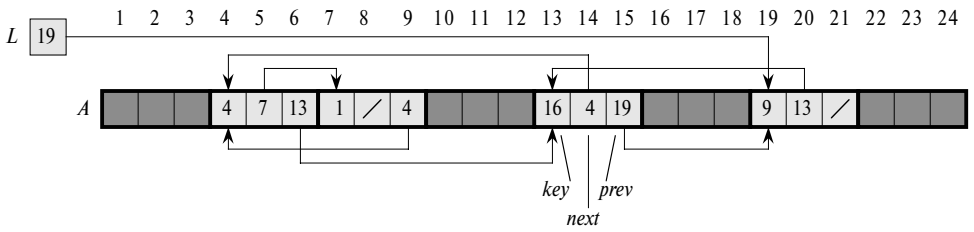


Рис. 10.6. Связанный список, представленный единственным массивом

которое нам предстоит рассмотреть, состоит из однородных элементов, для наших целей достаточно использовать представление объектов с помощью нескольких массивов.

Выделение и освобождение памяти

При добавлении нового элемента в список надо выделить для него место в памяти, что влечет за собой необходимость учета использования адресов. В некоторых системах функцию определения того, какой объект больше не используется, выполняет "***сборщик мусора***" (garbage collector). Однако многие приложения достаточно просты и вполне способны самостоятельно возвращать неиспользованное объектами пространство модулю управления памятью. Давайте рассмотрим задачу выделения и освобождения памяти для однородных объектов на примере дважды связанного списка, представленного с помощью нескольких массивов.

Предположим, что в таком представлении используются массивы длиной m, и что в какой-то момент динамическое множество содержит $n \leqslant m$ элементов. В этом случае n объектов представляют элементы, которые находятся в данный момент времени в динамическом множестве, а $m - n$ элементов ***свободны***. Их можно использовать для представления элементов, которые будут вставляться в динамическое множество в будущем.

Свободные объекты хранятся в однократно связанном списке, который мы назовем ***списком свободных позиций*** (free list). Список свободных позиций использует только массив $next$, в котором хранятся указатели $next$ списка. Головной элемент списка свободных позиций находится в глобальной переменной $free$. Когда динамическое множество, представленное связанным списком L, не является пустым, список свободных позиций используется вместе со списком L, как показано на рис. 10.7. Заметим, что каждый объект в таком представлении находится либо в списке L, либо в списке свободных позиций, но не в обоих списках одновременно.

Список свободных позиций — это стек: очередной выделяемый объект является последним освобожденным. Таким образом, реализовать процедуры выделения и освобождения памяти для объектов можно с помощью стековых операций PUSH

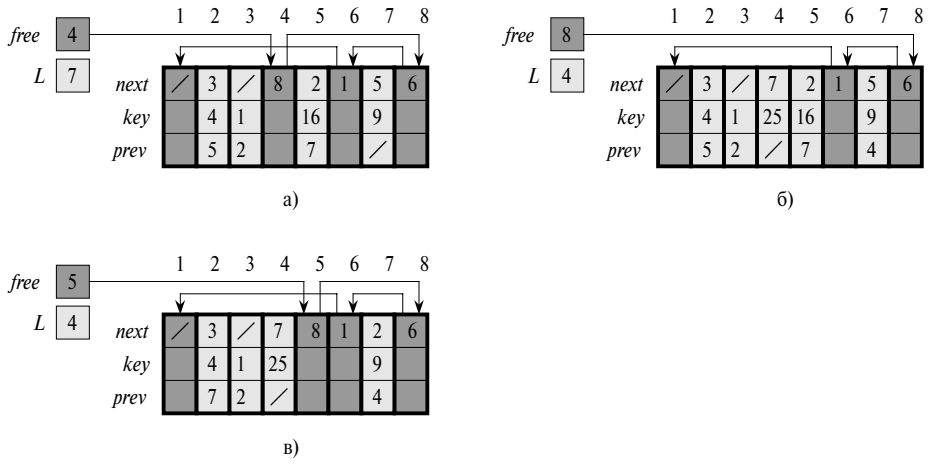


Рис. 10.7. Процедуры выделения и освобождения объекта

и POP. Глобальная переменная *free*, использующаяся в приведенных ниже процедурах, указывает на первый элемент списка свободных позиций:

```
ALLOCATE_OBJECT()
1  if free = NIL
2     then error "Нехватка памяти"
3     else x ← free
4          free ← next[x]
5          return x

FREE_OBJECT(x)
1  next[x] ← free
2  free ← x
```

В начальный момент список свободных позиций содержит все n объектов, для которых не выделено место. Когда в списке свободных позиций больше не остается элементов, процедура ALLOCATE_OBJECT выдает сообщение об ошибке.

На рис. 10.7 показано, как изменяется исходный список свободных позиций (рис. 10.7*а*) при вызове процедуры ALLOCATE_OBJECT(), которая возвращает индекс 4, и вставке в эту позицию нового элемента (рис. 10.7*б*), а также после вызова LIST_DELETE$(L, 5)$ с последующим вызовом FREE_OBJECT(5) (рис. 10.7*в*).

Часто один список свободных позиций обслуживает несколько связанных списков. На рис. 10.8 изображены два связанных списка и обслуживающий их список свободных позиций.

Время выполнения обеих процедур равно $O(1)$, что делает их весьма практичными. Эти процедуры можно модифицировать так, чтобы они работали с любым

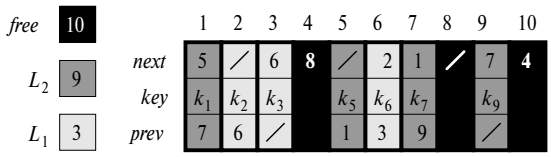


Рис. 10.8. Связанные списки L_1 (светло-серый) и L_2 (темно-серый), и обслуживающий их список свободных позиций (черный)

набором однородных объектов, лишь бы одно из полей объекта работало в качестве поля $next$ списка свободных позиций.

Упражнения

10.3-1. Изобразите последовательность $\langle 13, 4, 8, 19, 5, 11\rangle$, хранящуюся в дважды связанном списке, представленном с помощью нескольких массивов. Выполните это же задание для представления с помощью одного массива.

10.3-2. Разработайте процедуры ALLOCATE_OBJECT и FREE_OBJECT для однородного набора объектов, реализованных с помощью одного массива.

10.3-3. Почему нет необходимости присваивать или вновь устанавливать значения полей $prev$ при реализации процедур ALLOCATE_OBJECT и FREE_OBJECT?

10.3-4. Зачастую (например, при страничной организации виртуальной памяти) все элементы списка желательно располагать компактно, в непрерывном участке памяти. Разработайте такую реализацию процедур ALLOCATE_OBJECT и FREE_OBJECT, чтобы элементы списка занимали позиции $1..m$, где m — количество элементов списка. (*Указание*: воспользуйтесь реализацией стека с помощью массива.)

10.3-5. Пусть L — дважды связанный список длины m, который хранится в массивах key, $prev$ и $next$ длины n. Предположим, что управление этими массивами осуществляется с помощью процедур ALLOCATE_OBJECT и FREE_OBJECT, использующих дважды связанный список свободных позиций F. Предположим также, что ровно m из n элементов находятся в списке L, а остальные $n - m$ элементов — в списке свободных позиций. Разработайте процедуру COMPACTIFY_LIST(L, F), которая в качестве параметров получает список L и список свободных позиций F, и перемещает элементы списка L таким образом, чтобы они занимали ячейки массива с индексами $1, 2, \ldots, m$, а также преобразует список свободных позиций F так, что он остается корректным и содержит ячейки массива

с индексами $m+1, m+2, \ldots, n$. Время работы этой процедуры должно быть равным $\Theta(n)$, а объем использующейся ею дополнительной памяти не должен превышать некоторую фиксированную величину. Приведите тщательное обоснование корректности представленной процедуры.

10.4 Представление корневых деревьев

Приведенные в предыдущем разделе методы представления списков подходят для любых однородных структур данных. Данный раздел посвящен задаче представления корневых деревьев с помощью связанных структур данных. Мы начнем с рассмотрения бинарных деревьев, а затем рассмотрим представление деревьев с произвольным количеством дочерних элементов.

Каждый узел дерева представляет собой отдельный объект. Как и при изучении связанных списков, предполагается, что в каждом узле содержится поле key. Остальные интересующие нас поля — это указатели на другие узлы, и их вид зависит от типа дерева.

Бинарные деревья

Как показано на рис. 10.9, для хранения указателей на родительский, дочерний левый и дочерний правый узлы бинарного дерева T, используются поля p, $left$ и $right$. Если $p[x] = \text{NIL}$, то x — корень дерева. Если у узла x нет дочерних узлов, то $left[x] = right[x] = \text{NIL}$. Атрибут $root[T]$ указывает на корневой узел дерева T. Если $root[T] = \text{NIL}$, то дерево T пустое.

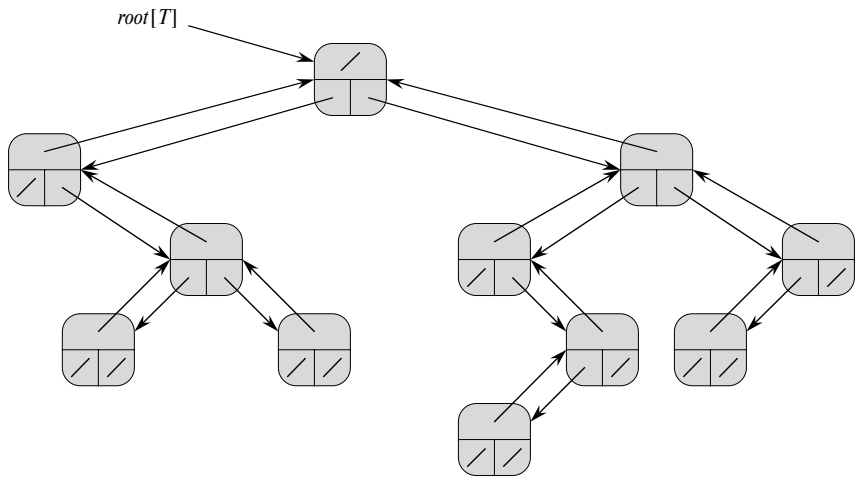


Рис. 10.9. Представление бинарного дерева T (поле key не показано)

Корневые деревья с произвольным ветвлением

Схему представления бинарных деревьев можно обобщить для деревьев любого класса, в которых количество дочерних узлов не превышает некоторой константы k. При этом поля правый и левый заменяются полями $child_1$, $child_2$, ..., $child_k$. Если количество дочерних элементов узла не ограничено, то эта схема не работает, поскольку заранее не известно, место для какого количества полей (или массивов, при использовании представления с помощью нескольких массивов) нужно выделить. Кроме того, если количество дочерних элементов k ограничено большой константой, но на самом деле у многих узлов потомков намного меньше, то значительный объем памяти расходуется напрасно.

К счастью, существует остроумная схема представления деревьев с произвольным количеством дочерних узлов с помощью бинарных деревьев. Преимущество этой схемы в том, что для любого корневого дерева с n дочерними узлами требуется объем памяти $O(n)$. На рис. 10.10 проиллюстрировано представление ***с левым дочерним и правым сестринским узлами*** (left-child, right-sibling representation). Как и в предыдущем представлении, в каждом узле этого представления содержится указатель p, а атрибут $root[T]$ указывает на корень дерева T. Однако вместо указателей на дочерние узлы каждый узел x содержит всего два указателя:

1. в поле $left\_child[x]$ хранится указатель на крайний левый дочерний узел узла x;

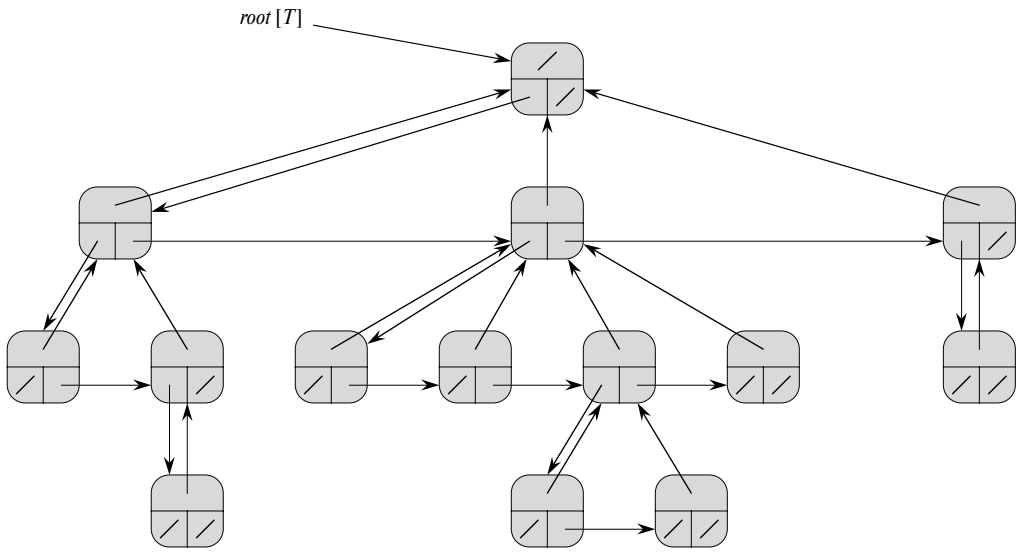


Рис. 10.10. Представление дерева с левым дочерним и правым сестринским узлами

2. в поле $right\_sibling[x]$ хранится указатель на узел, расположенный на одном уровне с узлом x справа от него.

Если узел x не имеет потомков, то $left\_child[x] = \text{NIL}$, а если узел x — крайний правый дочерний элемент какого-то родительского элемента, то $right\_sibling[x] = \text{NIL}$.

Другие представления деревьев

Иногда встречаются и другие способы представления корневых деревьев. Например, в главе 6 описано представление пирамиды, основанной на полном бинарном дереве, в виде индексированного массива. Деревья, о которых идет речь в главе 21, восходят только к корню, поэтому в них содержатся лишь указатели на родительские элементы; указатели на дочерние элементы отсутствуют. Здесь возможны самые различные схемы. Наилучший выбор схемы зависит от конкретного приложения.

Упражнения

10.4-1. Начертите бинарное дерево, корень которого имеет индекс 6, и которое представлено приведенными ниже полями.

Индекс	*key*	*left*	*right*
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

10.4-2. Разработайте рекурсивную процедуру, которая за время $O(n)$ выводит ключи всех n узлов бинарного дерева.

10.4-3. Разработайте нерекурсивную процедуру, которая за время $O(n)$ выводит ключи всех n узлов бинарного дерева. В качестве вспомогательной структуры данных воспользуйтесь стеком.

10.4-4. Разработайте процедуру, которая за время $O(n)$ выводит ключи всех n узлов произвольного корневого дерева. Дерево реализовано в представлении с левым дочерним и правым сестринским элементами.

⋆ 10.4-5. Разработайте нерекурсивную процедуру, которая за время $O(n)$ выводит ключи всех n узлов бинарного дерева. Объем дополнительной памяти (кроме той, что отводится под само дерево) не должен превышать некоторую константу. Кроме того, в процессе выполнения процедуры дерево (даже временно) не должно модифицироваться.

⋆ 10.4-6. В представлении произвольного корневого дерева с левым дочерним и правым сестринским узлами в каждом узле есть по три указателя: $left\_child$, $right\_sibling$ и $parent$. Если задан какой-то узел дерева, то определить его родительский узел и получить к нему доступ можно в течение фиксированного времени. Определить же все дочерние узлы и получить к ним доступ можно в течение времени, линейно зависящего от количества дочерних узлов. Разработайте способ хранения дерева с произвольным ветвлением, в каждом узле которого используется только два (а не три) указателя и одна логическая переменная, при котором родительский узел или все дочерние узлы определяются в течение времени, линейно зависящего от количества дочерних узлов.

Задачи

10-1. **Сравнения списков**

Определите асимптотическое время выполнения перечисленных в приведенной ниже таблице операций над элементами динамических множеств в наихудшем случае, если эти операции выполняются со списками перечисленных ниже типов.

	Неотсортированный однократно связанный список	Отсортированный однократно связанный список	Неотсортированный дважды связанный список	Отсортированный дважды связанный список
SEARCH(L, k)				
INSERT(L, x)				
DELETE(L, x)				
SUCCESSOR(L, x)				
PREDECESSOR(L, x)				
MINIMUM(L)				
MAXIMUM(L)				

10-2. **Реализация объединяемых пирамид с помощью связанных списков**

В ***объединяемой пирамиде*** (mergable heap) поддерживаются следующие операции: MAKE_HEAP (создание пустой пирамиды), INSERT, MINIMUM, EXTRACT_MINIMUM и UNION[1]. Покажите, как в каждом из перечисленных ниже случаев реализовать с помощью связанных списков объединяемые пирамиды. Постарайтесь, чтобы каждая операция выполнялась с максимальной эффективностью. Проанализируйте время работы каждой операции по отношению к размеру обрабатываемого динамического множества.

а) Списки отсортированы.

б) Списки не отсортированы.

в) Списки не отсортированы, и объединяемые динамические множества не перекрываются.

10-3. **Поиск в отсортированном компактном списке**

В упражнении 10.3-4 предлагается разработать компактную поддержку n-элементного списка в первых n ячейках массива. Предполагается, что все ключи различны, и что компактный список отсортирован, т.е. для всех $i = 1, 2, \ldots, n$, таких что $next\,[i] \neq$ NIL, выполняется соотношение $key\,[i] < key\,[next\,[i]]$. Покажите, что при данных предположениях математическое ожидание времени поиска с помощью приведенного ниже рандомизированного алгоритма равно $O\,(\sqrt{n})$:

```
COMPACT_LIST_SEARCH(L, n, k)
 1  i ← head[L]
 2  while i ≠ NIL и key[i] < k
 3        do j ← RANDOM(1, n)
 4           if key[i] < key[j] и key[j] ≤ k
 5              then i ← j
 6                   if key[i] = k
 7                      then return i
 8           i ← next[i]
 9  if i = NIL или key[i] > k
10     then return NIL
11     else  return i
```

[1]Поскольку в пирамиде поддерживаются операции MINIMUM и EXTRACT_MINIMUM, такую пирамиду можно назвать **объединяемой неубывающей пирамидой** (mergable min-heap). Аналогично, если бы в пирамиде поддерживались операции MAXIMUM и EXTRACT_MAXIMUM, ее можно было бы назвать **объединяемой невозрастающей пирамидой** (mergable max-heap).

Если в представленной выше процедуре опустить строки 3–7, получится обычный алгоритм, предназначенный для поиска в отсортированном связанном списке, в котором индекс i пробегает по всем элементам в списке. Поиск прекращается в тот момент, когда происходит "обрыв" индекса i в конце списка или когда $key\,[i] \geqslant k$. В последнем случае, если выполняется соотношение $key\,[i] = k$, то понятно, что ключ k найден. Если же $key\,[i] > k$, то ключ k в списке отсутствует и поэтому следует прекратить поиск.

В строках 3–7 предпринимается попытка перейти к случайно выбранной ячейке j. Такой переход дает преимущества, если величина $key\,[j]$ больше величины $key\,[i]$, но не превышает значения k. В этом случае индекс j соответствует элементу списка, к которому рано или поздно был бы осуществлен доступ при обычном поиске. Поскольку список компактен, любой индекс j в интервале от 1 до n отвечает некоторому объекту списка и не может быть пустым местом из списка свободных позиций.

Вместо анализа производительности процедуры COMPACT_LIST_SEARCH мы проанализируем связанный с ним алгоритм COMPACT_LIST_SEARCH′, в котором содержатся два отдельных цикла. В этом алгоритме используется дополнительный параметр t, определяющий верхнюю границу количества итераций в первом цикле:

```
COMPACT_LIST_SEARCH′(L, n, k, t)
 1  i ← head[L]
 2  for q ← 1 to t
 3       do j ← RANDOM(1, n)
 4          if key[i] < key[j] и key[j] ⩽ k
 5             then i ← j
 6                  if key[i] = k
 7                     then return i
 8  while i ≠ NIL и key[i] < k
 9       do i ← next[i]
10  if i = NIL или key[i] > k
11       then return NIL
12       else return i
```

Для простоты сравнения алгоритмов COMPACT_LIST_SEARCH(L, n, k) и COMPACT_LIST_SEARCH′(L, n, k, t) будем считать, что последовательность случайных чисел, которая возвращается процедурой RANDOM$(1, n)$, одна и та же для обоих алгоритмов.

а) Предположим, что в ходе работы цикла **while** в строках 2–8 процедуры COMPACT_LIST_SEARCH(L, n, k), выполняется t итераций. До-

кажите, что процедура COMPACT_LIST_SEARCH$'(L, n, k, t)$ даст тот же ответ, и что общее количество итераций в циклах **for** и **while** этой процедуры не меньше t.

Пусть при работе процедуры COMPACT_LIST_SEARCH$'(L, n, k, t)$ X_t — случайная величина, описывающая расстояние в связанном списке (т.е. длину цепочки из указателей $next$) от элемента с индексом i до искомого ключа k после t итераций цикла **for** в строках 2–7.

б) Докажите, что математическое ожидание времени работы процедуры COMPACT_LIST_SEARCH$'(L, n, k, t)$ равно $O\left(t + E\left[X_t\right]\right)$.

в) Покажите, что $E\left[X_t\right] \leqslant \sum_{r=1}^{n} \left(1 - r/n\right)^t$. (*Указание*: воспользуйтесь уравнением (B.24)).

г) Покажите, что $\sum_{r=0}^{n-1} r^t \leqslant n^{t+1}/(t+1)$.

д) Докажите, что $E\left[X_t\right] \leqslant n/(t+1)$.

е) Покажите, что математическое ожидание времени работы процедуры COMPACT_LIST_SEARCH$'(L, n, k, t)$ равно $O\left(t + n/t\right)$.

ж) Сделайте вывод о том, что математическое ожидание времени работы процедуры COMPACT_LIST_SEARCH равно $O\left(\sqrt{n}\right)$.

з) Объясните, зачем при анализе процедуры COMPACT_LIST_SEARCH понадобилось предположение о том, что все ключи различны. Покажите, что случайные переходы не обязательно приведут к сокращению асимптотического времени работы, если в списке содержатся ключи с одинаковыми значениями.

Заключительные замечания

Прекрасными справочными пособиями по элементарным структурам данных являются книги Ахо (Aho), Хопкрофта (Hopcroft) и Ульмана (Ullman) [6] и Кнута (Knuth) [182]. Описание базовых структур данных и их реализации в конкретных языках программирования можно также найти во многих других учебниках. В качестве примера можно привести книги Гудрича (Goodrich) и Тамазии (Tamassia) [128], Мейна (Main) [209], Шаффера (Shaffer) [273] и Вейса (Weiss) [310, 312, 313]. В книге Гоннета (Gonnet) [126] приведены экспериментальные данные, описывающие производительность операций над многими структурами данных.

Начало использования стеков и очередей в информатике в качестве структур данных по сей день остается недостаточно ясным. Вопрос усложняется тем, что соответствующие понятия существовали в математике и применялись на практике при выполнении деловых расчетов на бумаге еще до появления первых цифровых вычислительных машин. В своей книге [182] Кнут приводит относящиеся

к 1947 году цитаты А. Тьюринга (A.M. Turing), в которых идет речь о разработке стеков для компоновки подпрограмм.

По-видимому, структуры данных, основанные на применении указателей, также являются “народным” изобретением. Согласно Кнуту, указатели, скорее всего, использовались еще на первых компьютерах с памятью барабанного типа. В языке программирования A-1, разработанном Хоппером (G.M. Hopper) в 1951 году, алгебраические формулы представляются в виде бинарных деревьев. Кнут считает, что указатели получили признание и широкое распространение благодаря языку программирования IPL-II, разработанному в 1956 году Ньюэлом (A. Newell), Шоу (J.C. Shaw) и Симоном (H.A. Simon). В язык IPL-III, разработанный в 1957 году теми же авторами, операции со стеком включены явным образом.

ГЛАВА 11

Хеш-таблицы

Для многих приложений достаточны динамические множества, поддерживающие только стандартные словарные операции вставки, поиска и удаления. Например, компилятор языка программирования поддерживает таблицу символов, в которой ключами элементов являются произвольные символьные строки, соответствующие идентификаторам в языке. Хеш-таблица представляет собой эффективную структуру данных для реализации словарей. Хотя на поиск элемента в хеш-таблице может в наихудшем случае потребоваться столько же времени, что и в связанном списке, а именно $\Theta(n)$, на практике хеширование исключительно эффективно. При вполне обоснованных допущениях математическое ожидание времени поиска элемента в хеш-таблице составляет $O(1)$.

Хеш-таблица представляет собой обобщение обычного массива. Возможность прямой индексации элементов обычного массива обеспечивает доступ к произвольной позиции в массиве за время $O(1)$. Более подробно прямая индексация рассматривается в разделе 11.1; она применима, если мы в состоянии выделить массив размера, достаточного для того, чтобы для каждого возможного значения ключа имелась своя ячейка.

Если количество реально хранящихся в массиве ключей мало́ по сравнению с количеством возможных значений ключей, эффективной альтернативой массива с прямой индексацией становится хеш-таблица, которая обычно использует массив с размером, пропорциональным количеству реально хранящихся в нем ключей. Вместо непосредственного использования ключа в качестве индекса массива, индекс *вычисляется* по значению ключа. В разделе 11.2 представлены основные идеи хеширования, в первую очередь направленные на разрешение коллизий (когда несколько ключей отображается в один и тот же индекс массива) при помощи

цепочек. В разделе 11.3 описывается, каким образом на основе значений ключей могут быть вычислены индексы массива. Здесь будет рассмотрено и проанализировано несколько вариантов хеш-функций. В разделе 11.4 вы познакомитесь с методом открытой адресации, представляющим собой еще один способ разрешения коллизий. Главный вывод, который следует из всего изложенного материала: хеширование представляет собой исключительно эффективную и практичную технологию — в среднем все базовые словарные операции требуют только $O(1)$ времени. В разделе 11.5 будет дано пояснение, каким образом "идеальное хеширование" может поддерживать *наихудшее* время поиска $O(1)$ в случае использования статического множества хранящихся ключей (т.е. когда множество ключей, будучи сохраненным, более не изменяется).

11.1 Таблицы с прямой адресацией

Прямая адресация представляет собой простейшую технологию, которая хорошо работает для небольших множеств ключей. Предположим, что приложению требуется динамическое множество, каждый элемент которого имеет ключ из множества $U = \{0, 1, \ldots, m-1\}$, где m не слишком велико. Кроме того, предполагается, что никакие два элемента не имеют одинаковых ключей.

Для представления динамического множества мы используем массив, или ***таблицу с прямой адресацией***, который обозначим как $T[0..m-1]$, каждая ***позиция***, или ***ячейка*** (position, slot), которого соответствует ключу из пространства ключей U. На рис. 11.1 представлен данный подход. Ячейка k указывает на элемент множества с ключом k. Если множество не содержит элемента с ключом k, то $T[k] = \text{NIL}$. На рисунке каждый ключ из пространства $U = \{0, 1, \ldots, 9\}$ соответствует индексу таблицы. Множество реальных ключей $K = \{2, 3, 5, 8\}$ определяет ячейки таблицы, которые содержат указатели на элементы. Остальные ячейки (закрашенные темным цветом) содержат значение NIL.

Реализация словарных операций тривиальна:

DIRECT_ADDRESS_SEARCH(T, k)
  return $T[k]$

DIRECT_ADDRESS_INSERT(T, x)
  $T[key[x]] \leftarrow x$

DIRECT_ADDRESS_DELETE(T, x)
  $T[key[x]] \leftarrow \text{NIL}$

Каждая из приведенных операций очень быстрая: время их работы равно $O(1)$.

В некоторых приложениях элементы динамического множества могут храниться непосредственно в таблице с прямой адресацией. То есть вместо хранения

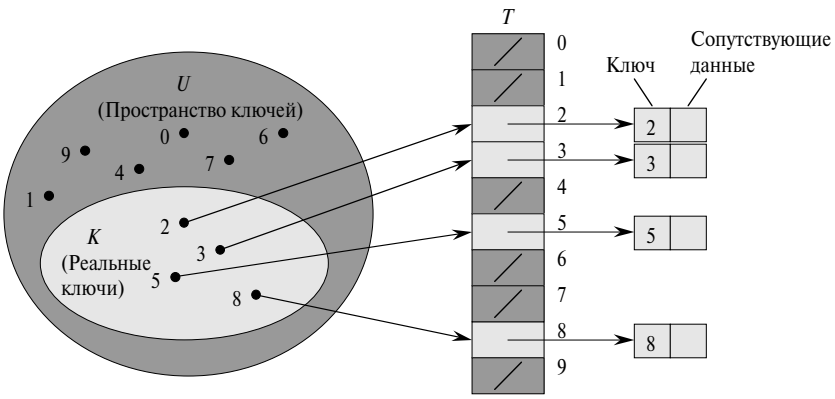


Рис. 11.1. Реализация динамического множества с использованием таблицы с прямой адресацией

ключей и сопутствующих данных элементов в объектах, внешних по отношению к таблице с прямой адресацией, а в таблице — указателей на эти объекты, эти объекты можно хранить непосредственно в ячейках таблицы (что тем самым приводит к экономии используемой памяти). Кроме того, зачастую хранение ключа не является необходимым условием, поскольку если мы знаем индекс объекта в таблице, мы тем самым знаем и его ключ. Однако если ключ не хранится в ячейке таблицы, то нам нужен какой-то иной механизм для того, чтобы помечать пустые ячейки.

Упражнения

11.1-1. Предположим, что динамическое множество S представлено таблицей с прямой адресацией T длины m. Опишите процедуру, которая находит максимальный элемент S. Чему равно время работы этой процедуры в наихудшем случае?

11.1-2. **Битовый вектор** представляет собой массив битов (нулей и единиц). Битовый вектор длиной m занимает существенно меньше места, чем массив из m указателей. Каким образом можно использовать битовый вектор для представления динамического множества различных элементов без сопутствующих данных? Словарные операции должны выполняться за время $O(1)$.

11.1-3. Предложите способ реализации таблицы с прямой адресацией, в которой ключи хранящихся элементов могут совпадать, а сами элементы — иметь сопутствующие данные. Все словарные операции — вставки, удаления и поиска — должны выполняться за время $O(1)$. (Не забудьте, что ар-

гументом процедуры удаления является указатель на удаляемый объект, а не ключ.)

⋆ 11.1-4. Предположим, что мы хотим реализовать словарь с использованием прямой адресации *очень большого* массива. Первоначально в массиве может содержаться "мусор", но инициализация всего массива нерациональна в силу его размера. Разработайте схему реализации словаря с прямой адресацией при описанных условиях. Каждый хранимый объект должен использовать $O(1)$ памяти; операции вставки, удаления и поиска должны выполняться за время $O(1)$; инициализация структуры данных также должна выполняться за время $O(1)$. (*Указание*: для определения, является ли данная запись в большом массиве корректной или нет, воспользуйтесь дополнительным стеком, размер которого равен количеству ключей, сохраненных в словаре.)

11.2 Хеш-таблицы

Недостаток прямой адресации очевиден: если пространство ключей U велико, хранение таблицы T размером $|U|$ непрактично, а то и вовсе невозможно — в зависимости от количества доступной памяти и размера пространства ключей. Кроме того, множество K *реально сохраненных* ключей может быть мало́ по сравнению с пространством ключей U, а в этом случае память, выделенная для таблицы T, в основном расходуется напрасно.

Когда множество K хранящихся в словаре ключей гораздо меньше пространства возможных ключей U, хеш-таблица требует существенно меньше места, чем таблица с прямой адресацией. Точнее говоря, требования к памяти могут быть снижены до $\Theta(|K|)$, при этом время поиска элемента в хеш-таблице остается равным $O(1)$. Надо только заметить, что это граница *среднего времени* поиска, в то время как в случае таблицы с прямой адресацией эта граница справедлива для *наихудшего случая*.

В случае прямой адресации элемент с ключом k хранится в ячейке k. При хешировании этот элемент хранится в ячейке $h(k)$, т.е. мы используем ***хеш-функцию*** h для вычисления ячейки для данного ключа k. Функция h отображает пространство ключей U на ячейки ***хеш-таблицы*** $T[0..m-1]$:

$$h : U \to \{0, 1, \ldots, m-1\}.$$

Мы говорим, что элемент с ключом k хешируется в ячейку $h(k)$; величина $h(k)$ называется ***хеш-значением*** ключа k. На рис. 11.2 представлена основная идея хеширования. Цель хеш-функции состоит в том, чтобы уменьшить рабочий диапазон индексов массива, и вместо $|U|$ значений мы можем обойтись всего лишь m значениями. Соответственно снижаются и требования к количеству памяти.

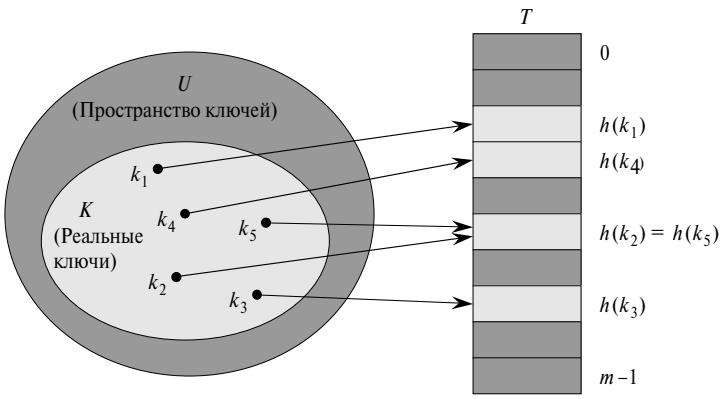


Рис. 11.2. Использование хеш-функции h для отображения ключей в ячейки хеш-таблицы. Ключи k_2 и k_5 отображаются в одну ячейку, вызывая коллизию

Однако здесь есть одна проблема: два ключа могут быть хешированы в одну и ту же ячейку. Такая ситуация называется ***коллизией***. К счастью, имеются эффективные технологии для разрешения конфликтов, вызываемых коллизиями.

Конечно, идеальным решением было бы полное устранение коллизий. Мы можем попытаться добиться этого путем выбора подходящей хеш-функции h. Одна из идей заключается в том, чтобы сделать h "случайной", что позволило бы избежать коллизий или хотя бы минимизировать их количество (этот характер функции хеширования отображается в самом глаголе "to hash", который означает "мелко порубить, перемешать"). Само собой разумеется, функция h должна быть детерминистической и для одного и того же значения k всегда давать одно и то же хеш-значение $h(k)$. Однако поскольку $|U| > m$, должно существовать как минимум два ключа, которые имеют одинаковое хеш-значение. Таким образом, полностью избежать коллизий невозможно в принципе, и хорошая хеш-функция в состоянии только минимизировать количество коллизий. Таким образом, нам крайне необходим метод разрешения возникающих коллизий.

В оставшейся части данного раздела мы рассмотрим простейший метод разрешения коллизий — метод цепочек. В разделе 11.4 вы познакомитесь с еще одним методом разрешения коллизий, который называется методом открытой адресации.

Разрешение коллизий при помощи цепочек

При использовании данного метода мы объединяем все элементы, хешированные в одну и ту же ячейку, в связанный список, как показано на рис. 11.3. Ячейка j содержит указатель на заголовок списка всех элементов, хеш-значение ключа которых равно j; если таких элементов нет, ячейка содержит значение NIL. На

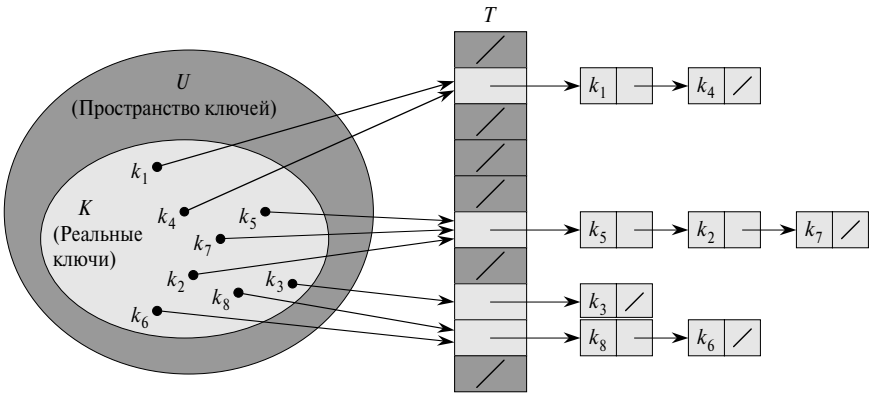


Рис. 11.3. Разрешение коллизий при помощи цепочек

рис. 11.3 показано разрешение коллизий, возникающих из-за того, что $h(k_1) = h(k_4)$, $h(k_5) = h(k_2) = h(k_7)$ и $h(k_8) = h(k_6)$.

Словарные операции в хеш-таблице с использованием цепочек для разрешения коллизий реализуются очень просто:

CHAINED_HASH_INSERT(T, x)

Вставить x в заголовок списка $T[h(key[x])]$

CHAINED_HASH_SEARCH(T, k)

Поиск элемента с ключом k в списке $T[h(k)]$

CHAINED_HASH_DELETE(T, x)

Удаление x из списка $T[h(key[x])]$

Время, необходимое для вставки в наихудшем случае, равно $O(1)$. Процедура вставки выполняется очень быстро, поскольку предполагается, что вставляемый элемент отсутствует в таблице. При необходимости это предположение может быть проверено путем выполнения поиска перед вставкой. Время работы поиска в наихудшем случае пропорционально длине списка; мы проанализируем эту операцию немного позже. Удаление элемента может быть выполнено за время $O(1)$ при использовании двусвязных списков. (Обратите внимание на то, что процедура CHAINED_HASH_DELETE принимает в качестве аргумента элемент x, а не его ключ, поэтому нет необходимости в предварительном поиске x. Если список односвязный, то передача в качестве аргумента x не дает нам особого выигрыша, поскольку для корректного обновления поля $next$ предшественника x нам все равно надо выполнить поиск x в списке $T[h(key[x])]$. В таком случае, как нетрудно понять, удаление и поиск имеют по сути одно и то же время работы.)

Анализ хеширования с цепочками

Насколько высока производительность хеширования с цепочками? В частности, сколько времени требуется для поиска элемента с данным ключом?

Пусть у нас есть хеш-таблица T с m ячейками, в которых хранятся n элементов. Определим ***коэффициент заполнения*** таблицы T как $\alpha = n/m$, т.е. как среднее количество элементов, хранящихся в одной цепочке. Наш анализ будет опираться на значение величины α, которая может быть меньше, равна или больше единицы.

В наихудшем случае хеширование с цепочками ведет себя крайне неприятно: все n ключей хешированы в одну и ту же ячейку, создавая список длиной n. Таким образом, время поиска в наихудшем случае равно $\Theta(n)$ плюс время вычисления хеш-функции, что ничуть не лучше, чем в случае использования связанного списка для хранения всех n элементов. Понятно, что использование хеш-таблиц в наихудшем случае совершенно бессмысленно. (Идеальное хеширование (применимое в случае статического множества ключей), которое будет рассмотрено в разделе 11.5, обеспечивает высокую производительность даже в наихудшем случае.)

Средняя производительность хеширования зависит от того, насколько хорошо хеш-функция h распределяет множество сохраняемых ключей по m ячейкам в среднем. Мы рассмотрим этот вопрос подробнее в разделе 11.3, а пока будем полагать, что все элементы хешируются по ячейкам равномерно и независимо, и назовем данное предположение "***простым равномерным хешированием***" (simple uniform hashing).

Обозначим длины списков $T[j]$ для $j = 0, 1, \ldots, m-1$ как n_j, так что

$$n = n_0 + n_1 + \cdots + n_{m-1}, \tag{11.1}$$

а среднее значение n_j равно $\mathrm{E}[n_j] = \alpha = n/m$.

Мы считаем, что хеш-значение $h(k)$ может быть вычислено за время $O(1)$, так что время, необходимое для поиска элемента с ключом k, линейно зависит от длины $n_{h(k)}$ списка $T[h(k)]$. Не учитывая время $O(1)$, требующееся для вычисления хеш-функции и доступа к ячейке $h(k)$, рассмотрим математическое ожидание количества элементов, которое должно быть проверено алгоритмом поиска (т.е. количество элементов в списке $T[h(k)]$, которые проверяются на равенство их ключей величине k). Мы должны рассмотреть два случая: во-первых, когда поиск неудачен и в таблице нет элементов с ключом k, и, во-вторых, когда поиск заканчивается успешно и в таблице определяется элемент с ключом k.

Теорема 11.1. В хеш-таблице с разрешением коллизий методом цепочек математическое ожидание времени неудачного поиска в предположении простого равномерного хеширования равно $\Theta(1 + \alpha)$.

Доказательство. В предположении простого равномерного хеширования любой ключ k, который еще не находится в таблице, может быть помещен с равной вероятностью в любую из m ячеек. Математическое ожидание времени неудачного поиска ключа k равно времени поиска до конца списка $T\left[h\left(k\right)\right]$, математическое ожидание длины которого $\mathrm{E}\left[n_{h(k)}\right] = \alpha$. Таким образом, при неудачном поиске математическое ожидание количества проверяемых элементов равно α, а общее время, необходимое для поиска, включая время вычисления хеш-функции $h\left(k\right)$, равно $\Theta\left(1+\alpha\right)$. ■

Успешный поиск несколько отличается от неудачного, поскольку вероятность поиска в списке различна для разных списков и пропорциональна количеству содержащихся в нем элементов. Тем не менее, и в этом случае математическое ожидание времени поиска остается равным $\Theta\left(1+\alpha\right)$.

Теорема 11.2. В хеш-таблице с разрешением коллизий методом цепочек математическое ожидание времени успешного поиска в предположении простого равномерного хеширования в среднем равно $\Theta\left(1+\alpha\right)$.

Доказательство. Мы считаем, что искомый элемент с равной вероятностью может быть любым элементом, хранящимся в таблице. Количество элементов, проверяемых в процессе успешного поиска элемента x, на 1 больше, чем количество элементов, находящихся в списке перед x. Элементы, находящиеся в списке до x, были вставлены в список после того, как элемент x был сохранен в таблице, так как новые элементы помещаются в начало списка. Для того чтобы найти математическое ожидание количества проверяемых элементов, мы возьмем среднее значение по всем элементам таблицы, которое равно 1 плюс математическое ожидание количества элементов, добавленных в список после искомого. Пусть x_i обозначает i-й элемент, вставленный в таблицу ($i = 1, 2, \ldots, n$), а $k_i = key\left[x_i\right]$. Определим для ключей k_i и k_j индикаторную случайную величину $X_{ij} = \mathrm{I}\left\{h\left(k_i\right) = h\left(k_j\right)\right\}$. В предположении простого равномерного хеширования, $\Pr\left\{h\left(k_i\right) = h\left(k_j\right)\right\} = 1/m$ и, в соответствии с леммой 5.1, $\mathrm{E}\left[X_{ij}\right] = 1/m$. Таким образом, математическое ожидание числа проверяемых элементов в случае успешного поиска равно

$$\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] =$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}\left[X_{ij}\right]\right) = \qquad \text{(в силу линейности математического ожидания)}$$

$$
\begin{aligned}
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) = \\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) = \\
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) = \\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) = \qquad \text{(в соответствии с (A.1))} \\
&= 1+\frac{n-1}{m} = \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}.
\end{aligned}
$$

Следовательно, полное время, необходимое для проведения успешного поиска (включая время на вычисление хеш-функции), составляет $\Theta(2+\alpha/2-\alpha/2n) = \Theta(1+\alpha)$. ■

Что же означает проведенный анализ? Если количество ячеек в хеш-таблице, как минимум, пропорционально количеству элементов, хранящихся в ней, то $n = O(m)$ и, следовательно, $\alpha = n/m = O(m)/m = O(1)$, а значит, поиск элемента в хеш-таблице в среднем требует постоянного времени. Поскольку в худшем случае вставка элемента в хеш-таблицу занимает $O(1)$ времени (как и удаление элемента при использовании двусвязных списков), можно сделать вывод, что все словарные операции в хеш-таблице в среднем выполняются за время $O(1)$.

Упражнения

11.2-1. Предположим, что у нас есть хеш-функция h для хеширования n различных ключей в массив T, длина которого равна m. Чему равно математическое ожидание количества коллизий в предположении простого равномерного хеширования (точнее, мощности множества $\{\{k,l\} : k \neq l$ и $h(k) = h(l)\}$)?

11.2-2. Продемонстрируйте вставку ключей 5, 28, 19, 15, 20, 33, 12, 17, 10 в хеш-таблицу с разрешением коллизий методом цепочек. Таблица имеет 9 ячеек, а хеш-функция имеет вид $h(k) = k \bmod 9$.

11.2-3. Профессор предполагает, что можно повысить эффективность хеширования с разрешением коллизий методом цепочек, если поддерживать списки в упорядоченном состоянии. Каким образом такое изменение алгоритма повлияет на время выполнения успешного поиска, неудачного поиска, вставки и удаления?

11.2-4. Предложите способ хранения элементов внутри самой хеш-таблицы, при котором неиспользуемые ячейки связываются в один список свободных мест. Считается, что в каждой ячейке может храниться флаг и либо один элемент и указатель, либо два указателя. Все словарные операции и операции над списком свободных мест должны выполняться за время $O(1)$. Должен ли список свободных мест быть двусвязным или можно обойтись односвязным списком?

11.2-5. Покажите, что если $|U| > nm$, то имеется подмножество пространства ключей U размером n, состоящее из ключей, которые хешируются в одну и ту же ячейку, так что время поиска в хеш-таблице с разрешением коллизий методом цепочек в худшем случае равно $\Theta(n)$.

11.3 Хеш-функции

В этом разделе мы рассмотрим некоторые вопросы, связанные с разработкой качественных хеш-функций, и познакомимся с тремя схемами их построения. Две из них, хеширование делением и хеширование умножением, эвристичны по своей природе, в то время как третья схема — универсальное хеширование — использует рандомизацию для обеспечения доказуемого качества.

Чем определяется качество хеш-функции

Качественная хеш-функция удовлетворяет (приближенно) предположению простого равномерного хеширования: для каждого ключа равновероятно помещение в любую из m ячеек, независимо от хеширования остальных ключей. К сожалению, это условие обычно невозможно проверить, поскольку, как правило, распределение вероятностей, в соответствии с которым поступают вносимые в таблицу ключи, неизвестно; кроме того, вставляемые ключи могут не быть независимыми.

Иногда распределение вероятностей оказывается известным. Например, если известно, что ключи представляют собой случайные действительные числа, равномерно распределенные в диапазоне $0 \leqslant k < 1$, то хеш-функция $h(k) = \lfloor km \rfloor$ удовлетворяет условию простого равномерного хеширования.

На практике при построении качественных хеш-функций зачастую используются различные эвристические методики. В процессе построения большую помощь оказывает информация о распределении ключей. Рассмотрим, например, таблицу символов компилятора, в которой ключами служат символьные строки, представляющие идентификаторы в программе. Зачастую в одной программе встречаются похожие идентификаторы, например, `pt` и `pts`. Хорошая хеш-функция должна минимизировать шансы попадания этих идентификаторов в одну ячейку хеш-таблицы.

При построении хеш-функции хорошим подходом является подбор функции таким образом, чтобы она никак не коррелировала с закономерностями, которым могут подчиняться существующие данные. Например, метод деления, который рассматривается в разделе 11.3.1, вычисляет хеш-значение как остаток от деления ключа на некоторое простое число. Если это простое число никак не связано с распределением исходных данных, метод часто дает хорошие результаты.

В заключение заметим, что некоторые приложения хеш-функций могут накладывать дополнительные требования, помимо требований простого равномерного хеширования. Например, мы можем потребовать, чтобы "близкие" в некотором смысле ключи давали далекие хеш-значения (это свойство особенно желательно при использовании линейного исследования, описанного в разделе 11.4). Универсальное хеширование, описанное в разделе 11.3.3, зачастую приводит к желаемым результатам.

Интерпретация ключей как целых неотрицательных чисел

Для большинства хеш-функций пространство ключей представляется множеством целых неотрицательных чисел $\mathbf{N} = \{0, 1, 2, \ldots\}$. Если же ключи не являются целыми неотрицательными числами, то можно найти способ их интерпретации как таковых. Например, строка символов может рассматриваться как целое число, записанное в соответствующей системе счисления. Так, идентификатор `pt` можно рассматривать как пару десятичных чисел $(112, 116)$, поскольку в ASCII-наборе символов `p` $= 112$ и `t` $= 116$. Рассматривая `pt` как число в системе счисления с основанием 128, мы находим, что оно соответствует значению $112 \cdot 128 + 116 = 14452$. В конкретных приложениях обычно не представляет особого труда разработать метод для представления ключей в виде (возможно, больших) целых чисел. Далее при изложении материала мы будем считать, что все ключи представляют целые неотрицательные числа.

11.3.1 Метод деления

Построение хеш-функции ***методом деления*** состоит в отображении ключа k в одну из ячеек путем получения остатка от деления k на m, т.е. хеш-функция имеет вид $h(k) = k \bmod m$.

Например, если хеш-таблица имеет размер $m = 12$, а значение ключа $k = 100$, то $h(k) = 4$. Поскольку для вычисления хеш-функции требуется только одна операция деления, хеширование методом деления считается достаточно быстрым.

При использовании данного метода мы обычно стараемся избегать некоторых значений m. Например, m не должно быть степенью 2, поскольку если $m = 2^p$, то $h(k)$ представляет собой просто p младших битов числа k. Если только заранее

не известно, что все наборы младших p битов ключей равновероятны, лучше строить хеш-функцию таким образом, чтобы ее результат зависел от всех битов ключа. В упражнении 11.3-3 требуется показать неудачность выбора $m = 2^p - 1$, когда ключи представляют собой строки символов, интерпретируемые как числа в системе счисления с основанием 2^p, поскольку перестановка символов ключа не приводит к изменению его хеш-значения.

Зачастую хорошие результаты можно получить, выбирая в качестве значения m простое число, достаточно далекое от степени двойки. Предположим, например, что мы хотим создать хеш-таблицу с разрешением коллизий методом цепочек для хранения $n = 2000$ символьных строк, размер символов в которых равен 8 битам. Нас устраивает проверка в среднем трех элементов при неудачном поиске, так что мы выбираем размер таблицы равным $m = 701$. Число 701 выбрано как простое число, близкое к величине $2000/3$ и не являющееся степенью 2. Рассматривая каждый ключ k как целое число, мы получаем искомую хеш-функцию:

$$h(k) = k \bmod 701.$$

11.3.2 Метод умножения

Построение хеш-функции ***методом умножения*** выполняется в два этапа. Сначала мы умножаем ключ k на константу $0 < A < 1$ и получаем дробную часть полученного произведения. Затем мы умножаем полученное значение на m и применяем к нему функцию "пол" т.е.

$$h(k) = \lfloor m(kA \bmod 1) \rfloor,$$

где выражение "$kA \bmod 1$" означает получение дробной части произведения kA, т.е. величину $kA - \lfloor kA \rfloor$.

Достоинство метода умножения заключается в том, что значение m перестает быть критичным. Обычно величина m из соображений удобства реализации функции выбирается равной степени 2. Пусть у нас имеется компьютер с размером слова w битов и k помещается в одно слово. Ограничим возможные значения константы A видом $s/2^w$, где s — целое число из диапазона $0 < s < 2^w$. Тогда мы сначала умножаем k на w-битовое целое число $s = A \cdot 2^w$. Результат представляет собой $2w$-битовое число $r_1 2^w + r_0$, где r_1 — старшее слово произведения, а r_0 — младшее. Старшие p битов числа r_0 представляют собой искомое p-битовое хеш-значение (рис. 11.4).

Хотя описанный метод работает с любыми значениями константы A, некоторые значения дают лучшие результаты по сравнению с другими. Оптимальный выбор зависит от характеристик хешируемых данных. В [185] Кнут предложил использовать дающее неплохие результаты значение

$$A \approx \left(\sqrt{5} - 1\right)/2 \approx 0.6180339887\ldots \tag{11.2}$$

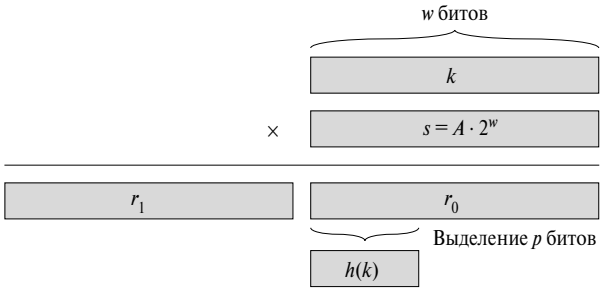


Рис. 11.4. Хеширование методом умножения

Возьмем в качестве примера $k = 123456$, $p = 14$, $m = 2^{14} = 16384$ и $w = 32$. Принимая предложение Кнута, выбираем значение A в виде $s/2^w$, ближайшее к величине $\left(\sqrt{5} - 1\right)/2$, так что $A = 2654435769/2^{32}$. Тогда

$$k \cdot s = 327\,706\,022\,297\,664 = \left(76\,300 \cdot 2^{32}\right) + 17\,612\,864,$$

и, соответственно, $r_1 = 76\,300$ и $r_0 = 17\,612\,864$. Старшие 14 битов числа r_0 дают нам хеш-значение $h\,(k) = 67$.

⋆ 11.3.3 Универсальное хеширование

Если недоброжелатель будет умышленно выбирать ключи для хеширования при помощи конкретной хеш-функции, то он сможет подобрать n значений, которые будут хешироваться в одну и ту же ячейку таблицы, приводя к среднему времени выборки $\Theta\,(n)$. Таким образом, любая фиксированная хеш-функция становится уязвимой, и единственный эффективный выход из ситуации — *случайный* выбор хеш-функции, *не зависящий* от того, с какими именно ключами ей предстоит работать. Такой подход, который называется ***универсальным хешированием***, гарантирует хорошую производительность в среднем, независимо от того, какие данные будут выбраны злоумышленником.

Главная идея универсального хеширования состоит в случайном выборе хеш-функции из некоторого тщательно отобранного класса функций в начале работы программы. Как и в случае быстрой сортировки, рандомизация гарантирует, что одни и те же входные данные не могут постоянно давать наихудшее поведение алгоритма. В силу рандомизации алгоритм будет работать всякий раз поразному, даже для одних и тех же входных данных, что гарантирует высокую среднюю производительность для любых входных данных. Возвращаясь к примеру с таблицей символов компилятора, мы обнаружим, что никакой выбор программистом имен идентификаторов не может привести к постоянному снижению производительности хеширования. Такое снижение возможно только тогда, когда компилятором выбрана случайная хеш-функция, которая приводит к плохому

хешированию конкретных входных данных; однако вероятность такой ситуации очень мала и одинакова для любого множества идентификаторов одного и то же размера.

Пусть $\mathcal{H}$ — конечное множество хеш-функций, которые отображают пространство ключей U в диапазон $\{0, 1, 2, \ldots, m-1\}$. Такое множество называется ***универсальным***, если для каждой пары различных ключей $k, l \in U$ количество хеш-функций $h \in \mathcal{H}$, для которых $h(k) = h(l)$, не превышает $|\mathcal{H}|/m$. Другими словами, при случайном выборе хеш-функции из $\mathcal{H}$ вероятность коллизии между различными ключами k и l не превышает вероятности совпадения двух случайным образом выбранных хеш-значений из множества $\{0, 1, 2, \ldots, m-1\}$, которая равна $1/m$.

Следующая теорема показывает, что универсальные хеш-функции обеспечивают хорошую среднюю производительность. В приведенной теореме n_i, как уже упоминалось, обозначает длину списка $T[i]$.

Теорема 11.3. Пусть хеш-функция h, выбранная из универсального множества хеш-функций, используется для хеширования n ключей в таблицу T размера m, с использованием для разрешения коллизий метода цепочек. Если ключ k отсутствует в таблице, то математическое ожидание $\mathrm{E}\left[n_{h(k)}\right]$ длины списка, в который хешируется ключ k, не превышает α. Если ключ k находится в таблице, то математическое ожидание $\mathrm{E}\left[n_{h(k)}\right]$ длины списка, в котором находится ключ k, не превышает $1 + \alpha$.

Доказательство. Заметим, что математическое ожидание вычисляется на множестве выборов функций и не зависит от каких бы то ни было предположений о распределении ключей. Определим для каждой пары различных ключей k и l индикаторную случайную величину $X_{kl} = \mathrm{I}\{h(k) = h(l)\}$. Поскольку по определению пара ключей вызывает коллизию с вероятностью не выше $1/m$, получаем, что $\Pr\{h(k) = h(l)\} \leqslant 1/m$, так что, в соответствии с леммой 5.1, $\mathrm{E}[X_{kl}] \leqslant 1/m$.

Далее для каждого ключа k определим случайную величину Y_k, которая равна количеству ключей, отличающихся от k и хешируемых в ту же ячейку, что и ключ k:

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl}.$$

Соответственно, получаем:

$$\begin{aligned}
\mathrm{E}\left[Y_k\right] &= \mathrm{E}\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] = \\
&= \sum_{\substack{l \in T \\ l \neq k}} \mathrm{E}\left[X_{kl}\right] \leqslant && \text{(в силу линейности математического ожидания)} \\
&\leqslant \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m}.
\end{aligned}$$

Оставшаяся часть доказательства зависит от того, находится ли ключ k в таблице T.

- Если $k \notin T$, то $n_{h(k)} = Y_k$ и $|\{l : l \in T \text{ и } l \neq k\}| = n$. Соответственно, $\mathrm{E}\left[n_{h(k)}\right] = \mathrm{E}\left[Y_k\right] \leqslant n/m = \alpha$.
- Если $k \in T$, то поскольку k находится в списке $T\left[h\left(k\right)\right]$ и значение Y_k не включает ключ k, мы имеем $n_{h(k)} = Y_k + 1$ и $|\{l : l \in T \text{ и } l \neq k\}| = n - 1$. Таким образом, $\mathrm{E}\left[n_{h(k)}\right] = \mathrm{E}\left[Y_k\right] + 1 \leqslant (n-1)/m + 1 = 1 + \alpha - 1/m < < 1 + \alpha$. ■

Следствие из данной теоремы гласит, что универсальное хеширование обеспечивает желаемый выигрыш: теперь невозможно выбрать последовательность операций, которые приведут к наихудшему времени работы. Путем рандомизации выбора хеш-функции в процессе работы программы гарантируется хорошее среднее время работы алгоритма для любых входных данных.

Следствие 11.4. Использование универсального хеширования и разрешения коллизий методом цепочек в хеш-таблице с m ячейками дает математическое ожидание времени выполнения любой последовательности из n вставок, поисков и удалений, в которой содержится $O\left(m\right)$ вставок, равное $\Theta\left(n\right)$.

Доказательство. Поскольку количество вставок равно $O\left(m\right)$, $n = O\left(m\right)$ и, соответственно, $\alpha = O\left(1\right)$. Время работы операций вставки и удаления — величина постоянная, а в соответствии с теоремой 11.3 математическое ожидание времени выполнения каждой операции поиска равно $O\left(1\right)$. Таким образом, используя свойство линейности математического ожидания, получаем, что ожидаемое время, необходимое для выполнения всей последовательности операций, равно $O\left(n\right)$. Поскольку каждая операция занимает время $\Omega\left(1\right)$, отсюда следует граница $\Theta\left(n\right)$. ■

Построение универсального множества хеш-функций

Построить такое множество довольно просто, что следует из теории чисел. Если вы с ней незнакомы, то можете сначала обратиться к главе 31.

Начнем с выбора простого числа p, достаточно большого, чтобы все возможные ключи находились в диапазоне от 0 до $p-1$ включительно. Пусть $\mathbf{Z}_p$ обозначает множество $\{0,1,\ldots,p-1\}$, а $\mathbf{Z}_p^*$ — множество $\{1,2,\ldots,p-1\}$. Поскольку p — простое число, мы можем решать уравнения по модулю p при помощи методов, описанных в главе 31. Из предположения о том, что пространство ключей больше, чем количество ячеек в хеш-таблице, следует, что $p > m$.

Теперь определим хеш-функцию $h_{a,b}$ для любых $a \in \mathbf{Z}_p^*$ и $b \in \mathbf{Z}_p$ следующим образом:

$$h_{a,b}(k) = ((ak+b) \bmod p) \bmod m. \tag{11.3}$$

Например, при $p = 17$ и $m = 6$ $h_{3,4}(8) = 5$. Семейство всех таких функций образует множество

$$\mathcal{H}_{p,m} = \left\{h_{a,b} : a \in \mathbf{Z}_p^* \text{ и } b \in \mathbf{Z}_p\right\}. \tag{11.4}$$

Каждая хеш-функция $h_{a,b}$ отображает $\mathbf{Z}_p$ на $\mathbf{Z}_m$. Этот класс хеш-функций обладает тем свойством, что размер m выходного диапазона произволен и не обязательно представляет собой простое число. Это свойство будет использовано нами в разделе 11.5. Поскольку число a можно выбрать $p-1$ способом, и p способами — число b, всего во множестве $\mathcal{H}_{p,m}$ содержится $p(p-1)$ хеш-функций.

Теорема 11.5. Множество хеш-функций $\mathcal{H}_{p,m}$, определяемое уравнениями (11.3) и (11.4), является универсальным.

Доказательство. Рассмотрим два различных ключа k и l из $\mathbf{Z}_p$, т.е. $k \neq l$. Пусть для данной хеш-функции $h_{a,b}$

$$r = (ak+b) \bmod p,$$
$$s = (al+b) \bmod p.$$

Заметим, что $r \neq s$. Почему? Рассмотрим разность

$$r - s \equiv a(k-l) \pmod{p}.$$

Поскольку p — простое число, причем как a, так и $(k-l)$ не равны нулю по модулю p, то отсюда следует что $r \neq s$, так что и их произведение также должно быть отлично от нуля по модулю p согласно теореме 31.6. Следовательно, вычисление любой хеш-функции $h_{a,b} \in \mathcal{H}_{p,m}$ для различных ключей k и l приводит к различным хеш-значениям r и s по модулю p. Таким образом, коллизии "по модулю p" отсутствуют. Более того, каждая из $p(p-1)$ возможных пар (a,b), в которых

$a \neq 0$, приводят к *различным* парам (r, s), в которых $r \neq s$. Чтобы убедиться в этом, достаточно рассмотреть возможность однозначного определения a и b по данным r и s:

$$a = \left((r - s)\left((k - l)^{-1} \bmod p\right)\right) \bmod p,$$
$$b = (r - ak) \bmod p,$$

где $\left((k - l)^{-1} \bmod p\right)$ обозначает единственное мультипликативное обратное по модулю p значения $k - l$. Поскольку имеется только $p(p-1)$ возможных пар (r, s), таких что $r \neq s$, то имеется взаимнооднозначное соответствие между парами (a, b), где $a \neq 0$, и парами (r, s), в которых $r \neq s$. Таким образом, для любой данной пары входных значений k и l при равномерном случайном выборе пары (a, b) из $\mathbf{Z}_p^* \times \mathbf{Z}_p$, получаемая в результате пара (r, s) может быть с равной вероятностью любой из пар с отличающимися значениями по модулю p.

Отсюда можно заключить, что вероятность того, что различные ключи k и l приводят к коллизии, равна вероятности того, что $r \equiv s \pmod{m}$ при произвольном выборе отличающихся по модулю p значений r и s. Для данного значения r имеется $p - 1$ возможное значение s. При этом число значений s, таких что $s \neq r$ и $s \equiv r \pmod{p}$, не превышает

$$\lceil p/m \rceil - 1 \leqslant ((p + m - 1)/m) - 1 = (p - 1)/m.$$

(Здесь использовано неравенство (3.6).) Вероятность того, что s приводит к коллизии с r при приведении по модулю m, не превышает

$$((p-1)/m)(p-1) = 1/(p-1) = 1/m.$$

Следовательно, для любой пары различных значений $k, l \in \mathbf{Z}_p$

$$\Pr\{h_{a,b}(k) = h_{a,b}(l)\} \leqslant 1/m,$$

так что множество хеш-функций $\mathcal{H}_{p,m}$ является универсальным. ■

Упражнения

11.3-1. Предположим, что мы выполняем поиск в связанном списке длиной n, в котором каждый элемент содержит ключ k вместе с хеш-значением $h(k)$. Каждый ключ представляет собой длинную символьную строку. Как можно использовать наличие хеш-значения при поиске элемента с заданным ключом?

11.3-2. Предположим, что строка из r символов хешируется в m ячеек путем ее интерпретации как числа, записанного в 128-ричной системе счисления, и использования метода деления. Число m легко представимо в виде 32-битового машинного слова, но представление строки как целого числа требует много слов. Каким образом можно применить метод деления для вычисления хеш-значения символьной строки с использованием фиксированного количества дополнительных машинных слов?

11.3-3. Рассмотрим версию метода деления, в которой $h(k) = k \bmod m$, где $m = 2^p - 1$, а k — символьная строка, интерпретируемая как целое число в системе счисления с основанием 2^p. Покажите, что если строка x может быть получена из строки y перестановкой символов, то хеш-значения этих строк одинаковы. Приведите пример приложения, где это свойство хеш-функции может оказаться крайне нежелательным.

11.3-4. Рассмотрим хеш-таблицу размером $m = 1000$ и соответствующую хеш-функцию $h(k) = \lfloor m(kA \bmod 1) \rfloor$ для $A = (\sqrt{5} - 1)/2$. Вычислите номера ячеек, в которые хешируются ключи 61, 62, 63, 64 и 65.

⋆ 11.3-5. Определим семейство хеш-функций $\mathcal{H}$, отображающих конечное множество U на конечное множество B как ***ε-универсальное***, если для всех пар различных элементов k и l из U

$$\Pr\{h(k) = h(l)\} \leqslant \varepsilon,$$

где вероятность вычисляется для случайного выбора хеш-функции h из множества $\mathcal{H}$. Покажите, ε-универсальное семейство хеш-функций должно обладать тем свойством, что

$$\varepsilon \geqslant \frac{1}{|B|} - \frac{1}{|U|}.$$

⋆ 11.3-6. Пусть U — множество наборов из n чисел, выбирающихся из $\mathbf{Z}_p$, и пусть $B = \mathbf{Z}_p$, где p — простое число. Определим хеш-функцию $h_b : U \to B$ для $b \in \mathbf{Z}_p$ и входного набора $\langle a_0, a_1, \ldots, a_{n-1} \rangle \in U$ следующим образом:

$$h_b(\langle a_0, a_1, \ldots, a_{n-1} \rangle) = \sum_{j=0}^{n-1} a_j b^j,$$

и пусть $\mathcal{H} = \{h_b : b \in \mathbf{Z}_p\}$. Докажите, что $\mathcal{H}$ является $((n-1)/p)$-универсальным множеством в соответствии с определением, данным в упражнении 11.3-5. (*Указание*: см. упражнение 31.4-4.)

11.4 Открытая адресация

При использовании метода ***открытой адресации*** все элементы хранятся непосредственно в хеш-таблице, т.е. каждая запись таблицы содержит либо элемент динамического множества, либо значение NIL. При поиске элемента мы систематически проверяем ячейки таблицы до тех пор, пока не найдем искомый элемент или пока не убедимся в его отсутствии в таблице. Здесь, в отличие от метода цепочек, нет ни списков, ни элементов, хранящихся вне таблицы. Таким образом, в методе открытой адресации хеш-таблица может оказаться заполненной, делая невозможной вставку новых элементов; коэффициент заполнения α не может превышать 1.

Конечно, при хешировании с разрешением коллизий методом цепочек можно использовать свободные места в хеш-таблице для хранения связанных списков (см. упражнение 11.2-4), но преимущество открытой адресации заключается в том, что она позволяет полностью отказаться от указателей. Вместо того чтобы следовать по указателям, мы *вычисляем* последовательность проверяемых ячеек. Дополнительная память, освобождающаяся в результате отказа от указателей, позволяет использовать хеш-таблицы большего размера при том же общем количестве памяти, потенциально приводя к меньшему количеству коллизий и более быстрой выборке.

Для выполнения вставки при открытой адресации мы последовательно проверяем, или ***исследуем*** (probe), ячейки хеш-таблицы до тех пор, пока не находим пустую ячейку, в которую помещаем вставляемый ключ. Вместо фиксированного порядка исследования ячеек $0, 1, \ldots, m-1$ (для чего требуется $\Theta(n)$ времени), последовательность исследуемых ячеек *зависит от вставляемого в таблицу ключа*. Для определения исследуемых ячеек мы расширим хеш-функцию, включив в нее в качестве второго аргумента номер исследования (начинающийся с 0). В результате хеш-функция становится следующей:

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\} .$$

В методе открытой адресации требуется, чтобы для каждого ключа k ***последовательность исследований***

$$\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$$

представляла собой перестановку множества $\langle 0, 1, \ldots, m-1 \rangle$, чтобы в конечном счете могли быть просмотрены все ячейки хеш-таблицы. В приведенном далее псевдокоде предполагается, что элементы в таблице T представляют собой ключи без сопутствующей информации; ключ k тождественен элементу, содержащему ключ k. Каждая ячейка содержит либо ключ, либо значение NIL (если она не заполнена):

```
HASH_INSERT(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3          if T[j] = NIL
4            then T[j] ← k
5                 return j
6            else  i ← i + 1
7     until i = m
8  error "Хеш-таблица переполнена"
```

Алгоритм поиска ключа k исследует ту же последовательность ячеек, что и алгоритм вставки ключа k. Таким образом, если при поиске встречается пустая ячейка, поиск завершается неуспешно, поскольку ключ k должен был бы быть вставлен в эту ячейку в последовательности исследований, и никак не позже нее. (Мы предполагаем, что удалений из хеш-таблицы не было.) Процедура HASH_SEARCH получает в качестве входных параметров хеш-таблицу T и ключ k и возвращает номер ячейки, которая содержит ключ k (или значение NIL, если ключ в хеш-таблице не обнаружен):

```
HASH_SEARCH(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3          if T[j] = k
4            then return j
5          i ← i + 1
6     until T[j] = NIL или i = m
7  return NIL
```

Процедура удаления из хеш-таблицы с открытой адресацией достаточно сложна. При удалении ключа из ячейки i мы не можем просто пометить ее значением NIL. Поступив так, мы можем сделать невозможным выборку ключа k, в процессе вставки которого исследовалась и оказалась занятой ячейка i. Одно из решений состоит в том, чтобы помечать такие ячейки специальным значением DELETED вместо NIL. При этом мы должны слегка изменить процедуру HASH_INSERT, с тем чтобы она рассматривала такую ячейку как пустую и могла вставить в нее новый ключ. В процедуре HASH_SEARCH никакие изменения не требуются, поскольку мы просто пропускаем такие ячейки при поиске и исследуем следующие ячейки в последовательности. Однако при использовании специального значения DELETED время поиска перестает зависеть от коэффициента заполнения α, и по этой причине, как правило, при необходимости удалений из хеш-таблицы в качестве метода разрешения коллизий выбирается метод цепочек.

В нашем дальнейшем анализе мы будем исходить из предположения ***равномерного хеширования***, т.е. мы предполагаем, что для каждого ключа в качестве последовательности исследований равновероятны все $m!$ перестановок множества $\{0, 1, \ldots, m-1\}$. Равномерное хеширование представляет собой обобщение определенного ранее простого равномерного хеширования, заключающееся в том, что теперь хеш-функция дает не одно значение, а целую последовательность исследований. Реализация истинно равномерного хеширования достаточно трудна, однако на практике используются подходящие аппроксимации (такие, например, как определенное ниже двойное хеширование).

Для вычисления последовательности исследований для открытой адресации обычно используются три метода: линейное исследование, квадратичное исследование и двойное хеширование. Эти методы гарантируют, что для каждого ключа k $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ является перестановкой $\langle 0, 1, \ldots, m-1\rangle$. Однако эти методы не удовлетворяют предположению о равномерном хешировании, так как ни один из них не в состоянии сгенерировать более m^2 различных последовательностей исследований (вместо $m!$, требующихся для равномерного хеширования). Наибольшее количество последовательностей исследований дает двойное хеширование и, как и следовало ожидать, дает наилучшие результаты.

Линейное исследование

Пусть задана обычная хеш-функция $h' : U \to \{0, 1, \ldots, m-1\}$, которую мы будем в дальнейшем именовать ***вспомогательной хеш-функцией*** (auxiliary hash function). Метод ***линейного исследования*** для вычисления последовательности исследований использует хеш-функцию

$$h(k,i) = \left(h'(k) + i\right) \bmod m,$$

где i принимает значения от 0 до $m-1$ включительно. Для данного ключа k первой исследуемой ячейкой является $T[h'(k)]$, т.е. ячейка, которую дает вспомогательная хеш-функция. Далее мы исследуем ячейку $T[h'(k)+1]$ и далее последовательно все до ячейки $T[m-1]$, после чего переходим в начало таблицы и последовательно исследуем ячейки $T[0]$, $T[1]$, и так до ячейки $T[h'(k)-1]$. Поскольку начальная исследуемая ячейка однозначно определяет всю последовательность исследований целиком, всего имеется m различных последовательностей.

Линейное исследование легко реализуется, однако с ним связана проблема ***первичной кластеризации***, связанной с созданием длинных последовательностей занятых ячеек, что, само собой разумеется, увеличивает среднее время поиска. Кластеры возникают в связи с тем, что вероятность заполнения пустой ячейки, которой предшествуют i заполненных ячеек, равна $(i+1)/m$. Таким образом, длинные серии заполненных ячеек имеют тенденцию ко все большему удлинению, что приводит к увеличению среднего времени поиска.

Квадратичное исследование

Квадратичное исследование использует хеш-функцию вида

$$h(k,i) = \left(h'(k) + c_1 i + c_2 i^2\right) \bmod m, \tag{11.5}$$

где h' — вспомогательная хеш-функция, c_1 и $c_2 \neq 0$ — вспомогательные константы, а i принимает значения от 0 до $m-1$ включительно. Начальная исследуемая ячейка — $T[h'(k)]$; остальные исследуемые позиции смещены относительно нее на величины, которые описываются квадратичной зависимостью от номера исследования i. Этот метод работает существенно лучше линейного исследования, но для того, чтобы исследование охватывало все ячейки, необходим выбор специальных значений c_1, c_2 и m (в задаче 11-3 показан один из путей выбора этих параметров). Кроме того, если два ключа имеют одну и то же начальную позицию исследования, то одинаковы и последовательности исследования в целом, так как из $h_1(k,0) = h_2(k,0)$ вытекает $h_1(k,i) = h_2(k,i)$. Это свойство приводит к более мягкой ***вторичной кластеризации***. Как и в случае линейного исследования, начальная ячейка определяет всю последовательность, поэтому всего используется m различных последовательностей исследования.

Двойное хеширование

Двойное хеширование представляет собой один из наилучших способов использования открытой адресации, поскольку получаемые при этом перестановки обладают многими характеристиками случайно выбираемых перестановок. ***Двойное хеширование*** использует хеш-функцию вида

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m,$$

где h_1 и h_2 — вспомогательные хеш-функции. Начальное исследование выполняется в позиции $T[h_1(k)]$, а смещение каждой из последующих исследуемых ячеек относительно предыдущей равно $h_2(k)$ по модулю m. Следовательно, в отличие от линейного и квадратичного исследования, в данном случае последовательность исследования зависит от ключа k по двум параметрам — в плане выбора начальной исследуемой ячейки и расстояния между соседними исследуемыми ячейками, так как оба эти параметра зависят от значения ключа.

На рис. 11.5 показан пример вставки при двойном хешировании. Вы видите хеш-таблицу размером 13 ячеек, в которой используются вспомогательные хеш-функции $h_1(k) = k \bmod 13$ и $h_2(k) = 1 + (k \bmod 11)$.Так как $14 \equiv 1 \pmod{13}$ и $14 \equiv 3 \pmod{11}$, ключ 14 вставляется в пустую ячейку 9, после того как при исследовании ячеек 1 и 5 выясняется, что эти ячейки заняты.

Для того чтобы последовательность исследования могла охватить всю таблицу, значение $h_2(k)$ должно быть взаимно простым с размером хеш-таблицы

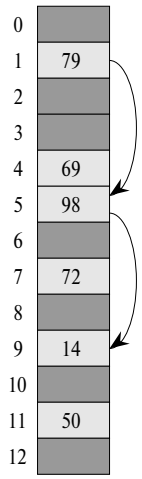


Рис. 11.5. Вставка при двойном хешировании

m (см. упражнение 11.4-3). Удобный способ обеспечить выполнение этого условия состоит в выборе числа m, равного степени 2, и разработке хеш-функции h_2 таким образом, чтобы она возвращала только нечетные значения. Еще один способ состоит в использовании в качестве m простого числа и построении хеш-функции h_2 такой, чтобы она всегда возвращала натуральные числа, меньшие m. Например, можно выбрать простое число в качестве m, а хеш-функции такими:

$$h_1(k) = k \bmod m,$$
$$h_2(k) = 1 + \left(k \bmod m'\right),$$

где m' должно быть немного меньше m (например, $m - 1$). Скажем, если $k = 123456$, $m = 701$, а $m' = 700$, то $h_1(k) = 80$ и $h_2(k) = 257$, так что первой исследуемой будет ячейка в 80-й позиции, а затем будет исследоваться каждая 257-я (по модулю m) ячейка, пока не будет обнаружена пустая ячейка, или пока не окажутся исследованы все ячейки таблицы.

Двойное хеширование превосходит линейное или квадратичное исследования в смысле количества $\Theta\left(m^2\right)$ последовательностей исследований, в то время как у упомянутых методов это количество равно $\Theta(m)$. Это связано с тем, что каждая возможная пара $(h_1(k), h_2(k))$ дает свою, отличающуюся от других последовательность исследований. В результате производительность двойного хеширования достаточно близка к производительности "идеальной" схемы равномерного хеширования.

Анализ хеширования с открытой адресацией

Анализ открытой адресации, как и анализ метода цепочек, выполняется с использованием коэффициента заполнения $\alpha = n/m$ хеш-таблицы при n и m, стремящихся к бесконечности. Само собой разумеется, при использовании открытой адресации у нас может быть не более одного элемента на ячейку таблицы, так что $n \leqslant m$ и, следовательно, $\alpha \leqslant 1$.

Будем считать, что используется равномерное хеширование. При такой идеализированной схеме последовательность исследований $\langle h(k,0), h(k,1), \ldots, h(k, m-1)\rangle$, используемая для вставки или поиска каждого ключа k, с равной вероятностью является одной из возможных перестановок $\langle 0, 1, \ldots, m-1\rangle$. Разумеется, с каждым конкретным ключом связана единственная фиксированная последовательность исследований, так что при рассмотрении распределения вероятностей ключей и хеш-функций все последовательности исследований оказываются равновероятными.

Сейчас мы проанализируем математическое ожидание количества исследований для хеширования с открытой адресацией в предположении равномерного хеширования, и начнем с анализа количества исследований в случае неуспешного поиска.

Теорема 11.6. Математическое ожидание количества исследований при неуспешном поиске в хеш-таблице с открытой адресацией и коэффициентом заполнения $\alpha = n/m < 1$ в предположении равномерного хеширования не превышает $1/(1-\alpha)$.

Доказательство. При неуспешном поиске каждая последовательность исследований завершается на пустой ячейке. Определим случайную величину X как равную количеству исследований, выполненных при неуспешном поиске, и события A_i $(i = 1, 2, \ldots)$, заключающиеся в том, что было выполнено i-е исследование, и оно пришлось на занятую ячейку. Тогда событие $\{X \geqslant i\}$ представляет собой пересечение событий $A_1 \cap A_2 \cap \ldots \cap A_{i-1}$. Ограничим вероятность $\Pr\{X \geqslant i\}$ путем ограничения вероятности $\Pr\{A_1 \cap A_2 \cap \ldots \cap A_{i-1}\}$. В соответствии с упражнением B.2-6,

$$\Pr\{A_1 \cap A_2 \cap \ldots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \ldots \\ \ldots \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \ldots \cap A_{i-2}\}.$$

Поскольку всего имеется n элементов и m ячеек, $\Pr\{A_1\} = n/m$. Вероятность того, что будет выполнено j-е исследование $(j > 1)$ и что оно будет проведено над заполненной ячейкой (при этом первые $j-1$ исследований проведены над заполненными ячейками), равна $(n-j+1)/(m-j+1)$. Эта вероятность определяется следующим образом: мы должны проверить один из оставшихся

$(n-(j-1))$ элементов, а всего неисследованных к этому времени ячеек остается $(m-(j-1))$. В соответствии с предположением о равномерном хешировании, вероятность равна отношению этих величин. Воспользовавшись тем фактом, что из $n < m$ для всех $0 \leqslant j < m$ следует соотношение $(n-j)/(m-j) \leqslant n/m$, для всех $1 \leqslant i \leqslant m$ мы получаем:

$$\Pr\{X \geqslant i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \leqslant \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}.$$

Теперь мы можем использовать уравнение (B.24) для того, чтобы ограничить математическое ожидание количества исследований:

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr\{X \geqslant i\} \leqslant \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}.$$

■

Полученная граница $1 + \alpha + \alpha^2 + \alpha^3 + \ldots$ имеет интуитивную интерпретацию. Одно исследование выполняется всегда. С вероятностью, приблизительно равной α, первое исследование проводится над заполненной ячейкой, и требуется выполнение второго исследования. С вероятностью, приблизительно равной α^2, две первые ячейки оказываются заполненными и требуется проведение третьего исследования, и т.д.

Если α — константа, то теорема 11.6 предсказывает, что неуспешный поиск выполняется за время $O(1)$. Например, если хеш-таблица заполнена наполовину, то среднее количество исследований при неуспешном поиске не превышает $1/(1-0.5) = 2$. При заполненности хеш-таблицы на 90% среднее количество исследований не превышает $1/(1-0.9) = 10$.

Теорема 11.6 практически непосредственно дает нам оценку производительности процедуры HASH_INSERT.

Следствие 11.7. Вставка элемента в хеш-таблицу с открытой адресацией и коэффициентом заполнения α в предположении равномерного хеширования, требует в среднем не более $1/(1-\alpha)$ исследований.

Доказательство. Элемент может быть вставлен в хеш-таблицу только в том случае, если в ней есть свободное место, так что $\alpha < 1$. Вставка ключа требует проведения неуспешного поиска, за которым следует размещение ключа в найденной пустой ячейке. Следовательно, математическое ожидание количества исследований не превышает $1/(1-\alpha)$. ■

Вычисление математического ожидания количества исследований при успешном поиске требует немного больше усилий.

Теорема 11.8. Математическое ожидание количества исследований при удачном поиске в хеш-таблице с открытой адресацией и коэффициентом заполнения $\alpha < 1$, в предположении равномерного хеширования и равновероятного поиска любого из ключей, не превышает

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}.$$

Доказательство. Поиск ключа k выполняется с той же последовательностью исследований, что и его вставка. В соответствии со следствием 11.7, если k был $(i+1)$-м ключом, вставленным в хеш-таблицу, то математическое ожидание количества проб при поиске k не превышает $1/(1-i/m) = m/(m-i)$. Усреднение по всем n ключам в хеш-таблице дает нам среднее количество исследований при удачном поиске:

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{1}{\alpha} (H_m - H_{m-n}),$$

где $H_i = \sum_{j=1}^{i} 1/j$ представляет собой i-е гармоническое число (определяемое уравнением (A.7)). Воспользовавшись методом ограничения суммы интегралом, описанном в разделе A.2 (неравенство (A.12)), получаем верхнюю границу математического ожидания количества исследований при удачном поиске:

$$\begin{aligned} \frac{1}{\alpha} (H_m - H_{m-n}) = \frac{1}{\alpha} \sum_{k=m-n+1}^{m} \frac{1}{k} \leqslant \frac{1}{\alpha} \int_{m-n}^{m} (1/x)\, dx = \\ = \frac{1}{\alpha} \ln \frac{m}{m-n} = \frac{1}{\alpha} \ln \frac{1}{1-\alpha}. \end{aligned}$$

■

Если хеш-таблица заполнена наполовину, ожидаемое количество исследований при успешном поиске не превышает 1.387; при заполненности на 90% это количество не превышает 2.559.

Упражнения

11.4-1. Рассмотрите вставку ключей 10, 22, 31, 4, 15, 28, 17, 88, 59 в хеш-таблицу длины $m = 11$ с открытой адресацией и вспомогательной хеш-функцией $h'(k) = k \bmod m$. Проиллюстрируйте результат вставки приведенного списка ключей при использовании линейного исследования, квадратичного исследования с $c_1 = 1$ и $c_2 = 3$, и двойного хеширования с $h_2(k) = 1 + (k \bmod (m-1))$.

11.4-2. Запишите псевдокод процедуры HASH_DELETE, описанной в тексте, и модифицируйте процедуру HASH_INSERT для обработки специального значения DELETED (удален).

⋆ 11.4-3. Предположим, что для разрешения коллизий мы используем двойное хеширование, т.е. хеш-функцию $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$. Покажите, что если m и $h_2(k)$ для некоторого ключа k имеют наибольший общий делитель $d \geqslant 1$, то неуспешный поиск ключа k проверяет $(1/d)$-ю часть хеш-таблицы до того, как возвращается в ячейку $h_1(k)$. Таким образом, когда $d = 1$, т.е. m и $h_2(k)$ взаимно простые числа, поиск может исследовать всю хеш-таблицу. (*Указание*: см. главу 31).

11.4-4. Пусть у нас есть хеш-таблица с открытой адресацией в предположении равномерного хеширования. Оцените верхнюю границу математического ожидания количества исследований при неуспешном поиске и при удачном поиске для коэффициентов заполнения таблицы $3/4$ и $7/8$.

⋆ 11.4-5. Пусть у нас есть хеш-таблица с открытой адресацией и коэффициентом заполнения α. Найдите ненулевое значение α, при котором математическое ожидание количества исследований в случае неуспешного поиска в два раза превышает математическое ожидание количества исследований в случае удачного поиска. Воспользуйтесь для решения поставленной задачи границами, приведенными в теоремах 11.6 и 11.8.

⋆ 11.5 Идеальное хеширование

Хотя чаще всего хеширование используется из-за превосходной *средней* производительности, возможна ситуация, когда реально получить превосходную производительность хеширования в *наихудшем* случае. Такой ситуацией является ***статическое*** множество ключей, т.е. после того как все ключи сохранены в таблице, их множество никогда не изменяется. Ряд приложений в силу своей природы работает со статическими множествами ключей. В качестве примера можно привести множество зарезервированных слов языка программирования или множество имен файлов на компакт-диске. ***Идеальным хешированием*** мы называем методику, которая в наихудшем случае выполняет поиск за $O(1)$ обращений к памяти.

Основная идея идеального хеширования достаточно проста. Мы используем двухуровневую схему хеширования с универсальным хешированием на каждом уровне (см. рис. 11.6).

Первый уровень по сути тот же, что и в случае хеширования с цепочками: n ключей хешируются в m ячеек с использованием хеш-функции h, тщательно выбранной из семейства универсальных хеш-функций.

Однако вместо того, чтобы создавать список ключей, хешированных в ячейку j, мы используем маленькую ***вторичную хеш-таблицу*** S_j со своей хеш-функцией h_j. Путем точного выбора хеш-функции h_j мы можем гарантировать отсутствие коллизий на втором уровне.

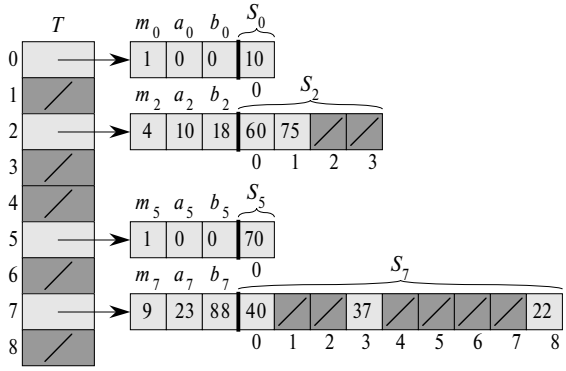


Рис. 11.6. Использование идеального хеширования для хранения множества $K = \{10, 22, 37, 40, 60, 70, 75\}$

Рассмотрим детальнее пример на рис. 11.6, где показано сохранение статического множества ключей $K = \{10, 22, 37, 40, 60, 70, 75\}$ в хеш-таблице с использованием технологии идеального хеширования. Внешняя хеш-функция имеет вид $h(k) = ((ak + b) \bmod p) \bmod m$, где $a = 3$, $b = 42$, $p = 101$ и $m = 9$. Например, $h(75) = 2$, так что ключ 75 хешируется в ячейку 2. Вторичная хеш-таблица S_j хранит все ключи, хешированные в ячейку j. Размер каждой таблицы S_j равен m_j, и с ней связана хеш-функция $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Поскольку $h_2(75) = 1$, ключ 75 хранится в ячейке 1 вторичной хеш-таблицы S_2. Ни в одной из вторичных таблиц нет ни одной коллизии, так что время поиска в худшем случае равно константе.

Для того чтобы гарантировать отсутствие коллизий на втором уровне, требуется, чтобы размер m_j хеш-таблицы S_j был равен квадрату числа n_j ключей, хешированных в ячейку j. Такая квадратичная зависимость m_j от n_j может показаться чрезмерно расточительной, однако далее мы покажем, что при корректном выборе хеш-функции первого уровня ожидаемое количество требуемой для хеш-таблицы памяти остается равным $O(n)$.

Мы выберем хеш-функцию из универсальных множеств хеш-функций, описанных в разделе 11.3.3. Хеш-функция первого уровня выбирается из множества $\mathcal{H}_{p,m}$, где, как и в разделе 11.3.3, p является простым числом, превышающим значение любого из ключей. Ключи, хешированные в ячейку j, затем повторно хешируются во вторичную хеш-таблицу S_j размером m_j с использованием хеш-функции h_j, выбранной из класса $\mathcal{H}_{p,m_j}$[1].

[1]При $n_j = m_j = 1$ для ячейки j хеш-функция не нужна; при выборе хеш-функции $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m_j$ для такой ячейки мы просто выбираем $a = b = 0$.

Работа будет выполнена в два этапа. Сначала мы выясним, как гарантировать отсутствие коллизий во вторичной таблице. Затем мы покажем, что ожидаемое количество памяти, необходимой для первичной и вторичной хеш-таблиц, равно $O(n)$.

Теорема 11.9. Если n ключей сохраняются в хеш-таблице размером $m = n^2$ с использованием хеш-функции h, случайно выбранной из универсального множества хеш-функций, то вероятность возникновения коллизий не превышает $1/2$.

Доказательство. Всего имеется $\binom{n}{2}$ пар ключей, которые могут вызвать коллизию. Если хеш-функция выбрана случайным образом из универсального семейства хеш-функций $\mathcal{H}$, то для каждой пары вероятность возникновения коллизии равна $1/m$. Пусть X — случайная величина, которая подсчитывает количество коллизий. Если $m = n^2$, то математическое ожидание числа коллизий равно

$$\mathrm{E}[X] = \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n^2 - n}{2} \cdot \frac{1}{n^2} < \frac{1}{2}.$$

(Обратите внимание на схожесть данного анализа с анализом парадокса дней рождения из раздела 5.4.1.) Применение неравенства Маркова (B.29), $\Pr\{X \geqslant t\} \leqslant \leqslant \mathrm{E}[X]/t$, при $t = 1$ завершает доказательство. ■

В ситуации, описанной в теореме 11.9, когда $m = n^2$, произвольно выбранная из множества $\mathcal{H}$ хеш-функция с большей вероятностью не приведет к коллизиям, чем приведет к ним. Для данного множества K, содержащего n ключей (напомним, что K — статическое множество), найти хеш-функцию h, не дающую коллизий, возможно после нескольких случайных попыток.

Если значение n велико, таблица размера $m = n^2$ оказывается слишком большой и приводит к ненужному перерасходу памяти. Тогда мы принимаем двухуровневую схему хеширования. Хеш-функция h первого уровня используется для хеширования ключей в $m = n$ ячеек. Затем, если в ячейку j оказывается хешировано n_j ключей, для того чтобы обеспечить отсутствие коллизий, используется вторичная хеш-таблица S_j размером $m_j = n_j^2$.

Вернемся к вопросу необходимого для описанной схемы количества памяти. Поскольку размер m_j j-ой вторичной хеш-таблицы растет с ростом n_j квадратично, возникает риск, что в целом потребуется очень большое количество памяти.

Если хеш-таблица первого уровня имеет размер $m = n$, то, естественно, нам потребуется количество памяти, равное $O(n)$, для первичной хеш-таблицы, а также для хранения размеров m_j вторичных хеш-таблиц и параметров a_j и b_j, определяющих вторичные хеш-функции h_j (выбираемые из множества $\mathcal{H}_{p,m_j}$ из раздела 11.3.3 (за исключением случая, когда $n_j = 1$; в этом случае мы просто принимаем $a = b = 0$)). Следующая теорема и следствия из нее позволяют нам вычислить границу суммарного размера всех вторичных таблиц.

Теорема 11.10. Если мы сохраняем n ключей в хеш-таблице размером $m = n$ с использованием хеш-функции h, выбираемой случайным образом из универсального множества хеш-функций, то

$$\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n,$$

где n_j — количество ключей, хешированных в ячейку j.

Доказательство. Начнем со следующего тождества, которое справедливо для любого неотрицательного целого a:

$$a^2 = a + 2\binom{a}{2}. \tag{11.6}$$

Итак, мы имеем:

$$\begin{aligned}
&\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] = \\
&= \mathrm{E}\left[\sum_{j=0}^{m-1} \left(n_j + 2\binom{n_j}{2}\right)\right] = && \text{(в соответствии с (11.6))} \\
&= \mathrm{E}\left[\sum_{j=0}^{m-1} n_j\right] + 2\mathrm{E}\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] = && \text{(в силу линейности математического ожидания)} \\
&= \mathrm{E}\left[n\right] + 2\mathrm{E}\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] = && \text{(в соответствии с (11.1))} \\
&= n + 2\mathrm{E}\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] && \text{(поскольку } n \text{ не является случайной переменной) .}
\end{aligned}$$

Для того чтобы вычислить сумму $\sum_{j=0}^{m-1} \binom{n_j}{2}$, заметим, что это просто общее количество коллизий. В соответствии со свойством универсального хеширования, математическое ожидание значения этой суммы не превышает

$$\binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2},$$

так как $m = n$. Таким образом,

$$\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] \leqslant n + 2\frac{n-1}{2} = 2n - 1 < 2n.$$

■

Следствие 11.11. Если мы сохраняем n ключей в хеш-таблице размером $m = n$ с использованием хеш-функции h, выбираемой случайным образом из универсального множества хеш-функций, и устанавливаем размер каждой вторичной хеш-таблицы равным $m_j = n_j^2$ ($j = 0, 1, \ldots, m-1$), то математическое ожидание количества необходимой для вторичных хеш-таблиц в схеме идеального хеширования памяти не превышает величины $2n$.

Доказательство. Поскольку $m_j = n_j^2$ для $j = 0, 1, \ldots, m-1$, согласно теореме 11.10 мы получаем:

$$\mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right] = \mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n, \tag{11.7}$$

что и требовалось доказать. ■

Следствие 11.12. Если мы сохраняем n ключей в хеш-таблице размером $m = n$ с использованием хеш-функции h, выбираемой случайным образом из универсального множества хеш-функций, и устанавливаем размер каждой вторичной хеш-таблицы равным $m_j = n_j^2$ ($j = 0, 1, \ldots, m-1$), то вероятность того, что общее количество необходимой для вторичных хеш-таблиц памяти не менее $4n$, меньше чем $1/2$.

Доказательство. Вновь применим неравенство Маркова (B.29), $\Pr\{X \geqslant t\} \leqslant \mathrm{E}[X]/t$, на этот раз к неравенству (11.7) (здесь $X = \sum_{j=0}^{m-1} m_j$ и $t = 4n$):

$$\Pr\left\{\sum_{j=0}^{m-1} m_j \geqslant 4n\right\} \leqslant \frac{\mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right]}{4n} < \frac{2n}{4n} = \frac{1}{2}.$$

■

Упражнения

⋆ 11.5-1. Предположим, что мы вставляем n ключей в хеш-таблицу размером m с использованием открытой адресации и равномерного хеширования. Обозначим через $p(n, m)$ вероятность отсутствия коллизий. Покажите, что $p(n, m) \leqslant e^{-n(n-1)/2m}$. (*Указание*: см. уравнение (3.11).) Докажите, что при n, превосходящем $\sqrt{m}$, вероятность избежать коллизий быстро падает до нуля.

Задачи

11-1. **Наибольшее число исследований при хешировании**

Имеется хеш-таблица размером m с открытой адресацией, в которой сохраняется n ключей, причем $n \leqslant m/2$.

а) Покажите, что в предположении равномерного хеширования вероятность того, что для любого $i = 1, 2, \ldots, n$ для вставки i-го ключа требуется более k исследований, не превышает 2^{-k}.

б) Покажите, что для любого $i = 1, 2, \ldots, n$ вероятность того, что вставка i-го ключа требует выполнения более $2 \lg n$ исследований, не превышает $1/n^2$.

Пусть X_i — случайная величина, обозначающая количество исследований, необходимое при вставке i-го ключа. В подзадаче *б*) вы должны были показать, что $\Pr\{X_i > 2 \lg n\} \leqslant 1/n^2$. Обозначим через X максимальное количество исследований, необходимое при любой из n вставок: $X = \max_{1 \leqslant i \leqslant n} X_i$.

в) Покажите, что $\Pr\{X > 2 \lg n\} \leqslant 1/n$.

г) Покажите, что математическое ожидание $\mathrm{E}[X]$ длины наибольшей последовательности исследований равно $O(\lg n)$.

11-2. **Длины цепочек при хешировании**

Имеется хеш-таблица с n ячейками и разрешением коллизий методом цепочек. Пусть в таблицу вставлено n ключей. Вероятность каждого ключа попасть в любую из ячеек одинакова для всех ключей и всех ячеек. Пусть M — максимальное количество ключей в ячейке после того, как все ключи вставлены в таблицу. Ваша задача — доказать, что математическое ожидание $\mathrm{E}[M]$ имеет верхнюю границу $O(\lg n / \lg \lg n)$.

а) Покажите, что вероятность Q_k того, что в определенной ячейке находится ровно k ключей, равна

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

б) Пусть P_k — вероятность того, что $M = k$, т.е. вероятность того, что ячейка, содержащая наибольшее количество ключей, содержит их ровно k. Покажите, что $P_k \leqslant nQ_k$.

в) Воспользуйтесь приближением Стирлинга (3.17), чтобы показать, что $Q_k < e^k/k^k$.

г) Покажите, что существует константа $c > 1$, такая что $Q_{k_0} < 1/n^3$ для $k_0 = c \lg n / \lg \lg n$. Выведите отсюда, что $P_k < 1/n^2$ для $k \geqslant k_0 = c \lg n / \lg \lg n$.

д) Покажите, что

$$\mathrm{E}[M] \leqslant \Pr\left\{M > \frac{c \lg n}{\lg \lg n}\right\} \cdot n + \Pr\left\{M \leqslant \frac{c \lg n}{\lg \lg n}\right\} \cdot \frac{c \lg n}{\lg \lg n}.$$

Выведите отсюда, что $\mathrm{E}[M] = O(\lg n / \lg \lg n)$.

11-3. **Квадратичное исследование**

Предположим, что у нас есть ключ k, который надо найти в хеш-таблице, ячейки которой пронумерованы от 0 до $m - 1$, и пусть у нас есть хеш-функция h, отображающая пространство ключей в множество $\{0, 1, \ldots, m - 1\}$. Схема поиска выглядит следующим образом.

1. Вычисляем значение $i \leftarrow h(k)$ и присваиваем $j \leftarrow 0$.
2. Исследуем ячейку i на соответствие искомому ключу k. Если ключ найден, или если ячейка пуста, поиск прекращается.
3. Присваиваем $j \leftarrow (j + 1) \bmod m$ и $i \leftarrow (i + j) \bmod m$, и переходим к шагу 2.

Предположим, что m является степенью 2.

а) Покажите, что описанная схема представляет собой частный случай общей схемы "квадратичного исследования" с соответствующими константами c_1 и c_2 в уравнении (11.5).

б) Докажите, что в худшем случае данный алгоритм просматривает все ячейки таблицы.

11-4. **k-универсальное хеширование**

Пусть $\mathcal{H} = \{h\}$ — семейство хеш-функций, в котором каждая хеш-функция h отображает пространство ключей U на множество $\{0, 1, \ldots, m - 1\}$. Мы говорим, что $\mathcal{H}$ является ***k-универсальным***, если для каждой последовательности из k различных ключей $\langle x^{(1)}, x^{(2)}, \ldots, x^{(k)} \rangle$ и случайным образом выбранной из $\mathcal{H}$ хеш-функции h последовательность $\langle h(x^{(1)}) h(x^{(2)}), \ldots, h(x^{(k)}) \rangle$ принимает все m^k последовательностей длины k из элементов множества $\{0, 1, \ldots, m - 1\}$ с равной вероятностью.

а) Покажите, что если множество $\mathcal{H}$ является 2-универсальным, то оно универсально.

б) Пусть U — множество наборов из n значений, выбранных из $\mathbf{Z}_p$, и пусть $B = \mathbf{Z}_p$, где p — простое число. Определим хеш-функцию $h_{a,b} : U \to B$ с $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ из U в качестве аргумента для любого набора $a = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ значений из $\mathbf{Z}_p$ и для любого $b \in \mathbf{Z}_p$ как

$$h_{a,b}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p,$$

и пусть $\mathcal{H} = \{h_{a,b}\}$. Покажите, что множество $\mathcal{H}$ является 2-универсальным.

в) Два друга договорились о "секретной" хеш-функции $h_{a,b}$ из 2-универсального множества хеш-функций $\mathcal{H}$. Один из них шлет сообщение $m \in U$ другому через Internet. Он подписывает это сообщение, пересылая одновременно с ним дескриптор $t = h_{a,b}(m)$, а его друг проверяет, удовлетворяет ли полученная им пара m, t условию $t = h_{a,b}(m)$. Злоумышленник перехватывает сообщение (m, t) и пытается обмануть получателя, подсунув ему собственное сообщение (m', t'). Покажите, что вероятность того, что злоумышленник сможет успешно совершить подмену, не превышает $1/p$, независимо от того, какими вычислительными мощностями он обладает.

Заключительные замечания

Алгоритмы хеширования прекрасно изложены в книгах Кнута (Knuth) [185] и Гоннета (Gonnet) [126]. Согласно Кнуту, хеш-таблицы и метод цепочек были изобретены Луном (H. P. Luhn) в 1953 году. Примерно тогда же Амдал (G. M. Amdahl) предложил идею открытой адресации.

Универсальные множества хеш-функций были предложены Картером (Carter) и Вегманом (Wegman) в 1979 году [52].

Схему идеального хеширования для статических множеств, представленную в разделе 11.5, разработали Фредман (Fredman), Комлёс (Komlós) и Семереди (Szemerédi) [96]. Расширение этого метода для динамических множеств предложено Дицфельбингером (Dietzfelbinger) и др. [73].

ГЛАВА 12

Бинарные деревья поиска

Деревья поиска представляют собой структуры данных, которые поддерживают многие операции с динамическими множествами, включая поиск элемента, минимального и максимального значения, предшествующего и последующего элемента, вставку и удаление. Таким образом, дерево поиска может использоваться и как словарь, и как очередь с приоритетами.

Основные операции в бинарном дереве поиска выполняются за время, пропорциональное его высоте. Для полного бинарного дерева с n узлами эти операции выполняются за время $\Theta(\lg n)$ в наихудшем случае. Как будет показано в разделе 12.4, математическое ожидание высоты построенного случайным образом бинарного дерева равно $O(\lg n)$, так что все основные операции над динамическим множеством в таком дереве выполняются в среднем за время $\Theta(\lg n)$.

На практике мы не всегда можем гарантировать случайность построения бинарного дерева поиска, однако имеются версии деревьев, в которых гарантируется хорошее время работы в наихудшем случае. В главе 13 будет представлена одна из таких версий, а именно — красно-черные деревья, высота которых $O(\lg n)$. В главе 18 вы познакомитесь с B-деревьями, которые особенно хорошо подходят для баз данных, хранящихся во вторичной памяти с произвольным доступом (на дисках).

После знакомства с основными свойствами деревьев поиска, в последующих разделах главы будет показано, как осуществляется обход дерева поиска для вывода его элементов в отсортированном порядке, как выполняется поиск минимального и максимального элементов, а также предшествующего данному элементу и следующего за ним, как вставлять элементы в дерево поиска и удалять их оттуда. Основные математические свойства деревьев описаны в приложении Б.

12.1 Что такое бинарное дерево поиска

Как следует из названия, бинарное дерево поиска в первую очередь является бинарным деревом, как показано на рис. 12.1. Такое дерево может быть представлено при помощи связанной структуры данных, в которой каждый узел является объектом. В дополнение к полям ключа key и сопутствующих данных, каждый узел содержит поля $left$, $right$ и p, которые указывают на левый и правый дочерние узлы и на родительский узел соответственно. Если дочерний или родительский узел отсутствуют, соответствующее поле содержит значение NIL. Единственный узел, указатель p которого равен NIL, — это корневой узел дерева. Ключи в бинарном дереве поиска хранятся таким образом, чтобы в любой момент удовлетворять следующему ***свойству бинарного дерева поиска***.

> Если x — узел бинарного дерева поиска, а узел y находится в левом поддереве x, то $key\,[y] \leqslant key\,[x]$. Если узел y находится в правом поддереве x, то $key\,[x] \leqslant key\,[y]$.

Так, на рис. 12.1*а* ключ корня равен 5, ключи 2, 3 и 5, которые не превышают значение ключа в корне, находятся в его левом поддереве, а ключи 7 и 8, которые не меньше, чем ключ 5, — в его правом поддереве. То же свойство, как легко убедиться, выполняется для каждого другого узла дерева. На рис. 12.1*б* показано дерево с теми же узлами и обладающее тем же свойством, однако менее эффективное в работе, поскольку его высота равна 4, в отличие от дерева на рис. 12.1*а*, высота которого равна 2.

Свойство бинарного дерева поиска позволяет нам вывести все ключи, находящиеся в дереве, в отсортированном порядке с помощью простого рекурсивного алгоритма, называемого ***центрированным (симметричным) обходом дерева*** (inorder tree walk). Этот алгоритм получил данное название в связи с тем, что ключ в корне поддерева выводится между значениями ключей левого поддерева

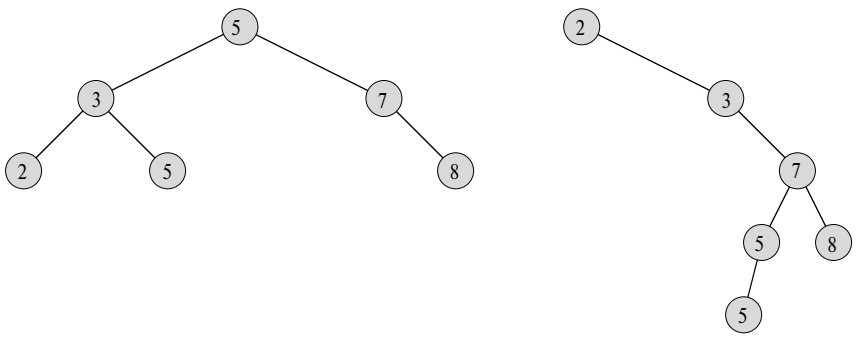


Рис. 12.1. Бинарные деревья поиска

и правого поддерева. Имеются и другие способы обхода, а именно — ***обход в прямом порядке*** (preorder tree walk), при котором сначала выводится корень, а потом — значения левого и правого поддерева, и ***обход в обратном порядке*** (postorder tree walk), когда первыми выводятся значения левого и правого поддерева, а уже затем — корня. Центрированный обход дерева T реализуется процедурой INORDER_TREE_WALK($root\,[T]$):

```
INORDER_TREE_WALK(x)
1  if x ≠ NIL
2     then INORDER_TREE_WALK(left[x])
3          print key[x]
4          INORDER_TREE_WALK(right[x])
```

В качестве примера рассмотрите центрированный обход деревьев, показанных на рис. 12.1, — вы получите в обоих случаях один и тот же порядок ключей, а именно 2, 3, 5, 5, 7, 8. Корректность описанного алгоритма следует непосредственно из свойства бинарного дерева поиска.

Для обхода дерева требуется время $\Theta(n)$, поскольку после начального вызова процедура вызывается ровно два раза для каждого узла дерева: один раз для его левого дочернего узла, и один раз — для правого. Приведенная далее теорема дает нам более формальное доказательство линейности времени центрированного обхода дерева.

Теорема 12.1. Если x — корень поддерева, в котором имеется n узлов, то процедура INORDER_TREE_WALK(x) выполняется за время $\Theta(n)$.

Доказательство. Обозначим через $T(n)$ время, необходимое процедуре INORDER_TREE_WALK в случае вызова с параметром, представляющим собой корень дерева с n узлами. При получении в качестве параметра пустого поддерева, процедуре требуется небольшое постоянное время для выполнения проверки $x \neq$ NIL, так что $T(0) = c$, где c — некоторая положительная константа.

В случае $n > 0$ будем считать, что процедура INORDER_TREE_WALK вызывается один раз для поддерева с k узлами, а второй — для поддерева с $n - k - 1$ узлами. Таким образом, время работы процедуры составляет $T(n) = T(k) + T(n-k-1) + d$, где d — некоторая положительная константа, в которой отражается время, необходимое для выполнения процедуры без учета рекурсивных вызовов.

Воспользуемся методом подстановки, чтобы показать, что $T(n) = \Theta(n)$, путем доказательства того, что $T(n) = (c+d)\,n + c$. При $n = 0$ получаем

$T(0) = (c+d) \cdot 0 + c = c$. Если $n > 0$, то

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + d = \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d = \\ &= (c+d)n + c - (c+d) + c + d = (c+d)n + c, \end{aligned}$$

что и завершает доказательство. ■

Упражнения

12.1-1. Начертите бинарные деревья поиска высотой 2, 3, 4, 5 и 6 для множества ключей $\{1, 4, 5, 10, 16, 17, 21\}$.

12.1-2. В чем заключается отличие свойства бинарного дерева поиска от свойства неубывающей пирамиды (раздел 6.1)? Можно ли использовать свойство неубывающей пирамиды для вывода ключей дерева с n узлами в отсортированном порядке за время $O(n)$? Поясните свой ответ.

12.1-3. Разработайте нерекурсивный алгоритм, осуществляющий обход дерева в симметричном порядке. (*Указание*: имеется простое решение, которое использует вспомогательный стек, и более сложное (и более элегантное) решение, которое обходится без стека, но предполагает возможность проверки равенства двух указателей).

12.1-4. Разработайте рекурсивный алгоритм, который осуществляет прямой и обратный обход дерева с n узлами за время $\Theta(n)$.

12.1-5. Покажите, что, поскольку сортировка n элементов требует в модели сортировки сравнением в худшем случае $\Omega(n \lg n)$ времени, любой алгоритм построения бинарного дерева поиска из произвольного списка, содержащего n элементов, также требует в худшем случае $\Omega(n \lg n)$ времени.

12.2 Работа с бинарным деревом поиска

Наиболее распространенной операцией, выполняемой с бинарным деревом поиска, является поиск в нем определенного ключа. Кроме того, бинарные деревья поиска поддерживают такие запросы, как поиск минимального и максимального элемента, а также предшествующего и последующего. В данном разделе мы рассмотрим все эти операции и покажем, что все они могут быть выполнены в бинарном дереве поиска высотой h за время $O(h)$.

Поиск

Для поиска узла с заданным ключом в бинарном дереве поиска используется следующая процедура TREE_SEARCH, которая получает в качестве параметров указатель на корень бинарного дерева и ключ k, а возвращает указатель на узел с этим ключом (если таковой существует; в противном случае возвращается значение NIL).

```
TREE_SEARCH(x, k)
1  if x = NIL или k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE_SEARCH(left[x], k)
5     else  return TREE_SEARCH(right[x], k)
```

Процедура поиска начинается с корня дерева и проходит вниз по дереву. Для каждого узла x на пути вниз его ключ $key\,[x]$ сравнивается с переданным в качестве параметра ключом k. Если ключи одинаковы, поиск завершается. Если k меньше $key\,[x]$, поиск продолжается в левом поддереве x; если больше — то поиск переходит в правое поддерево. Так, на рис. 12.2 для поиска ключа 13 мы должны пройти следующий путь от корня: $15 \to 6 \to 7 \to 13$. Узлы, которые мы посещаем при рекурсивном поиске, образуют нисходящий путь от корня дерева, так что время работы процедуры TREE_SEARCH равно $O\,(h)$, где h — высота дерева.

Ту же процедуру можно записать итеративно, "разворачивая" оконечную рекурсию в цикл **while**. На большинстве компьютеров такая версия оказывается более эффективной.

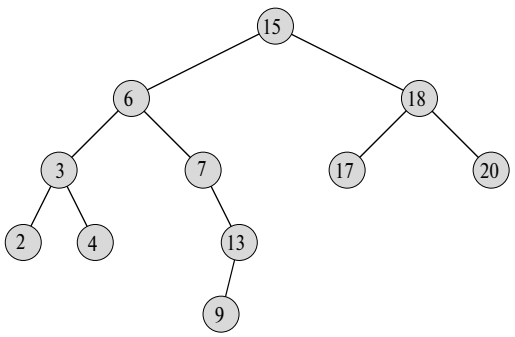


Рис. 12.2. Запросы в бинарном дереве поиска (пояснения в тексте)

```
ITERATIVE_TREE_SEARCH(x, k)
1  while x ≠ NIL и k ≠ key[x]
2        do if k < key[x]
3              then x ← left[x]
4              else  x ← right[x]
5  return x
```

Поиск минимума и максимума

Элемент с минимальным значением ключа легко найти, следуя по указателям $left$ от корневого узла до тех пор, пока не встретится значение NIL. Так, на рис. 12.2, следуя по указателям $left$, мы пройдем путь $15 \to 6 \to 3 \to 2$ до минимального ключа в дереве, равного 2. Вот как выглядит реализация описанного алгоритма:

```
TREE_MINIMUM(x)
1  while left[x] ≠ NIL
2        do x ← left[x]
3  return x
```

Свойство бинарного дерева поиска гарантирует корректность процедуры TREE_MINIMUM. Если у узла x нет левого поддерева, то поскольку все ключи в правом поддереве x не меньше ключа $key\,[x]$, минимальный ключ поддерева с корнем в узле x находится в этом узле. Если же у узла есть левое поддерево, то, поскольку в правом поддереве не может быть узла с ключом, меньшим $key\,[x]$, а все ключи в узлах левого поддерева не превышают $key\,[x]$, узел с минимальным значением ключа находится в поддереве, корнем которого является узел $left\,[x]$.

Алгоритм поиска максимального элемента дерева симметричен алгоритму поиска минимального элемента:

```
TREE_MAXIMUM(x)
1  while right[x] ≠ NIL
2        do x ← right[x]
3  return x
```

Обе представленные процедуры находят минимальный (максимальный) элемент дерева за время $O\,(h)$, где h — высота дерева, поскольку, как и в процедуре TREE_SEARCH, последовательность проверяемых узлов образует нисходящий путь от корня дерева.

Предшествующий и последующий элементы

Иногда, имея узел в бинарном дереве поиска, требуется определить, какой узел следует за ним в отсортированной последовательности, определяемой порядком

центрированного обхода бинарного дерева, и какой узел предшествует данному. Если все ключи различны, последующим по отношению к узлу x является узел с наименьшим ключом, большим $key\,[x]$. Структура бинарного дерева поиска позволяет нам найти этот узел даже не выполняя сравнение ключей. Приведенная далее процедура возвращает узел, следующий за узлом x в бинарном дереве поиска (если таковой существует) и NIL, если x обладает наибольшим ключом в бинарном дереве.

```
TREE_SUCCESSOR(x)
1  if right[x] ≠ NIL
2     then return TREE_MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL и x = right[y]
5        do x ← y
6           y ← p[y]
7  return y
```

Код процедуры TREE_SUCCESSOR разбивается на две части. Если правое поддерево узла x непустое, то следующий за x элемент является крайним левым узлом в правом поддереве, который обнаруживается в строке 2 вызовом процедуры TREE_MINIMUM$(right\,[x])$. Например, на рис. 12.2 следующим за узлом с ключом 15 является узел с ключом 17.

С другой стороны, как требуется показать в упражнении 12.2-6, если правое поддерево узла x пустое, и у x имеется следующий за ним элемент y, то y является наименьшим предком x, чей левый наследник также является предком x. На рис. 12.2 следующим за узлом с ключом 13 является узел с ключом 15. Для того чтобы найти y, мы просто поднимаемся вверх по дереву до тех пор, пока не встретим узел, который является левым дочерним узлом своего родителя. Это действие выполняется в строках 3–7 алгоритма.

Время работы алгоритма TREE_SUCCESSOR в дереве высотой h составляет $O\,(h)$, поскольку мы либо движемся по пути вниз от исходного узла, либо по пути вверх. Процедура поиска последующего узла в дереве TREE_PREDECESSOR симметрична процедуре TREE_SUCCESSOR и также имеет время работы $O\,(h)$.

Если в дереве имеются узлы с одинаковыми ключами, мы можем просто определить последующий и предшествующий узлы как те, что возвращаются процедурами TREE_SUCCESSOR и TREE_PREDECESSOR соответственно.

Таким образом, в этом разделе мы доказали следующую теорему.

Теорема 12.2. Операции поиска, определения минимального и максимального элемента, а также предшествующего и последующего, в бинарном дереве поиска высоты h могут быть выполнены за время $O\,(h)$. ■

Упражнения

12.2-1. Пусть у нас имеется ряд чисел от 1 до 1000, организованных в виде бинарного дерева поиска, и мы выполняем поиск числа 363. Какая из следующих последовательностей *не* может быть последовательностью проверяемых узлов?

а) 2, 252, 401, 398, 330, 344, 397, 363.

б) 924, 220, 911, 244, 898, 258, 362, 363.

в) 925, 202, 911, 240, 912, 245, 363.

г) 2, 399, 387, 219, 266, 382, 381, 278, 363.

д) 935, 278, 347, 621, 299, 392, 358, 363.

12.2-2. Разработайте рекурсивные версии процедур TREE_MINIMUM и TREE_MAXIMUM.

12.2-3. Разработайте процедуру TREE_PREDECESSOR.

12.2-4. Разбираясь с бинарными деревьями поиска, студент решил, что обнаружил их новое замечательное свойство. Предположим, что поиск ключа k в бинарном дереве поиска завершается в листе. Рассмотрим три множества: множество ключей слева от пути поиска A, множество ключей на пути поиска B и множество ключей справа от пути поиска C. Студент считает, что любые три ключа $a \in A$, $b \in B$ и $c \in C$ должны удовлетворять неравенству $a \leqslant b \leqslant c$. Приведите наименьший возможный контрпример, опровергающий предположение студента.

12.2-5. Покажите, что если узел в бинарном дереве поиска имеет два дочерних узла, то последующий за ним узел не имеет левого дочернего узла, а предшествующий ему — правого.

12.2-6. Рассмотрим бинарное дерево поиска T, все ключи которого различны. Покажите, что если правое поддерево узла x в бинарном дереве поиска T пустое и у x есть следующий за ним элемент y, то y является самым нижним предком x, чей левый дочерний узел также является предком x. (Вспомните, что каждый узел является своим собственным предком.)

12.2-7. Центрированный обход бинарного дерева поиска с n узлами можно осуществить путем поиска минимального элемента дерева при помощи процедуры TREE_MINIMUM с последующим $n - 1$ вызовом процедуры TREE_SUCCESSOR. Докажите, что время работы такого алгоритма равно $\Theta(n)$.

12.2-8. Докажите, что какой бы узел ни был взят в качестве исходного в бинарном дереве поиска высотой h, на k последовательных вызовов процедуры TREE_SUCCESSOR потребуется время $O(k + h)$.

12.2-9. Пусть T — бинарное дерево поиска с различными ключами, x — лист этого дерева, а y — его родительский узел. Покажите, что $key\,[y]$ либо является наименьшим ключом в дереве T, превышающим ключ $key\,[x]$, либо наибольшим ключом в T, меньшим ключа $key\,[x]$.

12.3 Вставка и удаление

Операции вставки и удаления приводят к внесению изменений в динамическое множество, представленное бинарным деревом поиска. Структура данных должна быть изменена таким образом, чтобы отражать эти изменения, но при этом сохранить свойство бинарных деревьев поиска. Как мы увидим в этом разделе, вставка нового элемента в бинарное дерево поиска выполняется относительно просто, однако с удалением придется повозиться.

Вставка

Для вставки нового значения v в бинарное дерево поиска T мы воспользуемся процедурой TREE_INSERT. Процедура получает в качестве параметра узел z, у которого $key\,[z] = v$, $left\,[z] =$ NIL и $right\,[z] =$ NIL, после чего она таким образом изменяет T и некоторые поля z, что z оказывается вставленным в соответствующую позицию в дереве.

```
TREE_INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else  x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z                    ▷ Дерево T — пустое
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
```

На рис. 12.3 показана работа процедуры TREE_INSERT. Подобно процедурам TREE_SEARCH и ITERATIVE_TREE_SEARCH, процедура TREE_INSERT начинает работу с корневого узла дерева и проходит по нисходящему пути. Указатель x отмечает проходимый путь, а указатель y указывает на родительский по отношению к x

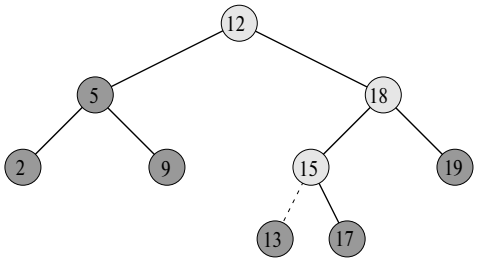


Рис. 12.3. Вставка элемента с ключом 13 в бинарное дерево поиска. Светлые узлы указывают путь от корня к позиции вставки; пунктиром указана связь, добавляемая при вставке нового элемента

узел. После инициализации цикл **while** в строках 3–7 перемещает эти указатели вниз по дереву, перемещаясь влево или вправо в зависимости от результата сравнения ключей $key\,[x]$ и $key\,[z]$, до тех пор пока x не станет равным NIL. Это значение находится именно в той позиции, куда следует поместить элемент z. В строках 8–13 выполняется установка значений указателей для вставки z.

Так же, как и другие примитивные операции над бинарным деревом поиска, процедура TREE_INSERT выполняется за время $O\,(h)$ в дереве высотой h.

Удаление

Процедура удаления данного узла z из бинарного дерева поиска получает в качестве аргумента указатель на z. Процедура рассматривает три возможные ситуации, показанные на рис. 12.4. Если у узла z нет дочерних узлов (рис. 12.4*а*), то мы просто изменяем его родительский узел $p\,[z]$, заменяя в нем указатель на z значением NIL. Если у узла z только один дочерний узел (рис. 12.4*б*), то мы удаляем узел z, создавая новую связь между родительским и дочерним узлом узла z. И наконец, если у узла z два дочерних узла (рис. 12.4*в*), то мы находим следующий за ним узел y, у которого нет левого дочернего узла (см. упражнение 12.2-5), убираем его из позиции, где он находился ранее, путем создания новой связи между его родителем и потомком, и заменяем им узел z.

Код процедуры TREE_DELETE реализует эти действия немного не так, как они описаны.

```
TREE_DELETE(T, z)
 1  if left[z] = NIL или right[z] = NIL
 2     then y ← z
 3     else y ← TREE_SUCCESSOR(z)
 4  if left[y] ≠ NIL
```

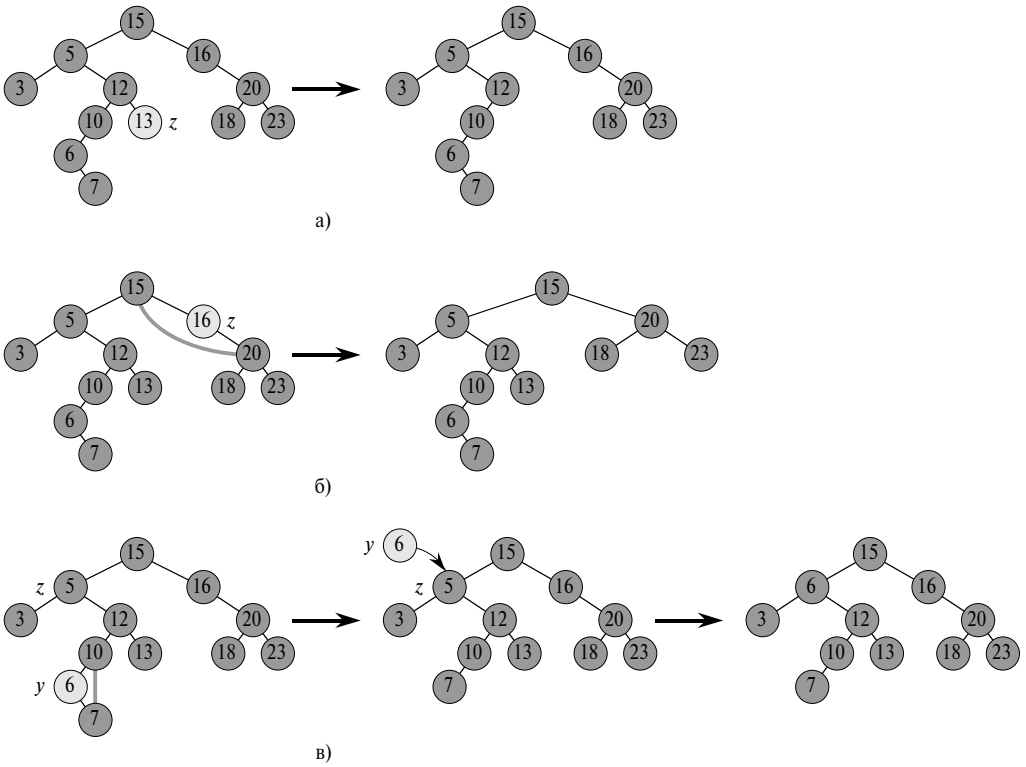


Рис. 12.4. Удаление узла z из бинарного дерева поиска

```
 5     then x ← left[y]
 6     else x ← right[y]
 7  if x ≠ NIL
 8     then p[x] ← p[y]
 9  if p[y] = NIL
10     then root[T] ← x
11     else if y = left[p[y]]
12             then left[p[y]] ← x
13             else right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
16             Копирование сопутствующих данных в z
17  return y
```

В строках 1–3 алгоритм определяет убираемый путем “склейки” родителя и потомка узел y. Этот узел представляет собой либо узел z (если у узла z не более одного дочернего узла), либо узел, следующий за узлом z (если у z два дочерних

узла). Затем в строках 4–6 x присваивается указатель на дочерний узел узла y либо значение NIL, если у y нет дочерних узлов. Затем узел y убирается из дерева в строках 7–13 путем изменения указателей в $p[y]$ и x. Это удаление усложняется необходимостью корректной отработки граничных условий (когда x равно NIL или когда y — корневой узел). И наконец, в строках 14–16, если удаленный узел y был следующим за z, мы перезаписываем ключ z и сопутствующие данные ключом и сопутствующими данными y. Удаленный узел y возвращается в строке 17, с тем чтобы вызывающая процедура могла при необходимости освободить или использовать занимаемую им память. Время работы описанной процедуры с деревом высотой h составляет $O(h)$.

Таким образом, в этом разделе мы доказали следующую теорему.

Теорема 12.3. Операции вставки и удаления в бинарном дереве поиска высоты h могут быть выполнены за время $O(h)$. ■

Упражнения

12.3-1. Приведите рекурсивную версию процедуры TREE_INSERT.

12.3-2. Предположим, что бинарное дерево поиска строится путем многократной вставки в дерево различных значений. Покажите, что количество узлов, проверяемых при поиске некоторого значения в дереве, на один больше, чем количество узлов, проверявшихся при вставке этого значения в дерево.

12.3-3. Отсортировать множество из n чисел можно следующим образом: сначала построить бинарное дерево поиска, содержащее эти числа (вызывая процедуру TREE_INSERT для вставки чисел в дерево одно за другим), а затем выполнить центрированный обход получившегося дерева. Чему равно время работы такого алгоритма в наилучшем и наихудшем случае?

12.3-4. Предположим, что указатель на узел y бинарного дерева поиска хранится в некоторой внешней структуре данных и что предшествующий ему узел z удаляется из дерева с помощью процедуры TREE_DELETE. Какая проблема при этом может возникнуть? Каким образом следует переписать процедуру TREE_DELETE, чтобы ее избежать?

12.3-5. Является ли операция удаления “коммутативной” в том смысле, что удаление x с последующим удалением y из бинарного дерева поиска приводит к тому же результирующему дереву, что и удаление y с последующим удалением x? Обоснуйте свой ответ.

12.3-6. Если узел z в процедуре TREE_DELETE имеет два дочерних узла, то можно воспользоваться не последующим за ним узлом, а предшествующим. Утверждается, что стратегия, которая состоит в равновероятном выборе

предшествующего или последующего узла, приводит к большей сбалансированности дерева. Каким образом следует изменить процедуру TREE_DELETE для реализации указанной стратегии?

⋆ 12.4 Случайное построение бинарных деревьев поиска

Мы показали, что все базовые операции с бинарными деревьями поиска имеют время выполнения $O(h)$, где h — высота дерева. Однако при вставке и удалении элементов высота дерева меняется. Если, например, все элементы вставляются в дерево в строго возрастающей последовательности, то такое дерево вырождается в цепочку высоты $n-1$. С другой стороны, как показано в упражнении Б.5-4, $h \geqslant \lfloor \lg n \rfloor$. Как и в случае быстрой сортировки, можно показать, что поведение алгоритма в среднем гораздо ближе к наилучшему случаю, чем к наихудшему.

К сожалению, в ситуации, когда при формировании бинарного дерева поиска используются и вставки, и удаления, о средней высоте образующихся деревьев известно мало, так что мы ограничимся анализом ситуации, когда дерево строится только с использованием вставок, без удалений. Определим ***случайное бинарное дерево поиска*** (randomly built binary search tree) с n ключами как дерево, которое возникает при вставке ключей в изначально пустое дерево в случайном порядке, когда все $n!$ перестановок входных ключей равновероятны (в упражнении 12.4-3 требуется показать, что это условие отличается от условия равновероятности всех возможных бинарных деревьев поиска с n узлами). В данном разделе мы покажем, что математическое ожидание высоты случайного бинарного дерева поиска с n ключами равно $O(\lg n)$. Мы считаем, что все ключи различны.

Начнем с определения трех случайных величин, которые помогут определить высоту случайного бинарного дерева поиска. Обозначая высоту случайного бинарного дерева поиска с n ключами через X_n, определим ***экспоненциальную высоту*** $Y_n = 2^{X_n}$. При построении бинарного дерева поиска с n ключами мы выбираем один из них в качестве корня. Обозначим через R_n случайную величину, равную ***рангу*** корневого ключа в множестве из всех n ключей, т.е. R_n содержит позицию, которую бы занимал ключ, если бы множество было отсортировано. Значение R_n с равной вероятностью может быть любым элементом множества $\{1, 2, \ldots, n\}$. Если $R_n = i$, то левое поддерево корня представляет собой случайное бинарное дерево поиска с $i-1$ ключами, а правое — с $n-i$ ключами. Поскольку высота бинарного дерева на единицу больше наибольшей из высот поддеревьев корневого узла, экспоненциальная высота бинарного дерева в два раза больше наибольшей экспоненциальной высоты поддеревьев корневого узла. Если $R_n = i$, то

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}).$$

Очевидно, что $Y_1 = 1$, поскольку экспоненциальная высота дерева с одним узлом равна $2^0 = 1$; кроме того, для удобства определим $Y_0 = 0$.

Далее мы определим индикаторные случайные величины $Z_{n,1}, Z_{n,2}, \ldots, Z_{n,n}$, где

$$Z_{n,i} = \mathrm{I}\{R_n = i\}.$$

Поскольку R_n с равной вероятностью может быть любым элементом множества $\{1, 2, \ldots, n\}$, $\Pr\{R_n = 1\} = 1/n$ для всех i от 1 до n, а значит, согласно лемме 5.1,

$$\mathrm{E}[Z_{n,i}] = 1/n \tag{12.1}$$

для всех $i = 1, 2, \ldots, n$. Кроме того, поскольку только одно значение $Z_{n,i}$ равно 1, а все остальные равны 0, имеем

$$Y_n = \sum_{i=1}^{n} Z_{n,i} \left(2 \cdot \max\left(Y_{i-1}, Y_{n-i}\right)\right).$$

Мы покажем, что $\mathrm{E}[Y_n]$ полиномиально зависит от n, что неизбежно приводит к выводу, что $\mathrm{E}[X_n] = O(\lg n)$.

Индикаторная случайная переменная $Z_{n,i} = \mathrm{I}\{R_n = i\}$ не зависит от значений Y_{i-1} и Y_{n-i}. Если $R_n = i$, то левое поддерево, экспоненциальная высота которого равна Y_{i-1}, случайным образом строится из $i - 1$ ключей, ранги которых меньше i. Это поддерево ничем не отличается от любого другого случайного бинарного дерева поиска из $i - 1$ ключей. Выбор $R_n = i$ никак не влияет на структуру этого дерева, а только на количество содержащихся в нем узлов. Следовательно, случайные величины Y_{i-1} и $Z_{n,i}$ независимы. Аналогично, правое поддерево, экспоненциальная высота которого равна Y_{n-i}, строится случайным образом из $n - i$ ключей, ранги которых больше i. Структура этого дерева не зависит от R_n, так что случайные величины Y_{n-i} и $Z_{n,i}$ независимы. Следовательно,

$$\begin{aligned}
\mathrm{E}[Y_n] &= \mathrm{E}\left[\sum_{i=1}^{n} Z_{n,i}\left(2 \cdot \max\left(Y_{i-1}, Y_{n-i}\right)\right)\right] = \\
&= \sum_{i=1}^{n} \mathrm{E}\left[Z_{n,i}\left(2 \cdot \max\left(Y_{i-1}, Y_{n-i}\right)\right)\right] = && \text{(в силу линейности математического ожидания)} \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}]\,\mathrm{E}\left[2 \cdot \max\left(Y_{i-1}, Y_{n-i}\right)\right] = && \text{(в силу независимости } Z \text{ и } Y\text{)} \\
&= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathrm{E}\left[2 \cdot \max\left(Y_{i-1}, Y_{n-i}\right)\right] = && \text{(в соответствии с (12.1))}
\end{aligned}$$

$$= \frac{2}{n}\sum_{i=1}^{n}\mathrm{E}\left[\max\left(Y_{i-1}, Y_{n-i}\right)\right] \leqslant \qquad \text{(в соответствии с (B.21))}$$

$$\leqslant \frac{2}{n}\sum_{i=1}^{n}\left(\mathrm{E}\left[Y_{i-1}\right] + \mathrm{E}\left[Y_{n-i}\right]\right) \qquad \text{(см. упражнение B.3-4).}$$

Каждый член $\mathrm{E}\left[Y_0\right], \mathrm{E}\left[Y_1\right], \ldots, \mathrm{E}\left[Y_{n-1}\right]$ в последней сумме встречается дважды: один раз как $\mathrm{E}\left[Y_{i-1}\right]$, и второй — как $E\left[Y_{n-i}\right]$, так что мы получаем следующее рекуррентное соотношение:

$$\mathrm{E}\left[Y_n\right] \leqslant \frac{4}{n}\sum_{i=0}^{n-1}\mathrm{E}\left[Y_i\right]. \tag{12.2}$$

Используя метод подстановок, покажем, что для всех натуральных n рекуррентное соотношение (12.2) имеет следующее решение:

$$\mathrm{E}\left[Y_n\right] \leqslant \frac{1}{4}\binom{n+3}{4}.$$

При этом мы воспользуемся тождеством

$$\sum_{i=0}^{n-1}\binom{i+3}{3} = \binom{n+3}{4}. \tag{12.3}$$

(В упражнении 12.4-1 предлагается самостоятельно доказать данное тождество.)

Убедимся, что для базовых случаев полученные границы справедливы:

$$0 = Y_0 = \mathrm{E}\left[Y_0\right] \leqslant \frac{1}{4}\binom{3}{3} = \frac{1}{4},$$

$$1 = Y_1 = \mathrm{E}\left[Y_1\right] \leqslant \frac{1}{4}\binom{1+3}{3} = 1.$$

Используя подстановку, получаем:

$$\mathrm{E}\left[Y_n\right] \leqslant \frac{4}{n}\sum_{i=0}^{n-i}\mathrm{E}\left[Y_i\right] \leqslant$$

$$\leqslant \frac{4}{n}\sum_{i=0}^{n-1}\frac{1}{4}\binom{i+3}{n} = \qquad \text{(в соответствии с гипотезой индукции)}$$

$$= \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3} =$$

$$= \frac{1}{n}\binom{n+3}{4} = \qquad \text{(в соответствии с (12.3))}$$

$$= \frac{1}{n} \cdot \frac{(n+3)!}{4!\,(n-1)!} =$$
$$= \frac{1}{4} \cdot \frac{(n+3)!}{3!n!} =$$
$$= \frac{1}{4}\binom{n+3}{3}.$$

Итак, мы получили границу для $\mathrm{E}[Y_n]$, но наша конечная цель — найти границу $\mathrm{E}[X_n]$. В упражнении 12.4-4 требуется показать, что функция $f(x) = 2^x$ выпуклая вниз (см. раздел B.3). Таким образом, мы можем применить неравенство Йенсена (B.25), которое гласит, что

$$2^{\mathrm{E}[X_n]} \leqslant \mathrm{E}\left[2^{X_n}\right] = \mathrm{E}[Y_n],$$

и получить, что

$$2^{\mathrm{E}[X_n]} \leqslant \frac{1}{4}\binom{n+3}{3} = \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} = \frac{n^3+6n^2+11n+6}{24}.$$

Логарифмируя обе части неравенства, находим, что $\mathrm{E}[X_n] = O(\lg n)$. Таким образом, нами доказана следующая теорема.

Теорема 12.4. Математическое ожидание высоты случайного бинарного дерева поиска с n ключами равно $O(\lg n)$. ■

Упражнения

12.4-1. Докажите уравнение (12.3).

12.4-2. Приведите пример бинарного дерева поиска с n узлами, такого что средняя глубина узла в дереве равна $\Theta(\lg n)$, в то время как высота дерева — $\omega(\lg n)$. Найдите асимптотическую верхнюю границу высоты бинарного дерева поиска с n узлами, средняя глубина узла в котором составляет $\Theta(\lg n)$.

12.4-3. Покажите, что при нашем определении случайных бинарных деревьев поиска с n узлами они не являются равновероятными. (*Указание*: рассмотрите все возможные деревья при $n = 3$.)

12.4-4. Покажите, что функция $f(x) = 2^x$ является выпуклой вниз.

⋆ 12.4-5. Рассмотрим алгоритм RANDOMIZED_QUICKSORT, работающий с входной последовательностью из n различных чисел. Докажите, что для любой константы $k > 0$ время работы алгоритма превышает $O(n \lg n)$ только для $O\left(1/n^k\right)$-й части всех возможных $n!$ перестановок.

Задачи

12-1. **Бинарные деревья поиска с одинаковыми ключами**

Одинаковые ключи приводят к проблеме при реализации бинарных деревьев поиска.

а) Чему равно асимптотическое время работы процедуры TREE_INSERT при вставке n одинаковых ключей в изначально пустое дерево?

Мы предлагаем улучшить алгоритм TREE_INSERT, добавив в него перед строкой 5 проверку равенства $key[z] = key[x]$, а перед строкой 11 — проверку равенства $key[z] = key[y]$. Если равенства выполняются, мы реализуем одну из описанных далее стратегий. Для каждой из них найдите асимптотическое время работы при вставке n одинаковых ключей в изначально пустое дерево. (Описания приведены для строки 5, в которой сравниваются ключи z и x. Для строки 11 замените в описании x на y.)

б) Храним в узле x флаг $b[x]$ с логическим значением FALSE или TRUE, которое и определяет выбор $left[x]$ или $right[x]$ при вставке ключа со значением, равным ключу x. Значение флага при каждой такой вставке изменяется на противоположное, что позволяет чередовать выбор дочернего узла.

в) Храним все элементы с одинаковыми ключами в одном узле при помощи списка, и при вставке просто добавляем элемент в этот список.

г) Случайным образом выбираем дочерний узел $left[x]$ или $right[x]$. (Каково будет время работы такой стратегии в наихудшем случае? Оцените среднее время работы данной стратегии.)

12-2. **Цифровые деревья**

Пусть имеется две строки $a = a_0a_1 \ldots a_p$ и $b = b_0b_1 \ldots b_q$, в которых все символы принадлежат некоторому упорядоченному множеству. Мы говорим, что строка a ***лексикографически меньше*** строки b, если выполняется одно из двух условий.

1) Существует целое $0 \leqslant j \leqslant \min(p, q)$, такое что $a_i = b_i$ для всех $i = 0, 1, \ldots, j - 1$ и $a_j < b_j$.

2) $p < q$ и для всех $i = 0, 1, \ldots, p$ выполняется равенство $a_i = b_i$.

Например, если a и b — битовые строки, то $10100 < 10110$ в соответствии с правилом 1 (здесь $j = 3$), а $10100 < 101000$ согласно правилу 2. Такое упорядочение подобно упорядочиванию по алфавиту слов в словаре естественного языка.

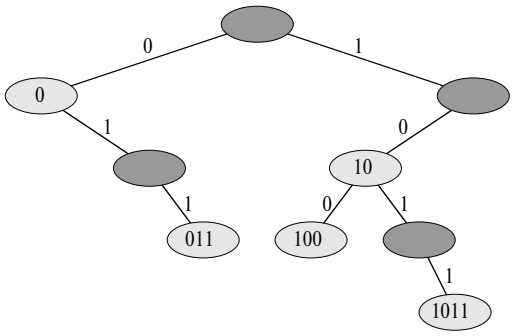


Рис. 12.5. Цифровое дерево. Ключ узла определяется путем прохода от корня к данному узлу и не должен храниться в дереве. Темным цветом показаны промежуточные узлы, не имеющие ключей

Структура ***цифрового дерева*** (radix tree) показана на рис. 12.5. Приведенное на рисунке дерево хранит битовые строки 1011, 10, 011, 100 и 0. При поиске ключа $a = a_0 a_1 \ldots a_p$ на i-м шаге мы переходим к левому узлу, если $a_i = 0$, и к правому, если $a_i = 1$. Пусть S — множество различных бинарных строк с суммарной длиной n. Покажите, как использовать цифровое дерево для лексикографической сортировки за время $\Theta(n)$. Для примера, приведенного на рис. 12.5, отсортированная последовательность должна выглядеть следующим образом: 0, 011, 10, 100, 1011.

12-3. **Средняя глубина вершины в случайном бинарном дереве поиска**

В данной задаче мы докажем, что средняя глубина узла в случайном бинарном дереве поиска с n узлами равна $O(\lg n)$. Хотя этот результат и является более слабым, чем в теореме 12.4, способ доказательства демонстрирует интересные аналогии между построением бинарного дерева поиска и работой алгоритма RANDOMIZED_QUICKSORT из раздела 7.3.

Определим ***общую длину путей*** $P(T)$ бинарного дерева T как сумму глубин всех узлов $x \in T$, которые мы будем обозначать как $d(x, T)$.

а) Покажите, что средняя глубина узла в дереве T равна

$$\frac{1}{n} \sum_{x \in T} d(x, T) = \frac{1}{n} P(T).$$

Таким образом, нам надо показать, что математическое ожидание $P(T)$ равно $O(n \lg n)$.

б) Обозначим через T_L и T_R соответственно левое и правое поддеревья дерева T. Покажите, что если дерево T имеет n узлов, то

$$P(T) = P(T_L) + P(T_R) + n - 1.$$

в) Обозначим через $P(n)$ среднюю общую длину путей случайного бинарного дерева поиска с n узлами. Покажите, что

$$P(n) = \frac{1}{n} \sum_{i=0}^{n-1} (P(i) + P(n-i-1) + n - 1).$$

г) Покажите, что

$$P(n) = \frac{2}{n} \sum_{k=1}^{n-1} P(k) + \Theta(n).$$

д) Вспоминая анализ рандомизированной версии алгоритма быстрой сортировки из задачи 7-2, сделайте вывод, что $P(n) = O(n \lg n)$.

В каждом рекурсивном вызове быстрой сортировки мы выбираем случайный опорный элемент разделения множества сортируемых элементов. Каждый узел в бинарном дереве также отделяет часть элементов, попадающих в поддерево, для которого данный узел является корневым.

е) Приведите реализацию алгоритма быстрой сортировки, в которой в процессе сортировки множества элементов выполняются те же сравнения, что и для вставки элементов в бинарное дерево поиска. (Порядок, в котором выполняются сравнения, может быть иным, однако сами множества сравнений должны совпадать.)

12-4. **Количество различных бинарных деревьев**

Обозначим через b_n количество различных бинарных деревьев с n узлами. В этой задаче будет выведена формула для b_n и найдена ее асимптотическая оценка.

а) Покажите, что $b_0 = 1$ и что для $n \geqslant 1$

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}.$$

б) Пусть $B(x)$ — производящая функция (определение производящей функции можно найти в задаче 4.5):

$$B(x) = \sum_{n=0}^{\infty} b_n x^n.$$

Покажите, что $B(x) = xB(x)^2 + 1$ и, следовательно, $B(x)$ можно записать как

$$B(x) = \frac{1}{2x}\left(1 - \sqrt{1-4x}\right).$$

Разложение в ряд Тейлора функции $f(x)$ вблизи точки $x = a$ определяется как

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k,$$

где $f^{(k)}(x)$ означает k-ю производную функции f в точке x.

в) Покажите, что b_n является n-м ***числом Каталана***

$$b_n = \frac{1}{n+1}\binom{2n}{n},$$

используя разложение в ряд Тейлора функции $\sqrt{1-4x}$ вблизи точки $x = 0$. (Если хотите, то вместо использования разложения в ряд Тейлора можно использовать обобщение биномиального разложения (B.4) для нецелого показателя степени n, когда для любого действительного числа n и целого k выражение $\binom{n}{k}$ интерпретируется как $n(n-1)\ldots(n-k+1)/k!$ при $k \geqslant 0$ и 0 в противном случае.)

г) Покажите, что

$$b_n = \frac{4^n}{\sqrt{\pi}n^{3/2}}\left(1 + O(1/n)\right).$$

Заключительные замечания

Подробное рассмотрение простых бинарных деревьев поиска (которые были независимо открыты рядом исследователей в конце 1950-х годов) и их различных вариаций имеется в работе Кнута (Knuth) [185]. Там же рассматриваются и цифровые деревья.

В разделе 15.5 будет показано, каким образом можно построить оптимальное бинарное дерево поиска, если частоты поисков известны заранее, до начала построения дерева. В этом случае дерево строится таким образом, чтобы при наиболее частых поисках просматривалось минимальное количество узлов дерева.

Граница математического ожидания высоты случайного бинарного дерева поиска в разделе 12.4 найдена Асламом (Aslam) [23]. Мартинец (Martínez) и Роура (Roura) [211] разработали рандомизированный алгоритм для вставки узлов в бинарное дерево поиска и удаления их оттуда, при использовании которого в результате получается случайное бинарное дерево поиска. Однако их определение случайного бинарного дерева поиска несколько отлично от приведенного в данной главе.

ГЛАВА 13

Красно-черные деревья

В главе 12 было показано, что бинарные деревья поиска высоты h реализуют все базовые операции над динамическими множествами (SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT и DELETE) за время $O(h)$. Таким образом, операции выполняются тем быстрее, чем меньше высота дерева. Однако в наихудшем случае производительность бинарного дерева поиска оказывается ничуть не лучше, чем производительность связанного списка. Красно-черные деревья представляют собой одну из множества "сбалансированных" схем деревьев поиска, которые гарантируют время выполнения операций над динамическим множеством $O(\lg n)$ даже в наихудшем случае.

13.1 Свойства красно-черных деревьев

Красно-черное дерево представляет собой бинарное дерево поиска с одним дополнительным битом ***цвета*** в каждом узле. Цвет узла может быть либо красным, либо черным. В соответствии с накладываемыми на узлы дерева ограничениями, ни один путь в красно-черном дереве не отличается от другого по длине более чем в два раза, так что красно-черные деревья являются приближенно ***сбалансированными***.

Каждый узел дерева содержит поля $color$, key, $left$, $right$ и p. Если не существует дочернего или родительского узла по отношению к данному, соответствующий указатель принимает значение NIL. Мы будем рассматривать эти значения NIL как указатели на внешние узлы (листья) бинарного дерева поиска. При этом все "нормальные" узлы, содержащие поле ключа, становятся внутренними узлами дерева.

Бинарное дерево поиска является красно-черным деревом, если оно удовлетворяет следующим ***красно-черным свойствам***.

1. Каждый узел является красным или черным.
2. Корень дерева является черным.
3. Каждый лист дерева (NIL) является черным.
4. Если узел — красный, то оба его дочерних узла — черные.
5. Для каждого узла все пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов.

На рис. 13.1*а* показан пример красно-черного дерева. На рисунке черные узлы показаны темным цветом, красные — светлым. Возле каждого узла показана его "черная" высота. У всех листьев она, само собой разумеется, равна 0.

Для удобства работы с красно-черным деревом мы заменим все листья одним ограничивающим узлом, представляющим значение NIL (с этим приемом мы уже встречались в разделе 10.2). В красно-черном дереве T ограничитель $nil\,[T]$ представляет собой объект с теми же полями, что и обычный узел дерева. Значение $color$ этого узла равно BLACK (черный), а все остальные поля могут иметь произвольные значения. Как показано на рис. 13.1*б*, все указатели на NIL заменяются указателем на ограничитель $nil\,[T]$.

Использование ограничителя позволяет нам рассматривать дочерний по отношению к узлу x NIL как обычный узел, родителем которого является узел x. Хотя можно было бы использовать различные ограничители для каждого значения NIL (что позволило бы точно определять их родительские узлы), этот подход привел бы к неоправданному перерасходу памяти. Вместо этого мы используем единственный ограничитель для представления всех NIL — как листьев, так и родительского узла корня. Величины полей p, $left$, $right$ и key ограничителя не играют никакой роли, хотя для удобства мы можем присвоить им те или иные значения.

В целом мы ограничим наш интерес к красно-черным деревьям только их внутренними узлами, поскольку только они хранят значения ключей. В оставшейся части данной главы при изображении красно-черных деревьев все листья опускаются, как это сделано на рис. 13.1*в*.

Количество черных узлов на пути от узла x (не считая сам узел) к листу будем называть ***черной высотой*** узла (black-height) и обозначать как $bh\,(x)$. В соответствии со свойством 5 красно-черных деревьев, черная высота узла — точно определяемое значение. Черной высотой дерева будем считать черную высоту его корня.

Следующая лемма показывает, почему красно-черные деревья хорошо использовать в качестве деревьев поиска.

Лемма 13.1. Красно-черное дерево с n внутренними узлами имеет высоту не более чем $2\lg(n+1)$.

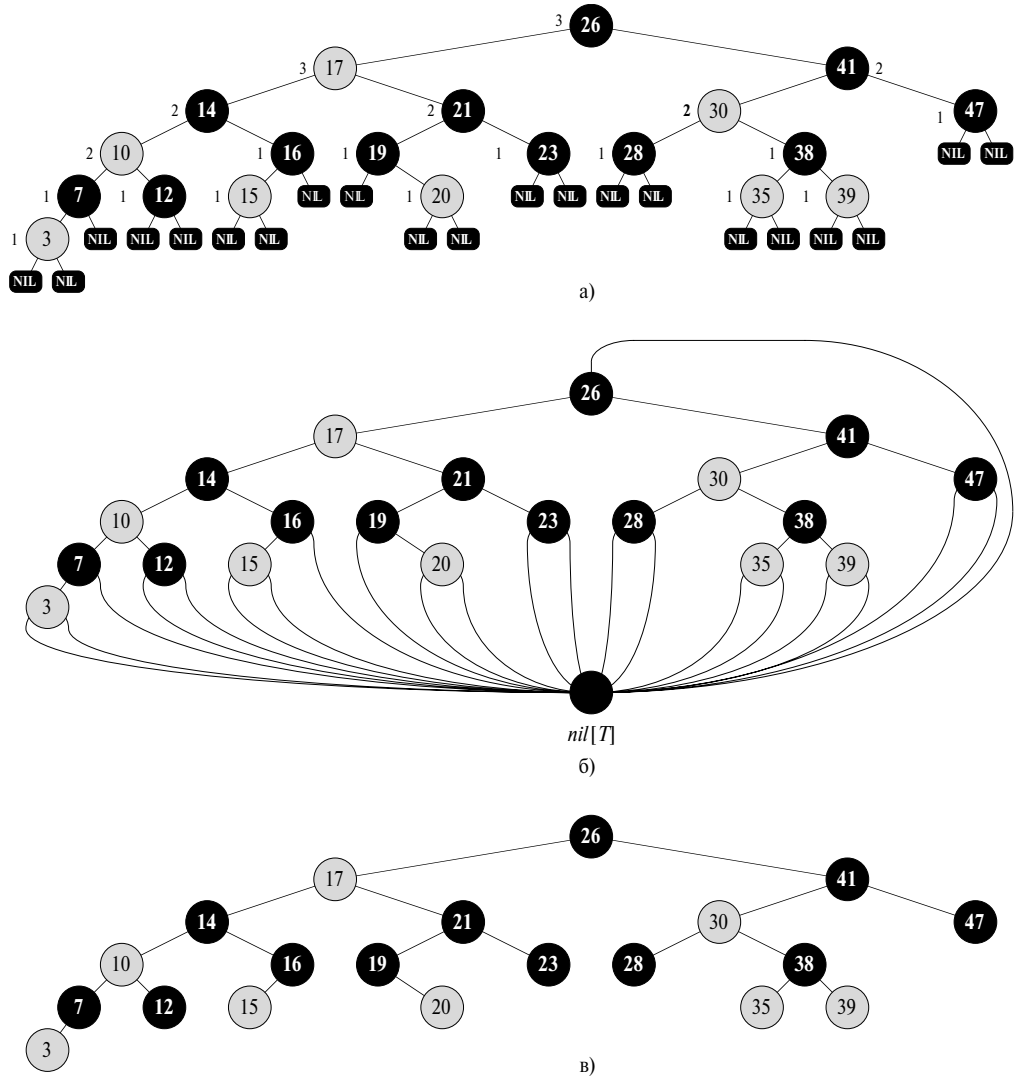


Рис. 13.1. Пример красно-черного дерева

Доказательство. Начнем с того, что покажем, что поддерево любого узла x содержит как минимум $2^{bh(x)} - 1$ внутренних узлов. Докажем это по индукции по высоте x. Если высота x равна 0, то узел x должен быть листом ($nil\,[T]$) и поддерево узла x содержит не менее $2^{bh(x)} - 1 = 2^0 - 1 = 0$ внутренних узлов. Теперь рассмотрим узел x, который имеет положительную высоту и представляет собой внутренний узел с двумя потомками. Каждый потомок имеет черную высоту либо $bh\,(x)$, либо $bh\,(x) - 1$, в зависимости от того, является ли его цвет соответственно красным или черным. Поскольку высота потомка x меньше, чем

высота самого узла x, мы можем использовать предположение индукции и сделать вывод о том, что каждый из потомков x имеет как минимум $2^{bh(x)} - 1$ внутренних узлов. Таким образом, дерево с корнем в вершине x содержит как минимум $\left(2^{bh(x)-1} - 1\right) + \left(2^{bh(x)-1} - 1\right) + 1 = 2^{bh(x)} - 1$ внутренних узлов, что и доказывает наше утверждение.

Для того чтобы завершить доказательство леммы, обозначим высоту дерева через h. Согласно свойству 4, по крайней мере половина узлов на пути от корня к листу, не считая сам корень, должны быть черными. Следовательно, черная высота корня должна составлять как минимум $h/2$; следовательно,

$$n \geqslant 2^{h/2} - 1.$$

Перенося 1 в левую часть и логарифмируя, получим, что $\lg(n+1) \geqslant h/2$, или $h \leqslant 2\lg(n+1)$. ■

Непосредственным следствием леммы является то, что такие операции над динамическими множествами, как SEARCH, MINIMUM, MAXIMUM, PREDECESSOR и SUCCESSOR, при использовании красно-черных деревьев выполняются за время $O(\lg h)$, поскольку, как показано в главе 12, время работы этих операций на дереве поиска высотой h составляет $O(h)$, а любое красно-черное дерево с n узлами является деревом поиска высотой $O(\lg n)$. (Само собой разумеется, ссылки на NIL в алгоритмах в главе 12 должны быть заменены ссылками на $nil[T]$.) Хотя алгоритмы TREE_INSERT и TREE_DELETE из главы 12 и характеризуются временем работы $O(\lg n)$, если использовать их для вставки и удаления из красно-черного дерева, непосредственно использовать их для выполнения операций INSERT и DELETE нельзя, поскольку они не гарантируют сохранение красно-черных свойств после внесения изменений в дерево. Однако в разделах 13.3 и 13.4 вы увидите, что эти операции также могут быть выполнены за время $O(\lg n)$.

Упражнения

13.1-1. Начертите в стиле рис. 13.1*a* полное бинарное дерево поиска высоты 3 с ключами $\{1, 2, \ldots, 15\}$. Добавьте к нему листья NIL и раскрасьте получившееся дерево разными способами, так чтобы в результате получились красно-черные деревья с черной высотой 2, 3 и 4.

13.1-2. Начертите красно-черное дерево, которое получится в результате вставки при помощи алгоритма TREE_INSERT ключа 36 в дерево, изображенное на рис. 13.1. Если вставленный узел закрасить красным, будет ли получившееся в результате дерево красно-черным? А если закрасить этот узел черным?

13.1-3. Определим ***ослабленное красно-черное дерево*** как бинарное дерево поиска, которое удовлетворяет красно-черным свойствам 1, 3, 4 и 5. Другими словами, корень может быть как черным, так и красным. Рассмотрите ослабленное красно-черное дерево T, корень которого красный. Если мы перекрасим корень дерева T из красного цвета в черный, будет ли получившееся в результате дерево красно-черным?

13.1-4. Предположим, что каждый красный узел в красно-черном дереве "поглощается" его черный родительским узлом, так что дочерний узел красного узла становится дочерним узлом его черного родителя (мы не обращаем внимания на то, что при этом происходит с ключами в узлах). Чему равен возможный порядок черного узла после того, как будут поглощены все его красные потомки? Что вы можете сказать о глубине листьев получившегося дерева?

13.1-5. Покажите, что самый длинный нисходящий путь от вершины x к листу красно-черного дерева имеет длину, не более чем в два раза превышающую кратчайший нисходящий путь от x к листу.

13.1-6. Чему равно наибольшее возможное число внутренних узлов в красно-черном дереве с черной высотой k? А наименьшее возможное число?

13.1-7. Опишите красно-черное дерево с n ключами с наибольшим возможным отношением количества красных внутренних узлов к количеству черных внутренних узлов. Чему равно это отношение? Какое дерево имеет наименьшее указанное отношение, и чему равна его величина?

13.2 Повороты

Операции над деревом поиска TREE_INSERT и TREE_DELETE, будучи применены к красно-черному дереву с n ключами, выполняются за время $O(\lg n)$. Поскольку они изменяют дерево, в результате их работы могут нарушаться красно-черные свойства, перечисленные в разделе 13.1. Для восстановления этих свойств мы должны изменить цвета некоторых узлов дерева, а также структуру его указателей.

Изменения в структуре указателей будут выполняться при помощи ***поворотов*** (rotations), которые представляют собой локальные операции в дереве поиска, сохраняющие свойство бинарного дерева поиска. На рис. 13.2 показаны два типа поворотов — левый и правый (здесь α, β и γ — произвольные поддеревья). При выполнении левого поворота в узле x предполагается, что его правый дочерний узел y не является листом $nil\,[T]$. Левый поворот выполняется "вокруг" связи между x и y, делая y новым корнем поддерева, левым дочерним узлом которого становится x, а бывший левый потомок узла y — правым потомком x.

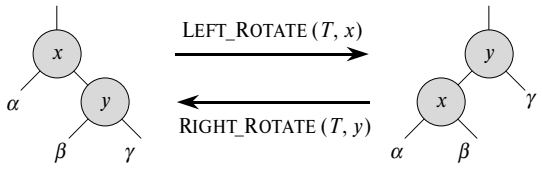


Рис. 13.2. Операции поворота в бинарном дереве поиска

В псевдокоде процедуры LEFT_ROTATE предполагается, что $right\,[x] \neq nil\,[T]$, а родитель корневого узла — $nil\,[T]$.

```
LEFT_ROTATE(T, x)
 1  y ← right[x]              ▷ Устанавливаем y.
 2  right[x] ← left[y]        ▷ Левое поддерево y становится
                              ▷ правым поддеревом x
 3  if left[y] ≠ nil[T]
 4     then p[left[y]] ← x
 5  p[y] ← p[x]               ▷ Перенос родителя x в y
 6  if p[x] = nil[T]
 7     then root[T] ← y
 8     else if x = left[p[x]]
 9             then left[p[x]] ← y
10             else right[p[x]] ← y
11  left[y] ← x               ▷ x — левый дочерний y
12  p[x] ← y
```

На рис. 13.3 показан конкретный пример выполнения процедуры LEFT_ROTATE. Код процедуры RIGHT_ROTATE симметричен коду LEFT_ROTATE. Обе эти процедуры выполняются за время $O\,(1)$. При повороте изменяются только указатели, все остальные поля сохраняют свое значение.

Упражнения

13.2-1. Запишите псевдокод процедуры RIGHT_ROTATE.

13.2-2. Покажите, что в каждом бинарном дереве поиска с n узлами имеется ровно $n - 1$ возможных поворотов.

13.2-3. Пусть a, b и c — произвольные узлы в поддеревьях α, β и γ в левой части рис. 13.2. Как изменится глубина узлов a, b и c при левом повороте в узле x?

13.2-4. Покажите, что произвольное бинарное дерево поиска с n узлами может быть преобразовано в любое другое бинарное дерево поиска с n узлами

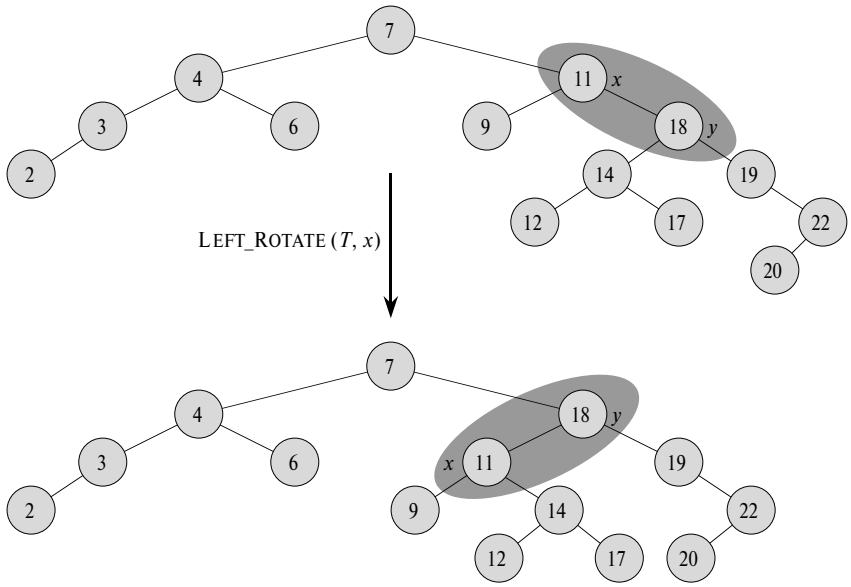


Рис. 13.3. Пример выполнения процедуры LEFT_ROTATE

с использованием $O(n)$ поворотов. (*Указание*: сначала покажите, что $n-1$ правых поворотов достаточно для преобразования дерева в правую цепочку.)

⋆ 13.2-5. Назовем бинарное дерево поиска T_1 ***правопреобразуемым*** в бинарное дерево поиска T_2, если можно получить T_2 из T_1 путем выполнения серии вызовов процедуры RIGHT_ROTATE. Приведите пример двух деревьев T_1 и T_2, таких что T_1 не является правопреобразуемым в T_2. Затем покажите, что если дерево T_1 является правопреобразуемым в T_2, то это преобразование можно выполнить при помощи $O\left(n^2\right)$ вызовов процедуры RIGHT_ROTATE.

13.3 Вставка

Вставка узла в красно-черное дерево с n узлами может быть выполнена за время $O(\lg n)$. Для вставки узла z в дерево T мы используем немного модифицированную версию процедуры TREE_INSERT (см. раздел 12.3), которая вставляет узел в дерево, как если бы это было обычное бинарное дерево поиска, а затем окрашивает его в красный цвет. Для того чтобы вставка сохраняла красно-черные свойства дерева, после нее вызывается вспомогательная процедура RB_INSERT_FIXUP, которая перекрашивает узлы и выполняет повороты. Вызов RB_INSERT(T, z) вставляет в красно-черное дерево T узел z с заполненным полем key:

```
RB_INSERT(T, z)
 1  y ← nil[T]
 2  x ← root[T]
 3  while x ≠ nil[T]
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else  x ← right[x]
 8  p[z] ← y
 9  if y = nil[T]
10     then root[T] ← z
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← RED
17  RB_INSERT_FIXUP(T, z)
```

Есть четыре отличия процедуры TREE_INSERT от процедуры RB_INSERT. Во-первых, все NIL в TREE_INSERT заменены на $nil\,[T]$. Во-вторых, для поддержки корректности структуры дерева в строках 14–15 процедуры RB_INSERT выполняется присвоение $nil\,[T]$ указателям $left\,[z]$ и $right\,[z]$. В третьих, в строке 16 мы назначаем узлу z красный цвет. И наконец, поскольку красный цвет z может вызвать нарушение одного из красно-черных свойств, в строке 17 вызывается вспомогательная процедура RB_INSERT_FIXUP(T, z), предназначение которой — восстановить красно-черные свойства дерева:

```
RB_INSERT_FIXUP(T, z)
 1  while color[p[z]] = RED
 2        do if p[z] = left[p[p[z]]]
 3              then y ← right[p[p[z]]]
 4                   if color[y] = RED
 5                      then color[p[z]] ← BLACK                ▷ Случай 1
 6                           color[y] ← BLACK                   ▷ Случай 1
 7                           color[p[p[z]]] ← RED               ▷ Случай 1
 8                           z ← p[p[z]]                        ▷ Случай 1
 9                      else if z = right[p[z]]
10                              then z ← p[z]                   ▷ Случай 2
11                                   LEFT_ROTATE(T, z)          ▷ Случай 2
12                           color[p[z]] ← BLACK                ▷ Случай 3
13                           color[p[p[z]]] ← RED               ▷ Случай 3
```

```
14                  RIGHT_ROTATE(T, p[p[z]])                ▷ Случай 3
15              else (то же, что и в "then", с заменой
                      left на right и наоборот)
16  color[root[T]] ← BLACK
```

Для того чтобы понять, как работает процедура RB_INSERT_FIXUP, мы разобьем рассмотрение кода на три основные части. Сначала мы определим, какие из красно-черных свойств нарушаются при вставке узла z и окраске его в красный цвет. Затем мы рассмотрим предназначение цикла **while** в строках 1–15. После этого мы изучим каждый из трех случаев[1], которые встречаются в этом цикле, и посмотрим, каким образом достигается цель в каждом случае. На рис. 13.4 показан пример выполнения процедуры RB_INSERT_FIXUP.

Какие из красно-черных свойств могут быть нарушены перед вызовом RB_ INSERT_FIXUP? Свойство 1 определенно выполняется (как и свойство 3), так как оба дочерних узла вставляемого узла являются ограничителями $nil\,[T]$. Свойство 5, согласно которому для каждого узла все пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов, также остается в силе, поскольку узел z замещает (черный) ограничитель, будучи при этом красным и имея черные дочерние узлы. Таким образом, может нарушаться только свойство 2, которое требует, чтобы корень красно-черного дерева был черным, и свойство 4, согласно которому красный узел не может иметь красного потомка. Оба нарушения возможны в силу того, что узел z после вставки окрашивается в красный цвет. Свойство 2 оказывается нарушенным, если узел z становится корнем, а свойство 4 — если родительский по отношению к z узел является красным. На рис. 13.4*а* показано нарушение свойства 4 после вставки узла z.

Цикл **while** в строках 1–15 сохраняет следующий инвариант, состоящий из трех частей.

В начале каждой итерации цикла:

а) узел z красный;

б) если $p\,[z]$ — корень дерева, то $p\,[z]$ — черный узел;

в) если имеется нарушение красно-черных свойств, то это нарушение только одно — либо нарушение свойства 2, либо свойства 4. Если нарушено свойство 2, то это вызвано тем, что корнем дерева является красный узел z; если нарушено свойство 4, то в этом случае красными являются узлы z и $p\,[z]$.

Часть *в*, в которой говорится о возможных нарушениях красно-черных свойств, наиболее важна для того, чтобы показать, что процедура RB_INSERT_FIXUP восстанавливает красно-черные свойства. Части *а* и *б* просто поясняют ситуацию.

[1] Случаи 2 и 3 не являются взаимоисключающими.

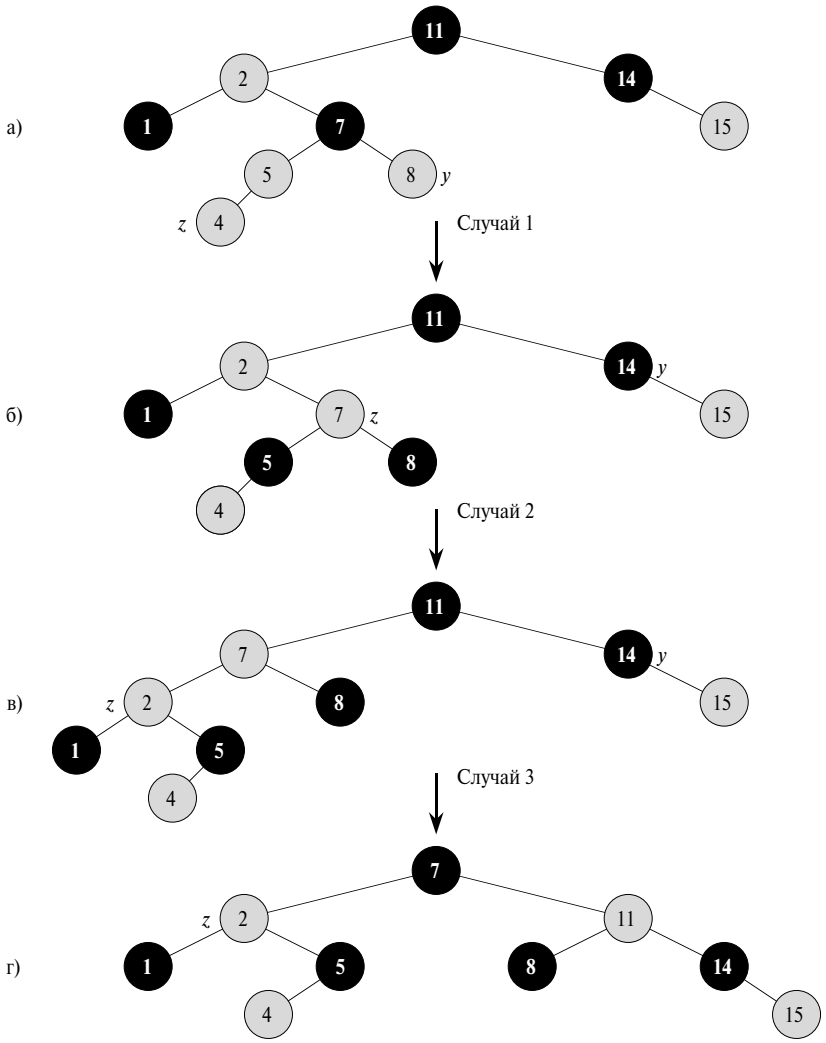


Рис. 13.4. Работа процедуры RB_INSERT_FIXUP

Поскольку мы сосредотачиваем свое рассмотрение только на узле z и узлах, находящихся в дереве вблизи него, полезно знать, что узел z — красный (часть *а*). Часть *б* используется для того, чтобы показать, что узел $p[p[z]]$, к которому мы обращаемся в строках 2, 3, 7, 8, 13 и 14, существует.

Вспомним, что мы должны показать, что инвариант цикла выполняется перед первой итерацией цикла, что любая итерация цикла сохраняет инвариант и что инвариант цикла обеспечивает выполнение требуемого свойства по окончании работы цикла.

Начнем с рассмотрения инициализации и завершения работы цикла, а затем, подробнее рассмотрев работу цикла, мы докажем, что он сохраняет инвариант цикла. Попутно мы покажем, что есть только два возможных варианта действий в каждой итерации цикла — указатель z перемещается вверх по дереву или выполняются некоторые повороты и цикл завершается.

Инициализация. Перед выполнением первой итерации цикла у нас имеется красно-черное дерево без каких-либо нарушений красно-черных свойств, к которому мы добавляем красный узел z. Покажем, что все части инварианта цикла выполняются к моменту вызова процедуры RB_INSERT_FIXUP.

а) В момент вызова процедуры RB_INSERT_FIXUP узел z — вставленный в дерево красный узел.

б) Если $p[z]$ — корневой узел дерева, то он является черным и не изменяется до вызова процедуры RB_INSERT_FIXUP.

в) Мы уже убедились в том, что красно-черные свойства 1, 3 и 5 сохраняются к моменту вызова процедуры RB_INSERT_FIXUP.

Если нарушается свойство 2, то красный корень должен быть добавленным в дерево узлом z, который при этом является единственным внутренним узлом дерева. Поскольку и родитель, и оба потомка z являются ограничителями, свойство 4 не нарушается. Таким образом, нарушение свойства 2 — единственное нарушение красно-черных свойств во всем дереве.

Если же нарушено свойство 4, то поскольку дочерние по отношению к z узлы являются черными ограничителями, а до вставки z никаких нарушений красно-черных свойств в дереве не было, нарушение заключается в том, что и z, и $p[z]$ — красные. Кроме этого, других нарушений красно-черных свойств не имеется.

Завершение. Как видно из кода, цикл завершает свою работу, когда $p[z]$ становится черным (если z — корневой узел, то $p[z]$ представляет собой черный ограничитель $nil[T]$). Таким образом, свойство 4 при завершении цикла не нарушается. В соответствии с инвариантом цикла, единственным нарушением красно-черных свойств может быть нарушение свойства 2. В строке 16 это свойство восстанавливается, так что по завершении работы процедуры RB_INSERT_FIXUP все красно-черные свойства дерева выполняются.

Сохранение. При работе цикла **while** следует рассмотреть шесть разных случаев, однако три из них симметричны другим трем; разница лишь в том, является ли родитель $p[z]$ левым или правым дочерним узлом по отношению к своему родителю $p[p[z]]$, что и выясняется в строке 2 (мы привели код только для ситуации, когда $p[z]$ является левым потомком). Узел $p[p[z]]$ существует,

поскольку, в соответствии с частью *б*) инварианта цикла, если $p[z]$ — корень дерева, то он черный. Поскольку цикл начинает работу, только если $p[z]$ — красный, то $p[z]$ не может быть корнем. Следовательно, $p[p[z]]$ существует.

Случай 1 отличается от случаев 2 и 3 цветом "брата" родительского по отношению к z узла, т.е. "дяди" узла z. После выполнения строки 3 указатель y указывает на дядю узла z — узел $right[p[p[z]]]$, и в строке 4 выясняется его цвет. Если y — красный, выполняется код для случая 1; в противном случае выполняется код для случаев 2 и 3. В любом случае, узел $p[p[z]]$ — черный, поскольку узел $p[z]$ — красный, а свойство 4 нарушается только между z и $p[z]$.

Случай 1: узел y красный.

На рис. 13.5 показана ситуация, возникающая в случае 1 (строки 5–8), когда и $p[z]$, и y — красные узлы. Поскольку $p[p[z]]$ — черный, мы можем исправить ситуацию, покрасив и $p[z]$, и y в черный цвет (после чего цвет красного родителя узла z становится черным, и нарушение между z и его родителем исчезает), а $p[p[z]]$ — в красный цвет, для того чтобы выполнялось свойство 5. После этого мы повторяем цикл **while** с узлом $p[p[z]]$ в качестве нового узла z. Указатель z перемещается, таким образом, на два уровня вверх.

На рис. 13.5*а* z — правый дочерний узел, а на рис. 13.5*б* — левый. Как видите, предпринимаемые в обоих случаях одни и те же действия приводят к одинаковому результату. Все поддеревья (α, β, γ и δ) имеют черные корни и одинаковое значение черной высоты. После перекраски свойство 5 сохраняется: все нисходящие пути от узла к листьям содержат одинаковое число черных узлов. После выполнения итерации новым узлом z становится узел $p[p[z]]$, и нарушение свойства 4 может быть только между новым узлом z и его родителем (который также может оказаться красным).

Теперь покажем, что в случае 1 инвариант цикла сохраняется. Обозначим через z узел z в текущей итерации, а через $z' = p[p[z]]$ — узел z, проверяемый в строке 1 при следующей итерации.

а) Поскольку в данной итерации цвет узла $p[p[z]]$ становится красным, в начале следующей итерации узел z' — красный.

б) Узел $p[z']$ в текущей итерации — $p[p[p[z]]]$, и цвет данного узла в пределах данной итерации не изменяется. Если это корневой узел, то его цвет до начала данной итерации был черным и остается таковым в начале следующей итерации.

в) Мы уже доказали, что в случае 1 свойство 5 сохраняется; кроме того, понятно, что при выполнении итерации не возникает нарушение свойств 1 или 3.

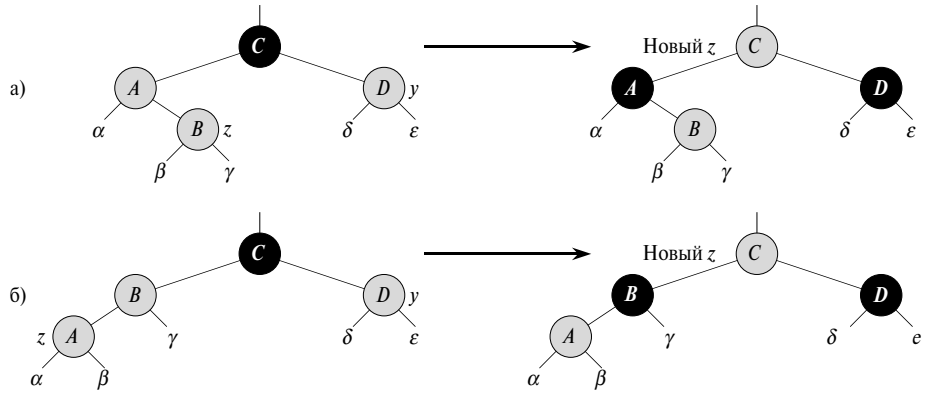


Рис. 13.5. Случай 1 процедуры RB_INSERT_FIXUP

Если узел z' в начале очередной итерации является корнем, то код, соответствующий случаю 1, корректирует единственное нарушение свойства 4. Поскольку узел z' — красный и корневой, единственным нарушенным становится свойство 2, причем это нарушение связано с узлом z'.

Если узел z' в начале следующей итерации корнем не является, то код, соответствующий случаю 1, не вызывает нарушения свойства 2. Этот код корректирует единственное нарушение свойства 4, имеющееся перед выполнением итерации. Коррекция выражается в том, что узел z' становится красным, в то время как цвет узла $p\,[z']$ не изменяется. Если узел $p\,[z']$ был черным, то свойство 4 не нарушается; если же этот узел был красным, то окрашивание узла z' в красный цвет приводит к нарушению свойства 4 между узлами z' и $p\,[z']$.

Случай 2: узел y черный и z — правый потомок.

Случай 3: узел y черный и z — левый потомок.

В случаях 2 и 3 цвет узла y, являющегося "дядей" узла z, черный. Эти два случая отличаются друг от друга тем, что z является левым или правым дочерним узлом по отношению к родительскому. Строки 10–11 псевдокода соответствуют случаю 2, который показан на рис. 13.6 вместе со случаем 3. В случае 2 узел z является правым потомком своего родительского узла. Мы используем левый поворот для преобразования сложившейся ситуации в случай 3 (строки 12–14), когда z является левым потомком. Поскольку и z, и $p\,[z]$ — красные узлы, поворот не влияет ни на черную высоту узлов, ни на выполнение свойства 5. Когда мы приходим к случаю 3 (либо непосредственно, либо поворотом из случая 2), узел y имеет черный цвет (поскольку иначе мы бы получили случай 1). Кроме того, обязательно существует узел

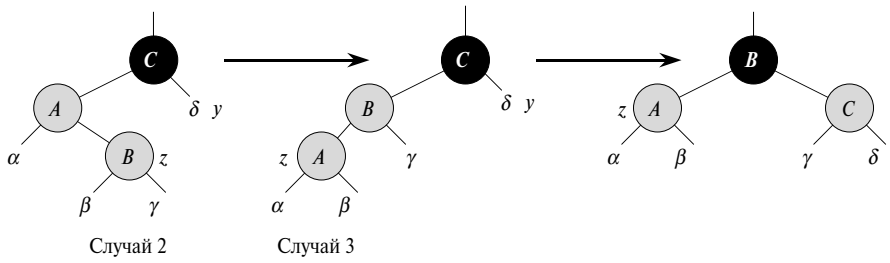


Рис. 13.6. Случаи 2 и 3 процедуры RB_INSERT_FIXUP

$p\,[p\,[z]]$, так как мы доказали, что этот узел существовал при выполнение строк 2 и 3, а также что после перемещения узла z на один узел вверх в строке 10 с последующим опусканием в строке 11 узел $p\,[p\,[z]]$ остается неизменным. В случае 3 мы выполняем ряд изменений цвета и правых поворотов, которые сохраняют свойство 5. После этого, так как у нас нет двух идущих подряд красных узлов, работа процедуры завершается. Больше тело цикла **while** не выполняется, так как узел $p\,[z]$ теперь черный.

Теперь покажем, что случаи 2 и 3 сохраняют инвариант цикла. (Как мы только что доказали, перед следующей проверкой в строке 1 узел $p\,[z]$ будет черным и тело цикла больше выполняться не будет.)

а) В случае 2 выполняется присвоение, после которого z указывает на красный узел $p\,[z]$. Никаких других изменений z или его цвета в случаях 2 и 3 не выполняется.

б) В случае 3 узел $p\,[z]$ делается черным, так что если $p\,[z]$ в начале следующей итерации является корнем, то этот корень — черный.

в) Как и в случае 1, в случае 2 и 3 свойства 1, 3 и 5 сохраняются.

   Поскольку узел z в случаях 2 и 3 не является корнем, нарушение свойства 2 невозможно. Случаи 2 и 3 не могут приводить к нарушению свойства 2, поскольку при повороте в случае 3 красный узел становится дочерним по отношению к черному.

   Таким образом, случаи 2 и 3 приводят к коррекции нарушения свойства 4, при этом не внося никаких новых нарушений красно-черных свойств.

Показав, что при любой итерации инвариант цикла сохраняется, мы тем самым показали, что процедура RB_INSERT_FIXUP корректно восстанавливает красно-черные свойства дерева.

Анализ

Чему равно время работы процедуры RB_INSERT? Поскольку высота красно-черного дерева с n узлами равна $O(\lg n)$, выполнение строк 1—16 процедуры RB_INSERT требует $O(\lg n)$ времени. В процедуре RB_INSERT_FIXUP цикл **while** повторно выполняется только в случае 1, и в этом случае указатель z перемещается вверх по дереву на два уровня. Таким образом, общее количество возможных выполнений тела цикла **while** равно $O(\lg n)$. Таким образом, общее время работы процедуры RB_INSERT равно $O(\lg n)$. Интересно, что в ней никогда не выполняется больше двух поворотов, поскольку цикл **while** в случаях 2 и 3 завершает работу.

Упражнения

13.3-1. В строке 16 процедуры RB_INSERT мы окрашиваем вновь вставленный узел в красный цвет. Заметим, что если бы мы окрашивали его в черный цвет, то свойство 4 не было бы нарушено. Так почему же мы не делаем этого?

13.3-2. Изобразите красно-черные деревья, которые образуются при последовательной вставке ключей 41, 38, 31, 12, 19 и 8 в изначально пустое красно-черное дерево.

13.3-3. Предположим, что черная высота каждого из поддеревьев α, β, γ, δ и ε на рис. 13.5 и 13.6 равна k. Найдите черные высоты каждого узла на этих рисунках, чтобы убедиться, что свойство 5 сохраняется при показанных преобразованиях.

13.3-4. Профессор озабочен вопросом, не может ли в процедуре RB_INSERT_FIXUP произойти присвоение значение RED узлу $nil[T]$ — ведь в этом случае проверка в строке 1 не вызовет окончания работы цикла в случае, если z — корень дерева. Покажите, что страхи профессора безосновательны, доказав, что процедура RB_INSERT_FIXUP никогда не окрашивает $nil[T]$ в красный цвет.

13.3-5. Рассмотрим красно-черное дерево, образованное вставкой n узлов при помощи процедуры RB_INSERT. Докажите, что если $n > 1$, то в дереве имеется как минимум один красный узел.

13.3-6. Предложите эффективную реализацию процедуры RB_INSERT в случае, когда представление красно-черных деревьев не включает указатель на родительский узел.

13.4 Удаление

Как и остальные базовые операции над красно-черными деревьями с n узлами, удаление узла выполняется за время $O(\lg n)$. Удаление оказывается несколько более сложной задачей, чем вставка.

Процедура RB_DELETE представляет собой немного измененную процедуру TREE_DELETE из раздела 12.3. После удаления узла в ней вызывается вспомогательная процедура RB_DELETE_FIXUP, которая изменяет цвета и выполняет повороты для восстановления красно-черных свойств дерева:

```
RB_DELETE(T, z)
 1  if left[z] = nil[T] или right[z] = nil[T]
 2     then y ← z
 3     else y ← TREE_SUCCESSOR(z)
 4  if left[y] ≠ nil[T]
 5     then x ← left[y]
 6     else x ← right[y]
 7  p[x] ← p[y]
 8  if p[y] = nil[T]
 9     then root[T] ← x
10     else if y = left[p[y]]
11             then left[p[y]] ← x
12             else right[p[y]] ← x
13  if y ≠ z
14     then key[z] ← key[y]
15          Копируем сопутствующие данные y в z
16  if color[y] = BLACK
17     then RB_DELETE_FIXUP(T, x)
18  return y
```

Имеется три различия между процедурами TREE_DELETE и RB_DELETE. Во-первых, все ссылки на NIL в TREE_DELETE заменены в RB_DELETE ссылками на ограничитель $nil\,[T]$. Во-вторых, удалена проверка в строке 7 процедуры TREE_DELETE (равно ли x NIL), и присвоение $p\,[x] \leftarrow p\,[y]$ выполняется в процедуре RB_DELETE безусловно. Таким образом, если x является ограничителем $nil\,[T]$, то его указатель на родителя указывает на родителя извлеченного из дерева узла y. В-третьих, в строках 16–17 процедуры RB_DELETE в случае, если узел y — черный, выполняется вызов вспомогательной процедуры RB_DELETE_FIXUP. Если узел y — красный, красно-черные свойства при извлечении y из дерева сохраняются в силу следующих причин:

- никакая черная высота в дереве не изменяется;
- никакие красные узлы не становятся соседними;

- так как y не может быть корнем в силу своего цвета, корень остается черным.

Узел x, который передается в качестве параметра во вспомогательную процедуру RB_DELETE_FIXUP, является одним из двух узлов: либо это узел, который был единственным потомком y перед извлечением последнего из дерева (если у y был дочерний узел, не являющийся ограничителем), либо, если у узла y нет дочерних узлов, x является ограничителем $nil\,[T]$. В последнем случае безусловное присвоение в строке 7 гарантирует, что родительским по отношению к x узлом становится узел, который ранее был родителем y, независимо от того, является ли x внутренним узлом с реальным ключом или ограничителем $nil\,[T]$.

Теперь мы можем посмотреть, как вспомогательная процедура RB_INSERT_FIXUP справляется со своей задачей восстановления красно-черных свойств дерева поиска.

```
RB_DELETE_FIXUP(T, x)
 1  while x ≠ root[T] и color[x] = BLACK
 2        do if x = left[p[x]]
 3              then w ← right[p[x]]
 4                   if color[w] = RED
 5                     then color[w] ← BLACK                          ▷ Случай 1
 6                          color[p[x]] ← RED                         ▷ Случай 1
 7                          LEFT_ROTATE(T, p[x])                      ▷ Случай 1
 8                          w ← right[p[x]]                           ▷ Случай 1
 9                   if color[left[w]] = BLACK и color[right[w]] = BLACK
10                     then color[w] ← RED                           ▷ Случай 2
11                          x ← p[x]                                 ▷ Случай 2
12                     else if color[right[w]] = BLACK
13                             then color[left[w]] ← BLACK           ▷ Случай 3
14                                  color[w] ← RED                   ▷ Случай 3
15                                  RIGHT_ROTATE(T, w)               ▷ Случай 3
16                                  w ← right[p[x]]                  ▷ Случай 3
17                          color[w] ← color[p[x]]                   ▷ Случай 4
18                          color[p[x]] ← BLACK                      ▷ Случай 4
19                          color[right[w]] ← BLACK                  ▷ Случай 4
20                          LEFT_ROTATE(T, p[x])                     ▷ Случай 4
21                          x ← root[T]                              ▷ Случай 4
22             else (то же, что и в "then", с заменой left на right и наоборот)
23  color[x] ← BLACK
```

Если извлекаемый из дерева в процедуре RB_DELETE узел y черный, то могут возникнуть три проблемы. Во-первых, если y был корнем, а теперь корнем стал красный потомок y, нарушается свойство 2. Во-вторых, если и x, и $p\,[y]$ (который теперь является также $p\,[x]$) были красными, то нарушается свойство 4. И, в-третьих, удаление y приводит к тому, что все пути, проходившие через y,

теперь имеют на один черный узел меньше. Таким образом, для всех предков y оказывается нарушенным свойство 5. Мы можем исправить ситуацию, утверждая, что узел x — "сверхчерный", т.е. при рассмотрении любого пути, проходящего через x, добавлять дополнительную 1 к количеству черных узлов. При такой интерпретации свойство 5 остается выполняющимся. При извлечении черного узла y мы передаем его "черноту" его потомку. Проблема заключается в том, что теперь узел x не является ни черным, ни красным, что нарушает свойство 1. Вместо этого узел x окрашен либо "дважды черным", либо "красно-черным" цветом, что дает при подсчете черных узлов на пути, содержащем x, вклад, равный, соответственно, 2 или 1. Атрибут $color$ узла x при этом остается равен либо BLACK (если узел дважды черный), либо RED (если узел красно-черный). Другими словами, цвет узла не соответствует его атрибуту $color$.

Процедура RB_DELETE_FIXUP восстанавливает свойства 1, 2 и 4. В упражнениях 13.4-1 и 13.4-2 требуется показать, что данная процедура восстанавливает свойства 2 и 4, так что в оставшейся части данного раздела мы сконцентрируем свое внимание на свойстве 1. Цель цикла **while** в строках 1–22 состоит в том, чтобы переместить дополнительную черноту вверх по дереву до тех пор, пока не выполнится одно из перечисленных условий.

1. x указывает на красно-черный узел — в этом случае мы просто делаем узел x "единожды черным" в строке 23.
2. x указывает на корень — в этом случае мы просто убираем излишнюю черноту.
3. Можно выполнить некоторые повороты и перекраску, после которых двойная чернота будет устранена.

Внутри цикла **while** x всегда указывает на дважды черную вершину, не являющуюся корнем. В строке 2 мы определяем, является ли x левым или правым дочерним узлом своего родителя $p[x]$. Далее приведен подробный код для ситуации, когда x — левый потомок. Для правого потомка код аналогичен и симметричен, и скрыт за описанием в строке 22. Указатель w указывает на второго потомка родителя x. Поскольку узел x дважды черный, узел w не может быть $nil[T]$ — в противном случае количество черных узлов на пути от $p[x]$ к (единожды черному) листу было бы меньше, чем количество черных узлов на пути от $p[x]$ к x.

Четыре разных возможных случая[2] показаны на рис. 13.7. Перед тем как приступить к детальному рассмотрению каждого случая, убедимся, что в каждом из случаев преобразования сохраняется свойство 5. Ключевая идея заключается в необходимости убедиться, что при преобразованиях в каждом случае количество черных узлов (включая дополнительную черноту в x) на пути от корня включительно до каждого из поддеревьев α, β, ..., ζ остается неизменным. Таким

[2]Как и в процедуре RB_INSERT_FIXUP, случаи не являются взаимоисключающими.

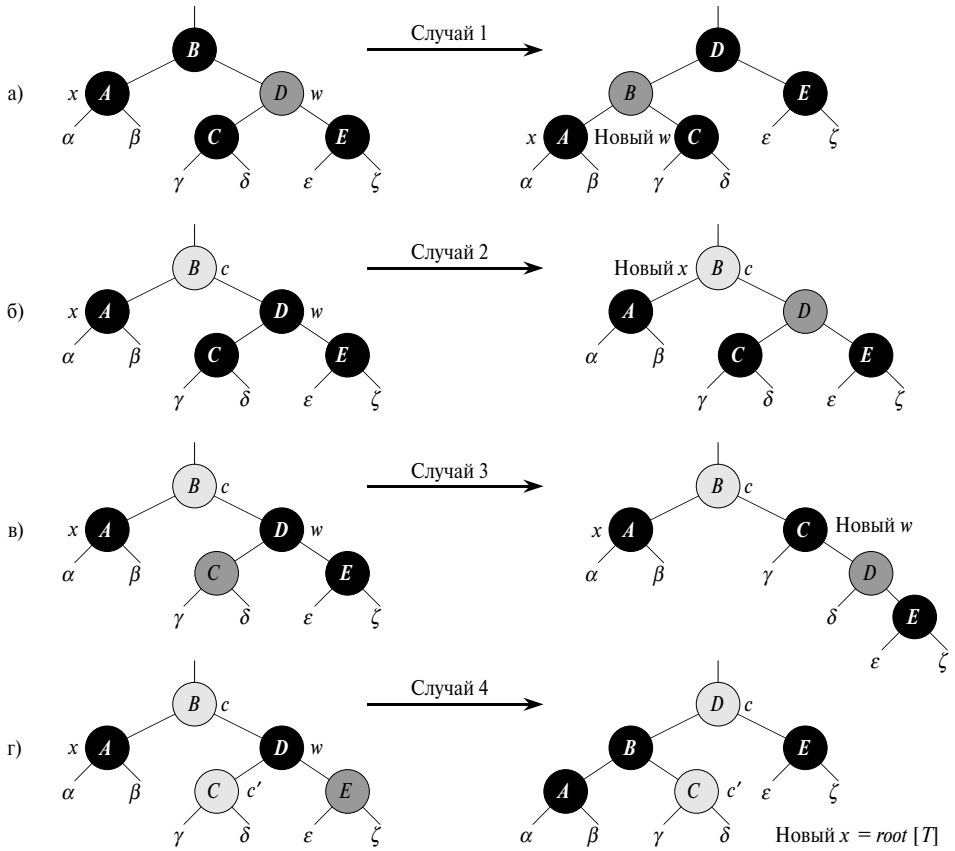


Рис. 13.7. Возможные ситуации, возникающие в цикле **while** процедуры RB_DELETE_FIXUP. Темные узлы имеют цвет BLACK, темно-серые — RED, а светлые могут иметь любой цвет (на рисунке показан как c и c'). Греческими буквами показаны произвольные поддеревья

образом, если свойство 5 выполнялось до преобразования, то оно выполняется и после него. Например, на рис. 13.7*а*, который иллюстрирует случай 1, количество черных узлов на пути от корня до поддеревьев α и β равно 3, как до, так и после преобразования (не забудьте о том, что узел x — дважды черный). Аналогично, количество черных узлов на пути от корня до любого из поддеревьев γ, δ, ε и ζ равно 2, как до, так и после преобразования. На рис. 13.7*б* подсчет должен включать значение c, равное значению атрибута $color$ корня показанного поддерева, которое может быть либо RED, либо BLACK. Так, на пути от корня до поддерева α количество черных узлов равно 2 плюс величина, равная 1, если c равно BLACK, и 0, если c равно RED; эта величина одинакова до и после выполнения преобразований. В такой ситуации после преобразования новый узел x имеет атрибут $color$, равный c, но реально это либо красно-черный узел (если $c =$ RED),

либо дважды черный (если $c = \text{BLACK}$). Прочие случаи могут быть проверены аналогичным способом (см. упражнение 13.4-5).

Случай 1: узел w красный.

Случай 1 (строки 5–8 процедуры RB_DELETE_FIXUP и рис. 13.7*а*) возникает, когда узел w ("брат" узла x) — красный. Поскольку w должен иметь черных потомков, можно обменять цвета w и $p[x]$, а затем выполнить левый поворот вокруг $p[x]$ без нарушения каких-либо красно-черных свойств. Новый "брат" x, до поворота бывший одним из потомков w, теперь черный. Таким путем случай 1 приводится к случаю 2, 3 или 4.

Случаи 2, 3 и 4 возникают при черном узле w и отличаются друг от друга цветами дочерних по отношению к w узлов.

Случай 2: узел w черный, оба его дочерних узла черные.

В этом случае (строки 10–11 процедуры RB_DELETE_FIXUP и рис. 13.7*б*) оба дочерних узла w черные. Поскольку узел w также черный, мы можем забрать черную окраску у x и w, сделав x единожды черным, а w — красным. Для того чтобы компенсировать удаление черной окраски у x и w, мы можем добавить дополнительный черный цвет узлу $p[x]$, который до этого мог быть как красным, так и черным. После этого будет выполнена следующая итерация цикла, в которой роль x будет играть текущий узел $p[x]$. Заметим, что если мы переходим к случаю 2 от случая 1, новый узел x — красно-черный, поскольку исходный узел $p[x]$ был красным. Следовательно, значение c атрибута $color$ нового узла x равно RED и цикл завершается при проверке условия цикла. После этого новый узел x окрашивается в обычный черный цвет в строке 23.

Случай 3: узел w черный, его левый дочерний узел красный, а правый — черный.

В этом случае (строки 13–16 процедуры RB_DELETE_FIXUP и рис. 13.7*в*) узел w черный, его левый дочерний узел красный, а правый — черный. Мы можем обменять цвета w и $left[w]$, а затем выполнить правый поворот вокруг w без нарушения каких-либо красно-черных свойств. Новым "братом" узла x после этого будет черный узел с красным правым дочерним узлом, и таким образом мы привели случай 3 к случаю 4.

Случай 4: узел w черный, его правый дочерний узел красный.

В этом случае (строки 17–21 процедуры RB_DELETE_FIXUP и рис. 13.7*г*) узел w черный, а его правый дочерний узел — красный. Выполняя обмен цветов и левый поворот вокруг $p[x]$, мы можем устранить излишнюю черноту в x, делая его просто черным, без нарушения каких-либо красно-черных свойств. Присвоение x указателя на корень дерева приводит к завершению работы при проверке условия цикла при следующей итерации.

Анализ

Чему равно время работы процедуры RB_DELETE? Поскольку высота дерева с n узлами равна $O(\lg n)$, общее время работы процедуры без выполнения вспомогательной процедуры RB_DELETE_FIXUP равно $O(\lg n)$. В процедуре RB_DELETE_FIXUP в случаях 1, 3 и 4 завершение работы происходит после выполнения постоянного числа изменений цвета и не более трех поворотов. Случай 2 — единственный, после которого возможно выполнение очередной итерации цикла **while**, причем указатель x перемещается вверх по дереву не более чем $O(\lg n)$ раз, и никакие повороты при этом не выполняются. Таким образом, время работы процедуры RB_DELETE_FIXUP составляет $O(\lg n)$, причем она выполняет не более трех поворотов. Общее время работы процедуры RB_DELETE, само собой разумеется, также равно $O(\lg n)$.

Упражнения

13.4-1. Покажите, что после выполнения процедуры RB_DELETE_FIXUP корень дерева должен быть черным.

13.4-2. Покажите, что если в процедуре RB_DELETE и x, и $p[x]$ красные, то при вызове RB_DELETE_FIXUP(T, x) свойство 4 будет восстановлено.

13.4-3. В упражнении 13.3-2 вы построили красно-черное дерево, которое является результатом последовательной вставки ключей 41, 38, 31, 12, 19 и 8 в изначально пустое дерево. Покажите теперь, что в результате последовательного удаления ключей 8, 12, 19, 31, 38 и 41 будут получаться красно-черные деревья.

13.4-4. В каких строках кода процедуры RB_DELETE_FIXUP мы можем обращаться к ограничителю $nil[T]$ или изменять его?

13.4-5. Для каждого из случаев, показанных на рис. 13.7, подсчитайте количество черных узлов на пути от корня показанного поддерева до каждого из поддеревьев α, β, ..., ζ и убедитесь, что оно не меняется при выполнении преобразований. Если узел имеет атрибут $color$, равный c или c', воспользуйтесь функцией $count(c)$, которая для красного узла равна 0, а для черного — 1.

13.4-6. Профессор озабочен вопросом, не может ли узел $p[x]$ не быть черным в начале случая 1 в процедуре RB_DELETE_FIXUP. Если профессор прав, то строки 5–6 процедуры ошибочны. Покажите, что в начале случая 1 узел $p[x]$ не может не быть черным, так что профессору нечего волноваться.

13.4-7. Предположим, что узел x вставлен в красно-черное дерево при помощи процедуры RB_INSERT, после чего немедленно удален из этого дерева. Будет ли получившееся в результате вставки и удаления красно-черное дерево таким же, как и исходное? Обоснуйте ваш ответ.

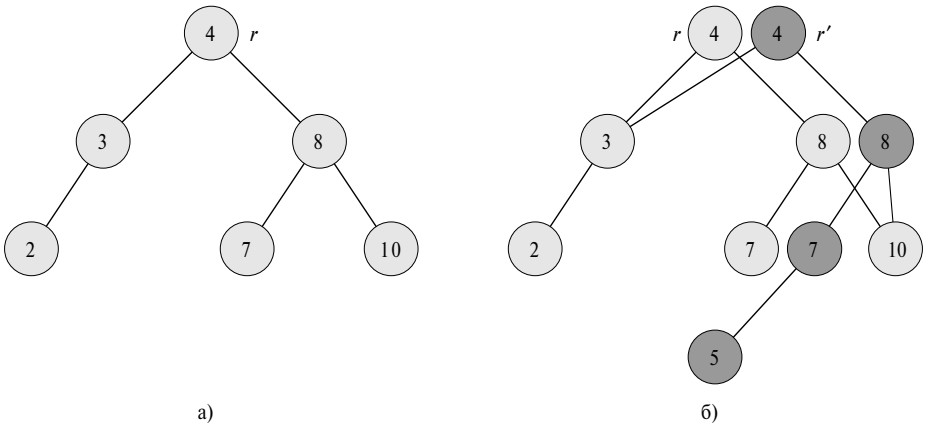


Рис. 13.8. Бинарное дерево поиска с ключами 2, 3, 4, 7, 8 и 10 (а) и перманентное бинарное дерево поиска, получающееся в процессе вставки ключа 5 (б)

Задачи

13-1. **Перманентные динамические множества**

Иногда полезно сохранять предыдущие версии динамического множества в процессе его обновления. Такие множества называются ***перманентными*** (persistent). Один из способов реализации перманентного множества состоит в копировании всего множества при внесении в него изменений, однако такой подход может существенно замедлить работу программы и вызвать перерасход памяти. Зачастую эту задачу можно решить гораздо более эффективно.

Рассмотрим перманентное множество S с операциями INSERT, DELETE и SEARCH, которое реализуется с использованием бинарных деревьев поиска, как показано на рис. 13.8*а*. Для каждой версии множества мы храним отдельный корень. Для добавления ключа 5 в множество создается узел с ключом 5, который становится левым дочерним узлом нового узла с ключом 7, так как менять существующий узел с ключом 7 мы не можем. Аналогично, новый узел с ключом 7 становится левым потомком нового узла с ключом 8, правым потомком которого является существующий узел с ключом 10. Новый узел с ключом 8 становится, в свою очередь, правым потомком нового корня r' с ключом 4, левым потомком которого является существующий узел с ключом 3. Таким образом, мы копируем только часть дерева, а в остальном используем старое дерево, как показано на рис. 13.8*б*.

Предположим, что каждый узел дерева имеет поля key, $left$ и $right$, но не имеет поля с указателем на родительский узел (см. также упражнение 13.3-6).

а) Определите, какие узлы перманентного бинарного дерева поиска должны быть изменены при вставке в него ключа k или удалении узла y в общем случае.

б) Напишите процедуру PERSISTENT_TREE_INSERT, которая для данного перманентного дерева T и вставляемого ключа k возвращает новое перманентное дерево T', получающееся в результате вставки k в T.

в) Если высота перманентного бинарного дерева поиска T равна h, сколько времени будет работать ваша реализация PERSISTENT_TREE_INSERT и какие требования к памяти она предъявляет? (Количество требуемой памяти пропорционально количеству новых узлов.)

г) Предположим, что в каждом узле имеется поле указателя на родительский узел. В этом случае процедура PERSISTENT_TREE_INSERT должна выполнять дополнительное копирование. Докажите, что в этом случае время работы процедуры и объем необходимой памяти равны $\Omega(n)$, где n — количество узлов в дереве.

д) Покажите, как можно использовать красно-черные деревья для того, чтобы при вставке и удалении в наихудшем случае время работы процедуры и объем необходимой памяти были равны $O(\lg n)$.

13-2. **Операция объединения красно-черных деревьев**

Операция ***объединения*** (join) применяется к двум динамическим множествам S_1 и S_2 и элементу x (такому, что для любых $x_1 \in S_1$ и $x_2 \in S_2$ выполняется неравенство $key[x_1] \leqslant key[x] \leqslant key[x_2]$). Результатом операции является множество $S = S_1 \cup \{x\} \cup S_2$. В данной задаче мы рассмотрим реализацию операции объединения красно-черных деревьев.

а) Будем хранить черную высоту красно-черного дерева T как атрибут $bh[T]$. Покажите, что этот атрибут может поддерживаться процедурами RB_INSERT и RB_DELETE без использования дополнительной памяти в узлах дерева и без увеличения асимптотического времени работы процедур. Покажите, что при спуске по дереву T можно определить черную высоту каждого посещаемого узла за время $O(1)$ на каждый посещенный узел.

Мы хотим реализовать операцию RB_JOIN(T_1, x, T_2), которая, разрушая деревья T_1 и T_2, возвращает красно-черное дерево $T = T_1 \cup \{x\} \cup T_2$. Пусть n — общее количество узлов в деревьях T_1 и T_2.

б) Считая, что $bh[T_1] \geqslant bh[T_2]$, разработайте алгоритм, который за время $O(\lg n)$ находит среди черных узлов дерева T_1, черная высота которых равна $bh[T_2]$, узел y с наибольшим значением ключа.

в) Пусть T_y — поддерево с корнем y. Опишите, как заменить T_y на $T_y \cup \{x\} \cup T_2$ за время $O(1)$ с сохранением свойства бинарного дерева поиска.

г) В какой цвет надо окрасить x, чтобы сохранились красно-черные свойства 1, 3 и 5? Опишите, как восстановить свойства 2 и 4 за время $O(\lg n)$.

д) Докажите, что предположение в пункте *б*) данной задачи не приводит к потере общности. Опишите симметричную ситуацию, возникающую при $bh[T_1] \leqslant bh[T_2]$.

е) Покажите, что время работы процедуры RB_JOIN равно $O(\lg n)$.

13-3. **AVL-деревья**

AVL-дерево представляет собой бинарное дерево поиска со ***сбалансированной высотой***: для каждого узла x высота левого и правого поддеревьев x отличается не более чем на 1. Для реализации AVL-деревьев мы воспользуемся дополнительным полем в каждом узле дерева $h[x]$, в котором хранится высота данного узла. Как и в случае обычных деревьев поиска, мы считаем, что $root[T]$ указывает на корневой узел.

а) Докажите, что AVL-дерево с n узлами имеет высоту $O(\lg n)$. (*Указание*: докажите, что в AVL-дереве высотой h имеется как минимум F_h узлов, где $F_h - h$-е число Фибоначчи.)

б) Для вставки узла в AVL-дерево он сначала размещается в соответствующем месте бинарного дерева поиска. После этого дерево может оказаться несбалансированным, в частности, высота левого и правого потомков некоторого узла может отличаться на 2. Разработайте процедуру BALANCE(x), которая получает в качестве параметра поддерево с корнем в узле x, левый и правый потомки которого сбалансированы по высоте и имеют высоту, отличающуюся не более чем на 2 (т.е. $|h[right[x]] - h[left[x]]| \leqslant 2$), и изменяет его таким образом, что поддерево с корнем в узле x становится сбалансированным по высоте. (*Указание*: воспользуйтесь поворотами.)

в) Используя решение подзадачи *б*, разработайте рекурсивную процедуру AVL_INSERT(x, z), которая получает в качестве параметров узел x в AVL-дереве и вновь созданный узел z (с заполненным полем ключа) и вставляет z в поддерево, корнем которого является узел x, сохраняя при этом свойство, заключающееся в том, что x — корень AVL-дерева. Как и в случае процедуры TREE_INSERT из раздела 12.3, считаем, что поле $key[z]$ заполнено и что $left[z] = right[z] =$ NIL. Кроме того, полагаем, что $h[z] = 0$. Таким образом, для вставки узла z в AVL-дерево T мы должны осуществить вызов AVL_INSERT$(root[T], z)$.

г) Покажите, что время работы операции AVL_INSERT для AVL-дерева с n узлами равно $O(\lg n)$, и она выполняет $O(1)$ поворотов.

13-4. **Дерамиды**[3]

Если мы вставляем в бинарное дерево поиска набор из n элементов, то получающееся в результате дерево может оказаться ужасно несбалансированным, что приводит к большому времени поиска. Однако, как мы видели в разделе 12.4, случайные бинарные деревья поиска обычно оказываются достаточно сбалансированными. Таким образом, стратегия построения сбалансированного дерева для фиксированного множества элементов состоит в случайной их перестановке с последующей вставкой в дерево.

Но что делать, если все элементы недоступны одновременно? Если мы получаем элементы по одному, можем ли мы построить из них случайное бинарное дерево поиска?

Рассмотрим структуру данных, которая позволяет положительно ответить на этот вопрос. ***Дерамида*** представляет собой бинарное дерево поиска с модифицированным способом упорядочения узлов. Пример дерамиды показан на рис. 13.9. Как обычно, каждый узел x в дереве имеет значение $key\,[x]$. Кроме того, мы назначим каждому узлу значение *приоритета* $priority\,[x]$, которое представляет собой случайное число, выбираемое для каждого узла независимо от других. Мы считаем, что все приоритеты и все ключи в дереве различны. Узлы дерамиды упорядочены таким образом, чтобы ключи подчинялись свойству бинарных деревьев поиска, а приоритеты — свойству неубывающей пирамиды.

- Если v является левым потомком u, то $key\,[v] < key\,[u]$.
- Если v является правым потомком u, то $key\,[v] > key\,[u]$.
- Если v является потомком u, то $priority\,[v] > priority\,[u]$.

Именно эта комбинация свойств дерева и пирамиды и дала название “дерамида” (treap).

Помочь разобраться с дерамидами может следующая интерпретация. Предположим, что мы вставляем в дерамиду узлы $x_1, x_2, \ldots, x_n$ со связанными с ними ключами. Тогда получающаяся в результате дерамида представляет собой дерево, которое получилось бы в результате вставки узлов в обычное бинарное дерево поиска в порядке, определяемом (случайно выбранными) приоритетами, т.е. $priority\,[x_i] < priority\,[x_j]$ означает, что узел x_i был вставлен до узла x_j.

а) Покажите, что для каждого заданного множества узлов $x_1, x_2, \ldots, x_n$ со связанными с ними ключами и приоритетами (все ключи и приоритеты различны) существует единственная дерамида.

[3] В оригинале — treaps; слово, образованное из двух — tree и heap. Аналогично, из “дерева” и “пирамиды” получился русский эквивалент. — *Прим. ред.*

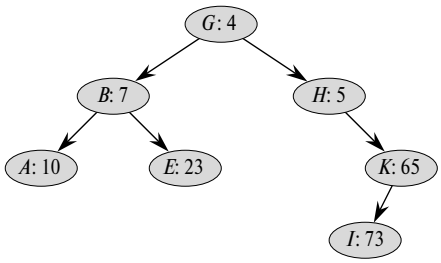


Рис. 13.9. Дерамида. Каждый узел имеет метку $key\,[x] : priority\,[x]$

б) Покажите, что математическое ожидание высоты дерамиды равно $\Theta(\lg n)$ и, следовательно, время поиска заданного значения в дерамиде составляет $\Theta(\lg n)$.

Рассмотрим процесс вставки нового узла в существующую дерамиду. Первое, что мы делаем — это назначаем новому узлу случайное значение приоритета. Затем мы вызываем процедуру TREAP_INSERT, работа которой показана на рис. 13.10. На рис. 13.10*а* показана исходная дерамида; на рис. 13.10*б* — та же дерамида после вставки узла с ключом C и приоритетом 25. Рис. 13.10*в–г* изображают промежуточные стадии вставки узла с ключом D и приоритетом 9, а рис. 13.10*д* — получившуюся в результате дерамиду. На рис. 13.10*е* показана дерамида после вставки узла с ключом F и приоритетом 2.

в) Объясните, как работает процедура TREAP_INSERT. Поясните принцип ее работы обычным русским языком и приведите ее псевдокод. (*Указание*: выполните обычную вставку в бинарное дерево поиска, а затем выполните повороты для восстановления свойства неубывающей пирамиды.)

г) Покажите, что математическое ожидание времени работы процедуры TREAP_INSERT составляет $\Theta(\lg n)$.

Процедура TREAP_INSERT выполняет поиск с последующей последовательностью поворотов. Хотя эти две операции имеют одно и то же математическое ожидание времени работы, на практике стоимость этих операций различна. Поиск считывает информацию из дерамиды, никак не изменяя ее. Повороты, напротив, приводят к изменению указателей на дочерние и родительские узлы в дерамиде. На большинстве компьютеров операция чтения существенно более быстрая, чем операция записи. Соответственно, желательно, чтобы поворотов при выполнении процедуры TREAP_INSERT было как можно меньше. Мы покажем, что ожидаемое количество выполняемых процедурой поворотов ограничено константой.

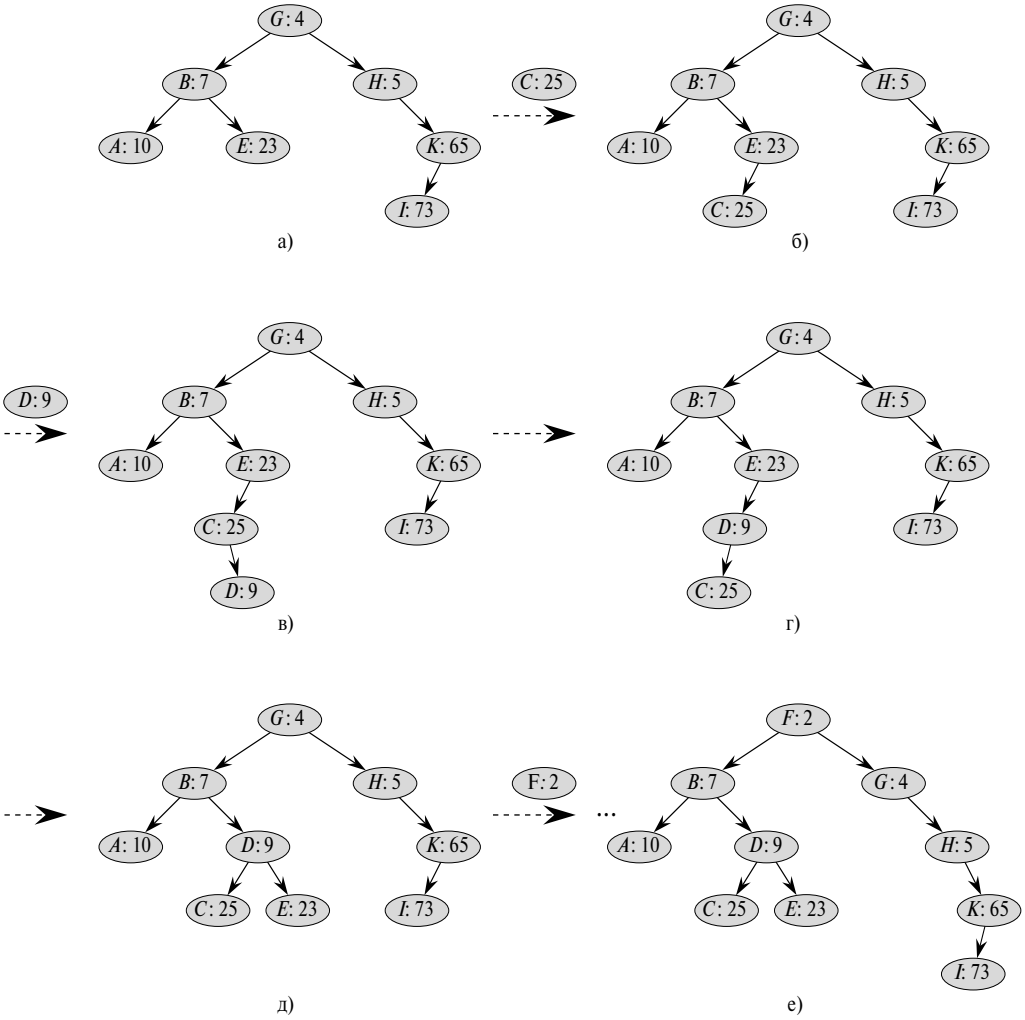


Рис. 13.10. Работа алгоритма TREAP_INSERT

Для этого нам надо дать несколько определений, проиллюстрированных на рис. 13.11. ***Левый хребет*** бинарного дерева поиска T представляет собой путь от корня к узлу с минимальным значением ключа. Другими словами, левый хребет представляет собой путь, состоящий только из левых ребер. Соответственно, ***правый хребет*** представляет собой путь, состоящий только из правых ребер. Длина хребта — это количество составляющих его узлов.

д) Рассмотрите дерамиду T после того, как в нее при помощи процедуры TREAP_INSERT вставляется узел x. Пусть C — длина правого

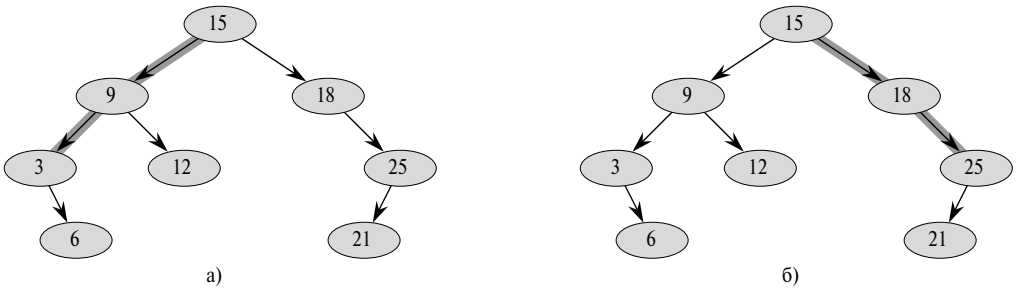


Рис. 13.11. Хребты в бинарных деревьях поиска (*а* — левый хребет, *б* — правый)

хребта левого поддерева x, а D — длина левого хребта правого поддерева x. Докажите, что общее количество поворотов, которые были выполнены в процессе вставки x, равно $C + D$.

Теперь мы вычислим математическое ожидание значений C и D. Без потери общности будем считать, что ключи представляют собой натуральные числа 1, 2, ..., n, поскольку мы сравниваем их только друг с другом.

Пусть для узлов x и y ($x \neq y$) $k = key[x]$ и $i = key[y]$. Определим индикаторную случайную величину

$$X_{i,k} = \mathrm{I}\{y \text{ находится в правом хребте левого поддерева } x\}.$$

е) Покажите, что $X_{i,k} = 1$ тогда и только тогда, когда $priority[y] > priority[x]$, $key[y] < key[x]$ и для каждого z такого, что $key[y] < key[z] < key[x]$, имеем $priority[y] < priority[z]$.

ж) Покажите, что

$$\Pr\{X_{i,k} = 1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}.$$

з) Покажите, что

$$\mathrm{E}[C] = \sum_{j=1}^{k-1} \frac{1}{j(j+1)} = 1 - \frac{1}{k}.$$

и) Используя симметрию, покажите, что

$$\mathrm{E}[D] = 1 - \frac{1}{n-k+1}.$$

к) Сделайте вывод о том, что математическое ожидание количества поворотов, выполняемых при вставке узла в дерамиду, меньше 2.

Заключительные замечания

Идея балансировки деревьев поиска принадлежит советским математикам Г.М. Адельсону-Вельскому и Е.М. Ландису [2], предложившим в 1962 году класс сбалансированных деревьев поиска, получивших название AVL-деревьев (описанных в задаче 13-3). Еще один класс деревьев поиска, называемых 2-3-деревьями, был предложен Хопкрофтом (J.E. Hopcroft, не опубликовано) в 1970 году. Баланс этих деревьев поддерживается при помощи изменения степеней ветвления в узлах. Обобщение 2-3-деревьев, предложенное Байером (Bayer) и Мак-Крейтом (McCreight) [32] под названием B-деревьев, рассматривается в главе 18.

Красно-черные деревья предложены Байером [31] под названием "симметричные бинарные B-деревья". Гибас (Guibas) и Седжвик (Sedgewick) [135] детально изучили их свойства и предложили использовать концепцию цветов. Андерсон (Anderson) [15] предложил вариант красно-черных деревьев, обладающих повышенной простотой кодирования (который был назван Вейссом (Weiss) [311] AA-деревьями). Эти деревья подобны красно-черным деревьям с тем отличием, что левый потомок в них не может быть красным.

Дерамиды были предложены Сиделем (Siedel) и Арагоном (Aragon) [271]. Они представляют собой реализацию словаря по умолчанию в LEDA, тщательно разработанном наборе структур данных и алгоритмов.

Имеется множество других вариантов сбалансированных бинарных деревьев, включая взвешенно-сбалансированные деревья [230], деревья с k соседями [213], т.н. "деревья-козлы отпущения" (scapegoat trees) [108] и другие. Возможно, наиболее интересны "косые" деревья (splay trees), разработанные Слитором (Sleator) и Таржаном (Tarjan) [282], обладающие свойством саморегуляции (хорошее описание косых деревьев можно найти в работе Таржана [292]). Косые деревья поддерживают сбалансированность без использования дополнительных условий балансировки типа цветов. Вместо этого всякий раз при обращении над ним выполняются "косые" операции (включающие, в частности, повороты). Амортизированная стоимость (см. главу 17) таких операций в дереве с n узлами составляет $O(\lg n)$.

Альтернативой сбалансированным бинарным деревьям поиска являются списки с пропусками (skip list) [251], которые представляют собой связанные списки, оснащенные рядом дополнительных указателей. Все словарные операции в таких списках с n элементами имеют математическое ожидание времени выполнения, равное $O(\lg n)$.

ГЛАВА 14

Расширение структур данных

Зачастую на практике возникают ситуации, когда “классических” структур данных — таких как дважды связанные списки, хеш-таблицы или бинарные деревья поиска — оказывается недостаточно для решения поставленных задач. Однако только в крайне редких ситуациях приходится изобретать совершенно новые структуры данных; как правило, достаточно расширить существующую структуру путем сохранения в ней дополнительной информации, что позволяет запрограммировать необходимую для данного приложения функциональность. Однако такое расширение структур данных — далеко не всегда простая задача, в первую очередь, из-за необходимости обновления и поддержки дополнительной информации стандартными операциями над структурой данных.

В этой главе рассматриваются две структуры данных, которые построены путем расширения красно-черных деревьев. В разделе 14.1 описывается структура, которая обеспечивает операции поиска порядковых статистик в динамическом множестве. С ее помощью мы можем быстро найти i-е по порядку наименьшее число или ранг данного элемента в упорядоченном множестве. В разделе 14.2 рассматривается общая схема расширения структур данных и доказывается теорема, которая может упростить расширение красно-черных деревьев. В разделе 14.3 эта теорема используется при разработке структуры данных, поддерживающей динамическое множество промежутков, например, промежутков времени. Такая структура позволяет быстро находить в множестве промежуток, перекрывающийся с данным.

14.1 Динамические порядковые статистики

В главе 9 было введено понятие порядковой статистики. В частности, i-й порядковой статистикой множества из n элементов ($i \in \{1, 2, \ldots, n\}$) является элемент множества с i-м в порядке возрастания ключом. Мы знаем, что любая порядковая статистика может быть найдена в неупорядоченном множестве за время $O(n)$. В этом разделе вы увидите, каким образом можно изменить красно-черные деревья для того, чтобы находить порядковую статистику за время $O(\lg n)$. Вы также узнаете, каким образом можно находить ***ранг*** элемента — его порядковый номер в линейно упорядоченном множестве — за то же время $O(\lg n)$.

На рис. 14.1 показана структура данных, которая поддерживает быстрые операции порядковой статистики. ***Дерево порядковой статистики*** T (order-statistic tree) представляет собой просто красно-черное дерево с дополнительной информацией, хранящейся в каждом узле. Помимо обычных полей узлов красно-черного дерева $key[x]$, $color[x]$, $p[x]$, $left[x]$ и $right[x]$, в каждом узле дерева порядковой статистики имеется поле $size[x]$, которое содержит количество внутренних) узлов в поддереве с корневым узлом x (включая сам x), т.е. размер поддерева. Если мы определим размер ограничителя как 0, т.е. $size[nil[T]] = 0$, то получим тождество $size[x] = size[left[x]] + size[right[x]] + 1$.

В дереве порядковой статистики условие различности всех ключей не ставится. Например, на рис. 14.1 имеются два ключа со значением 14, и два — с значением 21. В случае наличия одинаковых ключей определение ранга оказывается нечетким, и мы устраняем неоднозначность дерева порядковой статистики, определяя ранг элемента как позицию, в которой будет выведен данный элемент при центрированном обходе дерева. Например, на рис. 14.1 ключ 14, хранящийся в черном узле, имеет ранг 5, а в красном — ранг 6.

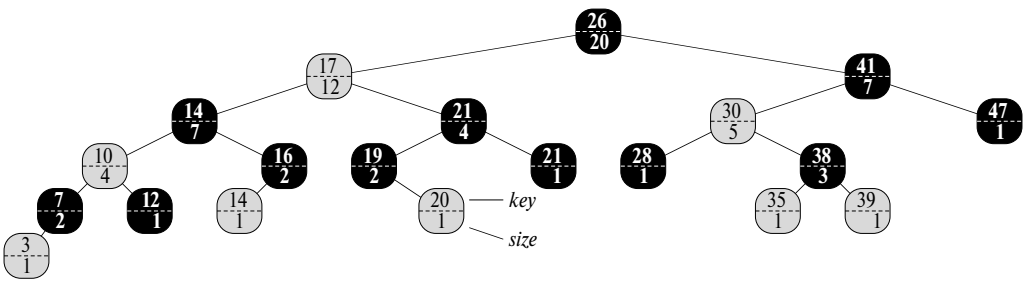


Рис. 14.1. Дерево порядковой статистики, являющееся расширением красно-черного дерева

Выборка элемента с заданным рангом

Перед тем как будет изучен вопрос об обновлении информации о размере поддеревьев в процессе вставки и удаления, давайте посмотрим на реализацию двух запросов порядковой статистики, которые используют дополнительную информацию. Начнем с операции поиска элемента с заданным рангом. Процедура OS_SELECT(x, i) возвращает указатель на узел, содержащий i-й в порядке возрастания ключ в поддереве, корнем которого является x (так что для поиска i-го в порядке возрастания ключа в дереве порядковой статистики T мы вызываем процедуру как OS_SELECT$(root\,[T]\,, i)$:

```
OS_SELECT(x, i)
1  r ← size[left[x]]+1
2  if i = r
3     then return x
4  elseif i < r
5     then return OS_SELECT(left[x], i)
6  else return OS_SELECT(right[x], i − r)
```

Идея, лежащая в основе процедуры OS_SELECT, аналогична идее, лежащей в основе алгоритмов, рассматривавшихся в главе 9. Значение $size\,[left\,[x]]$ представляет собой количество узлов, которые при центрированном обходе дерева с корнем в узле x будут выведены до узла x. Таким образом, $size\,[left\,[x]] + 1$ представляет собой ранг узла x в поддереве, корнем которого является x.

В первой строке псевдокода процедуры OS_SELECT мы вычисляем r — ранг узла x в поддереве, для которого он является корнем. Если $i = r$, то узел x является i-м в порядке возрастания элементом и мы возвращаем его в строке 3. Если же $i < r$, то i-й в порядке возрастания элемент находится в левом поддереве, так что мы рекурсивно ищем его в строке 5. Если $i > r$, то искомый элемент находится в правом поддереве и мы делаем соответствующий рекурсивный вызова в строке 6, с учетом того, что i-й в порядке возрастания в дереве с корнем x элемент является $(i - r)$-м в порядке возрастания в правом поддереве x.

Для того чтобы увидеть описанную процедуру в действии, рассмотрим поиск 17-го в порядке возрастания элемента в дереве порядковой статистики на рис. 14.1. Мы начинаем поиск с корневого узла, ключ которого равен 26, и $i = 17$. Поскольку размер левого поддерева элемента с ключом 26 равен 12, ранг самого элемента — 13. Теперь мы знаем, что элемент с рангом 17 является $17 - 13 = 4$-м в порядке возрастания элементом в правом поддереве элемента с ключом 26. После соответствующего рекурсивного вызова x становится узлом с ключом 41, а $i = 4$. Поскольку размер левого поддерева узла с ключом 41 равен 5, ранг этого узла в поддереве равен 6. Теперь мы знаем, что искомый узел находится в левом поддереве узла с ключом 41, и его номер в порядке возрастания — 4.

После очередного рекурсивного вызова x становится элементом с ключом 30, а его ранг $-$ 2, и мы рекурсивно ищем элемент с рангом $4 - 2 = 2$ в поддереве, корнем которого является узел с ключом 38. Размер его левого поддерева равен 1, так что ранг самого узла с ключом 38 равен 2, и это и есть наш искомый элемент, указатель на который и возвращает процедура.

Поскольку каждый рекурсивный вызов опускает нас на один уровень в дереве порядковой статистики, общее время работы процедуры OS_SECECT в наихудшем случае пропорционально высоте дерева. Поскольку рассматриваемое нами дерево порядковой статистики является красно-черным деревом, его высота равна $O(\lg n)$, где n — количество узлов в дереве. Следовательно, время работы процедуры OS_SECECT в динамическом множестве из n элементов равно $O(\lg n)$.

Определение ранга элемента

Процедура OS_RANK, псевдокод которой приведен далее, по заданному указателю на узел x дерева порядковой статистики T возвращает позицию данного узла при центрированном обходе дерева:

```
OS_RANK(T, x)
1  r ← size[left[x]] + 1
2  y ← x
3  while y ≠ root[T]
4        do if y = right[p[y]]
5              then r ← r + size[left[p[y]]] + 1
6           y ← p[y]
7  return r
```

Процедура работает следующим образом. Ранг x можно рассматривать как число узлов, предшествующих x при центрированном обходе дерева, плюс 1 для самого узла x. Процедура OS_RANK поддерживает следующий инвариант цикла.

> В начале каждой итерации цикла **while** в строках 3–6 r представляет собой ранг $key[x]$ в поддереве, корнем которого является узел y.

Мы воспользуемся этим инвариантом для того, чтобы показать корректность работы процедуры OS_RANK.

Инициализация. Перед первой итерацией строка 1 устанавливает r равным рангу $key[x]$ в поддереве с корнем x. Присвоение $y \leftarrow x$ в строке 2 делает инвариант истинным при первом выполнении проверки в строке 3.

Сохранение. В конце каждой итерации цикла **while** выполняется присвоение $y \leftarrow p[y]$. Таким образом, мы должны показать, что если r — ранг $key[x]$ в поддереве с корнем y в начале выполнения тела цикла, то в конце r становится рангом $key[x]$ в поддереве, корнем которого является $p[y]$. В каждой

итерации цикла **while** мы рассматриваем поддерево, корнем которого является $p[y]$. Мы уже подсчитали количество узлов в поддереве с корнем в узле y, который предшествует x при центрированном обходе дерева, так что теперь мы должны добавить узлы из поддерева, корнем которого является "брат" y (и который также предшествует x при центрированном обходе дерева), а также добавить 1 для самого $p[y]$ — если, конечно, этот узел также предшествует x. Если y — левый потомок, то ни $p[y]$, ни любой узел из поддерева правого потомка $p[y]$ не могут предшествовать x, так что r остается неизменным. В противном случае y является правым потомком, и все узлы в поддереве левого потомка $p[y]$ предшествуют x, так же как и $p[y]$. Соответственно, в строке 5 мы добавляем $size[left[p[y]]] + 1$ к текущему значению r.

Завершение. Цикл завершается, когда $y = root[T]$, так что поддерево, корнем которого является y, представляет собой все дерево целиком и, таким образом, r является рангом $key[x]$ в дереве в целом.

В качестве примера рассмотрим работу процедуры OS_RANK с деревом порядковой статистики, показанным на рис. 14.1. Если мы будем искать ранг узла с ключом 38, то получим следующую последовательность значений $key[y]$ и r в начале цикла **while**.

Итерация	$key[y]$	r
1	38	2
2	30	4
3	41	4
4	26	17

Таким образом, по окончании работы процедура возвращает значение ранга, равное 17.

Поскольку каждая итерация цикла занимает время $O(1)$, а y при каждой итерации поднимается на один уровень вверх, общее время работы процедуры OS_RANK в наихудшем случае пропорционально высоте дерева, т.е. равно $O(\lg n)$ в случае дерева порядковой статистики с n узлами.

Поддержка размера поддеревьев

При наличии поля $size$ в узлах дерева процедуры OS_SELECT и OS_RANK позволяют быстро вычислять порядковые статистики. Однако это поле оказывается совершенно бесполезным, если оно не будет корректно обновляться базовыми модифицирующими операциями над красно-черными деревьями. Давайте рассмотрим, какие изменения надо внести в алгоритмы вставки и удаления для того, чтобы они поддерживали поля размеров поддеревьев при сохранении асимптотического времени работы.

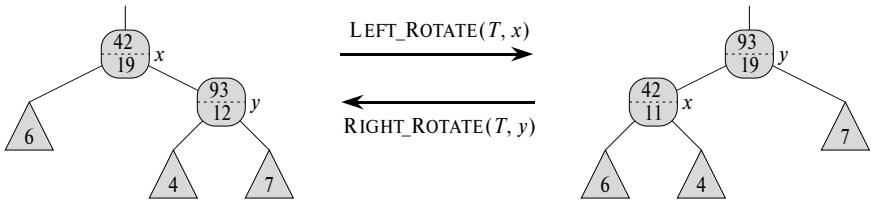


Рис. 14.2. Обновление размеров поддеревьев в процессе поворота

В разделе 13.3 мы видели, что вставка в красно-черное дерево состоит из двух фаз. Первая фаза заключается в проходе вниз по дереву и вставке нового узла в качестве дочернего для уже существующего. При второй фазе выполняется проход вверх по дереву, при котором выполняются изменения цветов узлов и повороты для сохранения красно-черных свойств дерева.

Для поддержки поля $size$ в первой фазе достаточно просто увеличить значение $size\,[x]$ для каждого узла x на пути от корня к листьям. Новый узел получает значение поля $size$, равное 1. Поскольку вдоль пути имеется $O\,(\lg n)$ листьев, дополнительное время, требующееся для поддержания поля $size$ в первой фазе, составляет $O\,(\lg n)$.

Во второй фазе структурные изменения дерева вызываются только поворотами, которых, как мы знаем, может быть не больше двух. Кроме того, поворот является локальной операцией — после его выполнения становятся некорректными значения $size$ только у двух узлов, вокруг связи между которыми выполняется поворот. Возвращаясь к коду LEFT_ROTATE(T, x) в разделе 13.2, нам надо просто добавить в него следующие строки:

```
13  size[y] ← size[x]
14  size[x] ← size[left[x]] + size[right[x]] + 1
```

На рис. 14.2 проиллюстрировано обновление полей $size$ при поворотах. Изменения в процедуре RIGHT_ROTATE симметричны только что рассмотренным.

Поскольку при вставке в красно-черное дерево выполняется не более двух поворотов, дополнительное время, требующееся для поддержки актуальности полей $size$ во второй фазе, равно $O\,(1)$. Таким образом, общее время вставки в дерево порядковой статистики с n узлами составляет $O\,(\lg n)$, т.е. остается тем же, что и в случае обычного красно-черного дерева.

Удаление узла из красно-черного дерева также состоит из двух фаз — первая удаляет узел из дерева поиска, лежащего в основе красно-черного дерева, а вторая восстанавливает красно-черные свойства, выполняя не более трех поворотов и не внося никаких других структурных изменений (см. раздел 13.4). В первой фазе из дерева извлекается узел y. Для обновления размеров поддеревьев мы просто проходим по пути от узла y до корня дерева, уменьшая величину поля $size$

каждого узла на нашем пути. Поскольку вдоль пути в красно-черном дереве с n узлами имеется $O(\lg n)$ листьев, дополнительное время, требующееся для поддержания поля $size$ в первой фазе, составляет $O(\lg n)$. Во второй фазе выполняется $O(1)$ поворотов, которые требуют соответствующего времени для поддержания актуальности размеров поддеревьев. Итак, как видим, и вставка, и удаление в состоянии поддерживать корректность значений полей $size$ в дереве, при этом время их работы в дереве порядковой статистики с n узлами составляет $O(\lg n)$.

Упражнения

14.1-1. Покажите, как работает вызов процедуры OS_SELECT$(root[T], 10)$ для дерева T, изображенного на рис. 14.1.

14.1-2. Покажите, как работает вызов процедуры OS_RANK(T, x) для дерева T, изображенного на рис. 14.1, если $key[x] = 35$.

14.1-3. Напишите нерекурсивную версию процедуры OS_SELECT.

14.1-4. Разработайте рекурсивную процедуру OS_KEY_RANK(T, k), которая получает в качестве входных параметров дерево T и ключ k и возвращает значение ранга ключа k в динамическом множестве, представленном T. Считаем, что все ключи в T различны.

14.1-5. Даны элемент дерева порядковой статистики с n узлами x и неотрицательное целое число i. Каким образом можно найти i-й в порядке возрастания элемент, начиная отсчет от x, за время $O(\lg n)$?

14.1-6. Заметим, что процедуры OS_SELECT и OS_RANK используют поле $size$ только для вычисления ранга узла в поддереве, корнем которого является данный узел. Предположим, что вместо этого в каждом узле хранится его ранг в таком поддереве. Покажите, каким образом можно поддерживать актуальность этой информации в процессе вставки и удаления (вспомните, что эти две операции могут выполнять повороты).

14.1-7. Покажите, как использовать деревья порядковой статистики для подсчета числа инверсий (см. задачу 2-4) в массиве размером n за время $O(n \lg n)$.

⋆ 14.1-8. Рассмотрим n хорд окружности, каждая из которых определяется своими конечными точками. Опишите алгоритм определения количества пар пересекающихся хорд за время $O(n \lg n)$. (Например, если все n хорд представляют собой диаметры, пересекающиеся в центре круга, то правильным ответом будет $\binom{n}{2}$). Считаем, что все конечные точки хорд различны.

14.2 Расширение структур данных

При разработке алгоритмов процесс расширения базовых структур данных для поддержки дополнительной функциональности встречается достаточно часто. В следующем разделе он будет использован для построения структур данных, которые поддерживают операции с промежутками. В данном разделе мы рассмотрим шаги, которые необходимо выполнить при таком расширении, а также докажем теорему, которая позволит нам упростить расширение красно-черных деревьев.

Расширение структур данных можно разбить на четыре шага.

1. Выбор базовой структуры данных.
2. Определение необходимой дополнительной информации, которую следует хранить в базовой структуре данных и поддерживать ее актуальность.
3. Проверка того, что дополнительная информация может поддерживаться основными модифицирующими операциями над базовой структурой данных.
4. Разработка новых операций.

Приведенные правила представляют собой общую схему, которой вы не обязаны жестко следовать. Конструирование — это искусство, зачастую опирающееся на метод проб и ошибок, и все шаги могут на практике выполняться параллельно. Так, нет особого смысла в определении дополнительной информации и разработке новых операций (шаги 2 и 4), если мы не в состоянии эффективно поддерживать работу с этой информацией. Тем не менее, описанная схема позволяет с пониманием дела направлять ваши усилия, а также помочь в организации документирования расширенной структуры данных.

Мы следовали приведенной схеме при разработке деревьев порядковой статистики в разделе 14.1. На шаге 1 в качестве базовой структуры данных мы выбрали красно-черные деревья в связи с их эффективной поддержкой других операций порядковых над динамическим множеством, таких как MINIMUM, MAXIMUM, SUCCESSOR и PREDECESSOR.

На шаге 2 мы добавили в структуру данных поле $size$, в котором каждый узел x хранит размер своего поддерева. В общем случае дополнительная информация делает операции более эффективными, что наглядно видно на примере операций OS_SELECT и OS_RANK, которые при использовании поля $size$ выполняются за время $O(\lg n)$. Зачастую дополнительная информация представляет собой не данные, а указатели, как, например, в упражнении 14.2-1.

На следующем шаге мы убедились в том, что модифицирующие операции вставки и удаления в состоянии поддерживать поле $size$ с неизменным асимптотическим временем работы $O(\lg n)$. В идеале для поддержки дополнительной информации требуется обновлять только малую часть элементов структуры данных. Например, если хранить в каждом узле его ранг в дереве, то процедуры OS_SELECT и OS_RANK будут работать быстро, но вставка нового минимального

элемента потребует при такой схеме внесения изменений в каждый узел дерева. При хранении размеров поддеревьев вставка нового элемента требует изменения информации только в $O(\lg n)$ узлах.

Последний шаг состоял в разработке операций OS_SELECT и OS_RANK. В конце концов, именно необходимость новых операций в первую очередь приводит нас к расширению структуры данных. Иногда, вместо разработки новых операций мы используем дополнительную информацию для ускорения существующих (см. упражнение 14.2-1).

Расширение красно-черных деревьев

При использовании в качестве базовой структуры данных красно-черных деревьев мы можем доказать, что определенные виды дополнительной информации могут эффективно обновляться при вставках и удалениях, делая тем самым шаг 3 очень простым. Доказательство следующей теоремы аналогично рассуждениям из раздела 14.1 о возможности поддержки поля $size$ деревьями порядковой статистики.

Теорема 14.1 (Расширение красно-черных деревьев). Пусть f — поле, которое расширяет красно-черное дерево T из n узлов, и пусть содержимое поля f узла x может быть вычислено с использованием лишь информации, хранящейся в узлах x, $left[x]$ и $right[x]$, включая $f[left[x]]$ и $f[right[x]]$. В таком случае мы можем поддерживать актуальность информации f во всех узлах дерева T в процессе вставки и удаления без влияния на асимптотическое время работы данных процедур $O(\lg n)$.

Доказательство. Основная идея доказательства заключается в том, что изменение поля f узла x воздействует на значения поля f только у предков узла x. Таким образом, изменение $f[x]$ может потребовать изменения $f[p[x]]$, но не более; изменение $f[p[x]]$ может привести только к изменению $f[p[p[x]]]$, и т.д. вверх по дереву. При обновлении $f[root[T]]$ от этого значения не зависят никакие другие, так что процесс обновлений на этом завершается. Поскольку высота красно-черного дерева равна $O(\lg n)$, изменение поля f в узле требует времени $O(\lg n)$ для обновления всех зависящих от него узлов.

Вставка узла x в T состоит из двух фаз (см. раздел 13.3). Во время первой фазы узел x вставляется в дерево в качестве дочернего узла некоторого существующего узла $p[x]$. Значение $f[x]$ можно вычислить за время $O(1)$, поскольку, в соответствии с условием теоремы, оно зависит только от информации в других полях x и информации в дочерних по отношению к x узлах; однако дочерними узлами x являются ограничители $nil[T]$. После вычисления $f[x]$ изменения распространяются вверх по дереву. Таким образом, общее время выполнения первой фазы вставки равно $O(\lg n)$. Во время второй фазы единственным преобразованием, способным вызвать структурные изменения дерева, являются повороты.

Поскольку при повороте изменения затрагивают только два узла, общее время, необходимое для обновления полей f, — $O(\lg n)$ на один поворот. Поскольку при вставке выполняется не более двух поворотов, общее время работы процедуры вставки — $O(\lg n)$.

Так же, как и вставка, удаление выполняется в две стадии (см. раздел 13.4). Сначала выполняются изменения в дереве, при которых удаляемый узел замещается следующим за ним, а затем из дерева извлекается удаляемая вершина (или следующая за ней). Обновления значений f занимают время $O(\lg n)$, поскольку вносимые изменения носят локальный характер. На втором этапе при восстановлении красно-черных свойств выполняется не более трех поворотов, каждый из которых требует для обновления всех полей f на пути к корню время, не превышающее $O(\lg n)$. Таким образом, общее время удаления, как и вставки, составляет $O(\lg n)$. ■

Во многих случаях (в частности, в случае поля $size$ в деревьях порядковой статистики) время, необходимое для обновления полей после вращения, составляет $O(1)$, а не $O(\lg n)$, приведенное в доказательстве теоремы 14.1. Пример такого поведения можно также найти в упражнении 14.2-4.

Упражнения

14.2-1. Покажите, каким образом расширить дерево порядковой статистики, чтобы операции MAXIMUM, MINIMUM, SUCCESSOR и PREDECESSOR выполнялись за время $O(1)$ в наихудшем случае. Асимптотическая производительность остальных операций над деревом порядковой статистики должна при этом остаться неизменной. (*Указание*: добавьте к узлам указатели.)

14.2-2. Может ли черная высота узлов в красно-черном дереве поддерживаться как поле узла дерева без изменения асимптотической производительности операций над красно-черными деревьями? Покажите, как этого достичь (или докажите, что это невозможно).

14.2-3. Может ли глубина узлов в красно-черном дереве эффективно поддерживаться в виде поля узла? Покажите, как этого достичь (или докажите, что это невозможно).

⋆ 14.2-4. Пусть $\otimes$ — ассоциативный бинарный оператор, а a — поле, поддерживаемое в каждом узле красно-черного дерева. Предположим, что мы хотим включить в каждый узел x дополнительное поле f, такое что $f[x] = a[x_1] \otimes a[x_2] \otimes \cdots \otimes a[x_m]$, где $x_1, x_2, \ldots, x_m$ — центрированный список узлов поддерева, корнем которого является x. Покажите, что после поворота все поля f могут быть корректно обновлены за время $O(1)$. Проведите подобное рассуждение для поля $size$.

⋆ 14.2-5. Мы хотим добавить к красно-черным деревьям операцию RB_ENUMERATE(x, a, b), которая выводит все ключи $a \leqslant k \leqslant b$ в дереве, корнем которого является x. Опишите, как можно реализовать процедуру RB_ENUMERATE, чтобы время ее работы составляло $\Theta(m + \lg n)$, где n — количество внутренних узлов в дереве, а m — число выводимых ключей. (*Указание*: добавлять новые поля в красно-черное дерево нет необходимости.)

14.3 Деревья отрезков

В этом разделе мы расширим красно-черные деревья для поддержки операций над динамическими множествами промежутков. ***Отрезком*** называется упорядоченная пара действительных чисел $[t_1, t_2]$, таких что $t_1 \leqslant t_2$. Отрезок $[t_1, t_2]$ представляет множество $\{t \in \mathbf{R} : t_1 \leqslant t \leqslant t_2\}$. ***Интервал*** (t_1, t_2) представляет собой отрезок без конечных точек, т.е. множество $\{t \in \mathbf{R} : t_1 < t < t_2\}$, а ***полуинтервалы*** $[t_1, t_2)$ и $(t_1, t_2]$ образуются из отрезка при удалении из него одной из конечных точек. В случае, когда принадлежность концов несущественна, обычно говорят о ***промежутках***. В данном разделе мы будем работать с отрезками, но расширение результатов на интервалы и полуинтервалы не должно составить для читателя никакого труда.

Отрезки удобны для представления событий, которые занимают некоторый промежуток времени. Мы можем, например, сделать запрос к базе данных о том, какие события происходили в некоторый промежуток времени. Рассматриваемая в данном разделе структура данных обеспечивает эффективное средство для поддержки такой базы данных, работающей с промежутками.

Мы можем представить отрезок $[t_1, t_2]$ в виде объекта i с полями $low[i] = t_1$ (левый, или нижний, конец отрезка) и $high[i] = t_2$ (правый, или верхний, конец). Мы говорим, что отрезки i и i' ***перекрываются*** (overlap), если $i \cap i' \neq \emptyset$, т.е. если $low[i] \leqslant high[i']$ и $low[i'] \leqslant high[i]$. Для любых двух отрезков i и i' выполняется только одно из трех свойств (трихотомия отрезков):

а) i и i' перекрываются;

б) i находится слева от i' (т.е. $high[i] < low[i']$);

в) i находится справа от i' (т.е. $high[i'] < low[i]$).

Эти варианты взаимного расположения отрезков проиллюстрированы на рис. 14.3.

Дерево отрезков представляет собой красно-черное дерево, каждый элемент которого содержит отрезок $int[x]$. Деревья отрезков поддерживают следующие операции.

INTERVAL_INSERT(T, x), которая добавляет в дерево отрезков T элемент x, поле int которого содержит некоторый отрезок.

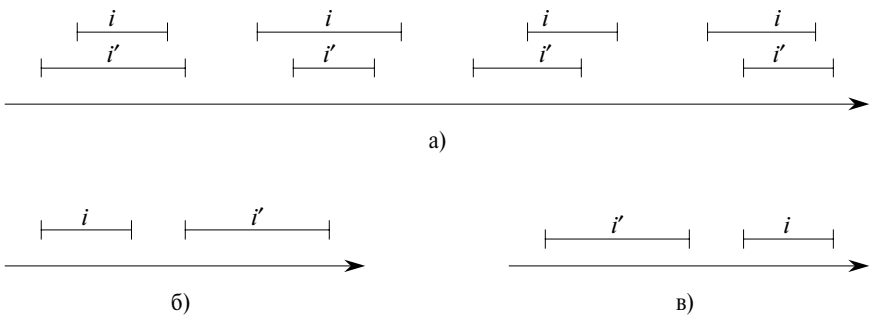


Рис. 14.3. Варианты взаиморасположения отрезков i и i'

INTERVAL_DELETE(T, x), которая удаляет элемент x из дерева отрезков T.

INTERVAL_SEARCH(T, i), которая возвращает указатель на элемент x из дерева отрезков T, такой что $int\,[x]$ перекрывается с отрезком i (либо ограничитель $nil\,[T]$, если такого элемента в дереве нет).

На рис. 14.4 показано множество отрезков и его представление в виде дерева отрезков. Давайте рассмотрим этапы расширения структур данных из раздела 14.2 при решении задачи разработки дерева отрезков и операций над ним.

Шаг 1. Выбор базовой структуры данных.

В качестве базовой структуры данных мы выбираем красно-черное дерево, каждый узел x которого содержит отрезок $int\,[x]$, а ключом узла является левый конец отрезка $low\,[int[x]]$. Таким образом, центрированный обход дерева приводит к перечислению отрезков в порядке сортировки по их левым концам.

Шаг 2. Дополнительная информация.

В дополнение к самим отрезкам, каждый узел x содержит значение $max\,[x]$, которое представляет собой максимальное значение всех конечных точек отрезков, хранящихся в поддереве, корнем которого является x.

Шаг 3. Поддержка информации.

Мы должны убедиться, что вставка и удаление в дереве с n узлами могут быть выполнены за время $O(\lg n)$. Определить значение поля max в узле x можно очень просто с использованием полей max дочерних узлов:

$$max\,[x] = \max\left(high\,[int\,[x]]\,, max\,[left\,[x]]\,, max\,[right\,[x]]\right).$$

Таким образом, по теореме 14.1, вставка в дерево отрезков и удаление из него может быть выполнена за время $O(\lg n)$. В действительности, как показано в упражнениях 14.2-4 и 14.3-1, обновление поля max после вращения может быть выполнено за время $O(1)$.

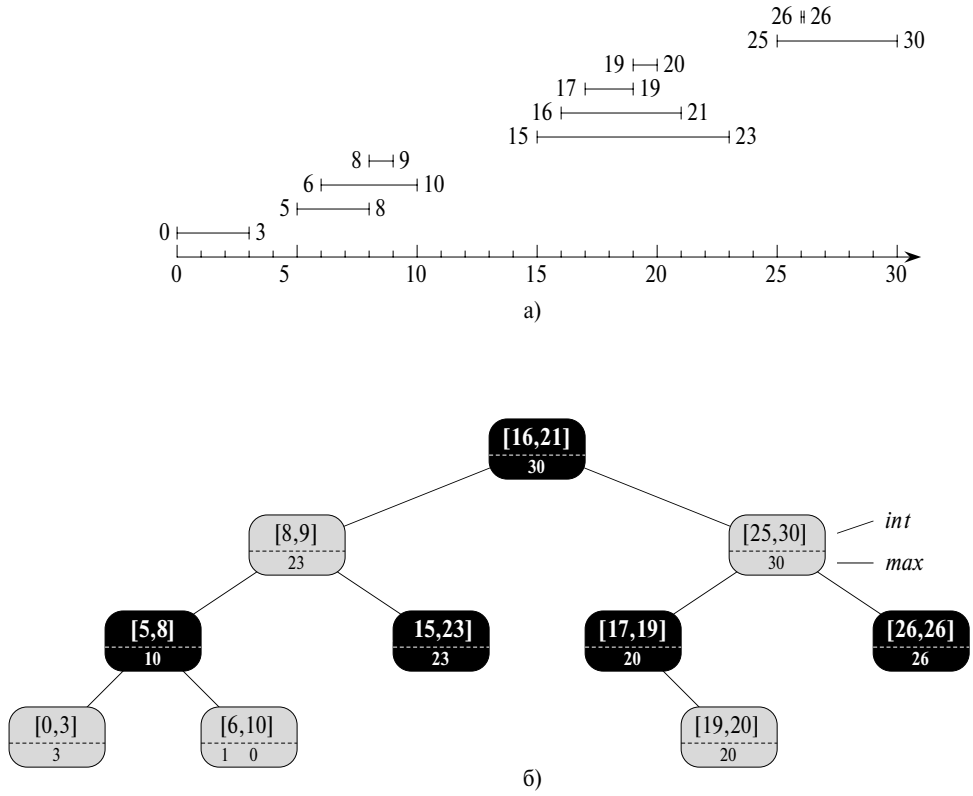


Рис. 14.4. Множество отрезков и его представление в виде дерева отрезков

Шаг 4. Разработка новых операций.

Единственная новая операция, которую мы хотим разработать, — это INTERVAL_SEARCH(T, i), которая осуществляет поиск отрезка в дереве T, который перекрывается с данным. Если такого отрезка в дереве нет, процедура возвращает указатель на ограничитель $nil\,[T]$:

INTERVAL_SEARCH(T, i)
1 $x \leftarrow root[T]$
2 **while** $x \neq nil[T]$ и i не перекрывается с $int[x]$
3 **do if** $left[x] \neq nil[T]$ и $max[left[x]] \geq low[i]$
4 **then** $x \leftarrow left[x]$
5 **else** $x \leftarrow right[x]$
6 **return** x

Для поиска отрезка, который перекрывается с i, мы начинаем с присвоения указателю x корня дерева и выполняем спуск по дереву. Спуск завершается когда мы находим перекрывающийся отрезок или когда x указывает на ограничитель

$nil\,[T]$. Поскольку каждая итерация основного цикла выполняется за время $O\,(1)$, а высота красно-черного дерева с n узлами равна $O\,(\lg n)$, время работы процедуры INTERVAL_SEARCH равно $O\,(\lg n)$.

Перед тем как мы убедимся в корректности процедуры INTERVAL_SEARCH, давайте посмотрим, как она работает, на примере дерева, показанного на рис. 14.4. Предположим, что мы хотим найти отрезок, перекрывающийся с отрезком $i = [22, 25]$. Мы начинаем работу с корня, в котором содержится отрезок $[16, 21]$, который не перекрывается с отрезком i. Поскольку значение $max\,[left\,[x]] = 23$ превышает $low\,[i] = 22$, цикл продолжает выполнение с x, указывающим на левый дочерний узел корня. Этот узел содержит отрезок $[8, 9]$, который также не перекрывается с отрезком i. Теперь $max\,[left\,[x]] = 10$ меньше $low\,[i] = 22$, так что мы переходим к правому дочернему узлу. В нем содержится отрезок $[15, 23]$, перекрывающийся с x, так что процедура возвращает указатель на данный узел.

В качестве примера неудачного поиска попробуем найти в том же дереве отрезок, перекрывающийся с отрезком $i = [11, 14]$. Мы вновь начинаем с корня. Поскольку отрезок в корне $[16, 21]$ не перекрывается с i, и поскольку $max\,[left\,[x]] = 23$ больше, чем $low\,[i] = 11$, мы переходим к левому дочернему узлу корня с отрезком $[8, 9]$. Отрезок $[8, 9]$ также не перекрывается с i, а $max\,[left\,[x]] = 10$, что меньше, чем $low\,[i] = 11$, поэтому мы переходим вправо (обратите внимание, что теперь в левом поддереве нет ни одного отрезка, перекрывающегося с i. Отрезок $[15, 23]$ не перекрывается с i, его левый дочерний узел — $nil\,[T]$, так что цикл завершается и процедура возвращает указатель на ограничитель $nil\,[T]$.

Для того чтобы убедиться в корректности процедуры INTERVAL_SEARCH, нам надо разобраться, почему для поиска нам достаточно пройти по дереву всего лишь по одному пути. Основная идея заключается в том, что в любом узле x, если $int\,[x]$ не перекрывается с i, то дальнейший поиск всегда идет в правильном направлении, т.е. перекрывающийся отрезок, если таковой имеется в дереве, гарантированно будет обнаружен в исследуемой части дерева. Более точно это свойство сформулировано в следующей теореме.

Теорема 14.2. Процедура INTERVAL_SEARCH(T, i) либо возвращает узел, отрезок которого перекрывается с отрезком i, либо, если в дереве T не содержится отрезка, перекрывающегося с i, возвращает $nil\,[T]$.

Доказательство. Цикл **while** в строках 2–5 завершается, если $x = nil\,[T]$ либо если i перекрывается с $int\,[x]$. В последнем случае процедура тривиально возвращает значение x. Поэтому нас интересует первый случай, когда цикл завершается из-за того, что $x = nil\,[T]$.

Воспользуемся следующим инвариантом цикла **while** в строках 2–5.

> Если дерево T содержит отрезок, который перекрывается с i, то этот отрезок находится в поддереве, корнем которого является узел x.

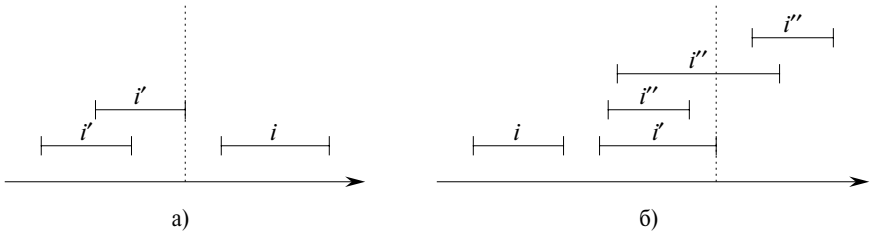


Рис. 14.5. Отрезки в доказательстве теоремы 14.2. Пунктирной линией показана величина $max\,[left\,[x]]$

Инициализация. Перед выполнением первой итерации в строке 1 переменной x присваивается указатель на корень дерева, так что инвариант выполняется.

Сохранение. При каждой итерации цикла **while** выполняется либо строка 4, либо строка 5. Покажем, что инвариант цикла сохраняется в любом случае.

При выполнении строки 5 (в силу невыполнения условия в строке 3) мы имеем $left\,[x] = nil\,[T]$ или $max\,[left[x]] < low\,[i]$. Если $left\,[x] = nil\,[T]$, то поддерево, корнем которого является $left\,[x]$, очевидно, не может содержать отрезок, перекрывающийся с i, так что присвоение $x \leftarrow right\,[x]$ сохраняет инвариант цикла. Предположим, следовательно, что $left\,[x] \neq nil\,[T]$ и $max\,[left[x]] < low\,[i]$. Как показано на рис. 14.5*а*, для каждого интервала i' из левого поддерева x выполняется следующее соотношение:

$$high\,[i'] \leqslant max\,[left\,[x]] < low\,[i]\,.$$

В соответствии со свойством трихотомии отрезков, i' и i не перекрываются. Таким образом, левое поддерево x не содержит отрезков, перекрывающихся с i, и присвоение $x \leftarrow right\,[x]$ сохраняет инвариант.

Пусть выполняется строка 4. В этом случае мы покажем, что если в левом поддереве x нет отрезка, перекрывающегося с i, то его вообще нет в дереве. Строка 4 выполняется в случае, когда $max\,[left\,[x]] \geqslant low\,[i]$. Кроме того, по определению поля max в левом поддереве x должен быть некоторый интервал i', такой что $high\,[i'] = max\,[left\,[x]] \geqslant low\,[i]$. (Данная ситуация проиллюстрирована на рис. 14.5*б*.)

Поскольку i и i' не перекрываются и поскольку неверно, что $high\,[i'] < low\,[i]$, отсюда в соответствии со свойством трихотомии отрезков следует, что $high\,[i] < low\,[i']$. Поскольку дерево отрезков упорядочено в соответствии с левыми концами отрезков, из свойства дерева поиска вытекает, что для любого отрезка i'' из правого поддерева x

$$high\,[i] < low\,[i'] \leqslant low\,[i'']\,.$$

Из трихотомии отрезков следует, что i и i'' не перекрываются. Мы можем, таким образом, заключить, что независимо от того, имеется ли в левом поддереве x отрезок, перекрывающийся с i, присвоение $x \leftarrow left\,[x]$ сохраняет инвариант цикла.

Завершение. Если цикл завершается по условию $x = nil\,[T]$, то в дереве, корнем которого является x, нет отрезков, перекрывающихся с i. Обращение инварианта цикла приводит к заключению, что дерево T не содержит отрезков, перекрывающихся с i. Следовательно, возвращаемое процедурой значение $nil\,[T]$ совершенно корректно. ■

Итак, процедура INTERVAL_SEARCH работает корректно.

Упражнения

14.3-1. Напишите псевдокод процедуры LEFT_ROTATE, которая работает с узлами дерева отрезков и обновляет поля max за время $O\,(1)$.

14.3-2. Перепишите код процедуры INTERVAL_SEARCH для корректной работы с интервалами (отрезками без конечных точек).

14.3-3. Разработайте эффективно работающий алгоритм, который для данного отрезка i возвращает отрезок, перекрывающийся с i и имеющий минимальное значение левого конца (либо $nil\,[T]$, если такого отрезка не существует).

14.3-4. Пусть у нас имеется дерево отрезков T и отрезок i. Опишите, каким образом найти все отрезки в дереве T, перекрывающиеся с отрезком i, за время $O\,(\min\,(n, k \lg n))$, где k — количество найденных отрезков. (*Дополнительное условие*: попробуйте найти решение, не изменяющее дерево.)

14.3-5. Какие изменения надо внести в процедуры дерева отрезков для поддержки новой операции INTERVAL_SEARCH_EXACTLY(T, i), которая возвращает указатель на узел x дерева отрезков T, такой что $low\,[int\,[x]] = low\,[i]$ и $high\,[int\,[x]] = high\,[i]$ (либо $nil\,[T]$, если такого узла в дереве T нет). Все операции, включая INTERVAL_SEARCH_EXACTLY, должны выполняться в дереве с n узлами за время $O\,(\lg n)$.

14.3-6. Пусть у нас есть динамическое множество натуральных чисел Q, на котором определена операция MIN_GAP, возвращающая минимальное расстояние между соседними числами в Q. Например, если $Q = \{1, 5, 9, 15, 18, 22\}$, то MIN_GAP$(Q)$ возвратит значение $18 - 15 = 3$, так как 15 и 18 — ближайшие соседние числа в Q. Разработайте максимально эффективные процедуры INSERT, DELETE, SEARCH и MIN_GAP и проанализируйте их время работы.

⋆ 14.3-7. Базы данных при разработке СБИС зачастую представляют интегральную схему как список прямоугольников. Предположим, что все прямоугольники ориентированы вдоль осей x и y, так что представление прямоугольника состоит из минимальных и максимальных координат x и y. Разработайте алгоритм для выяснения, имеются ли в данном множестве из n прямоугольников два перекрывающихся (искать все перекрывающиеся пары не надо). Перекрытием считается также ситуация, когда один прямоугольник лежит полностью внутри другого, пусть при этом их границы и не пересекаются. Время работы алгоритма должно составлять $O(n \lg n)$. (*Указание*: перемещайте "строку развертки" по множеству прямоугольников.)

Задачи

14-1. **Точка максимального перекрытия**

Предположим, что мы хотим найти ***точку максимального перекрытия*** множества отрезков, т.е. точку, в которой перекрывается наибольшее количество отрезков множества.

а) Покажите, что такая точка всегда имеется и представляет собой конечную точку одного из отрезков.

б) Разработайте структуру данных, которая поддерживает эффективную работу операций INTERVAL_INSERT, INTERVAL_DELETE и FIND_POM (которая возвращает точку максимального перекрытия). (*Указание*: воспользуйтесь красно-черным деревом всех конечных точек отрезков. С каждым левым концом отрезка связано значение $+1$, с правым — значение -1. Добавьте к узлам дерева некоторую дополнительную информацию для поддержки точки максимального перекрытия.)

14-2. **Перестановка Иосифа**

Задача Иосифа[1] формулируется следующим образом. Предположим, что n человек расставлены в круг и задано некоторое натуральное число $m \leqslant n$. Начиная с определенного человека, мы идем по кругу, удаляя каждого m-го человека. После удаления человека счет продолжается дальше. Процесс продолжается, пока все n человек не будут удалены. Порядок, в котором люди удаляются из круга, определяет (n, m)-***перестановку***

[1] Достаточно подробно об этой задаче, ее происхождении и вариациях можно прочесть, например, в книге У. Болл, Г. Коксетер. *Математические эссе и развлечения*. — М.:Мир, 1986 (стр. 43–47). — *Прим. ред.*

Иосифа целых чисел 1, 2, ..., n. Например, $(7, 3)$-перестановка Иосифа имеет вид $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.

а) Пусть m — некоторая фиксированная константа. Разработайте алгоритм, который для данного n за время $O(n)$ выводит (n, m)-перестановку Иосифа.

б) Пусть m не является константой. Разработайте алгоритм, который для данных n и m за время $O(n \lg n)$ выводит (n, m)-перестановку Иосифа.

Заключительные замечания

В книге [247] Препарата (Preparata) и Шамос (Shamos) описывают ряд деревьев отрезков, встречавшихся литературе, и приводят результаты из работ Эдельсбруннера (H. Edelsbrunner, 1980) и Мак-Крейта (McCreight, 1981). В книге приведено дерево отрезков, в котором для данной статической базы данных из n отрезков все k отрезков, перекрывающихся с заданным, могут быть найдены за время $O(k + \lg n)$.

ЧАСТЬ IV

Усовершенствованные методы разработки и анализа

Введение

В этой части описаны три важных метода разработки и анализа эффективных алгоритмов: динамическое программирование (глава 15), жадные алгоритмы (глава 16) и амортизационный анализ (глава 17). В предыдущих частях были представлены другие широко распространенные методы, такие как метод разбиения, рандомизация и решение рекуррентных соотношений. Новые подходы, с которыми нам предстоит ознакомиться в этой части, более сложные, однако они полезны для решения многих вычислительных задач. К темам, рассмотренным в данной части, мы еще обратимся в последующих частях.

Динамическое программирование обычно находит применение в задачах оптимизации, в которых для получения оптимального решения необходимо сделать определенное множество выборов. После того как выбор сделан, часто возникают вспомогательные подзадачи, по виду напоминающие основную. Динамическое программирование эффективно тогда, когда определенная вспомогательная подзадача может возникнуть в результате нескольких вариантов выборов. Основной метод решения таких задач заключается в сохранении решения каждой подзадачи, которая может возникнуть повторно. В главе 15 показано, как благодаря этой простой идее алгоритм, время решения которого экспоненциально зависит от объема входных данных, иногда можно преобразовать в алгоритм с полиномиальным временем работы.

Жадные алгоритмы, как и алгоритмы, применяющиеся в динамическом программировании, используются в задачах оптимизации, для рационального решения которых требуется сделать ряд выборов. Идея, лежащая в основе жадного алгоритма, заключается в том, чтобы каждый выбор был локально оптимальным. В качестве простого примера приведем задачу о выдаче сдачи: чтобы свести к минимуму количество монет, необходимых для выдачи определенной суммы, достаточно каждый раз выбирать монету наибольшего достоинства, не превышающую той суммы, которую осталось выдать. Можно сформулировать много таких задач, оптимальное решение которых с помощью жадных алгоритмов получается намного быстрее, чем с помощью методов динамического программирования. Однако не всегда просто выяснить, окажется ли эффективным жадный алгоритмы. В главе 16 приводится обзор теории матроида, которая часто оказывается полезной для принятия подобных решений.

Амортизационный анализ — это средство анализа алгоритмов, в которых выполняется последовательность однотипных операций. Вместо того чтобы накладывать границы на время выполнения каждой операции, с помощью амортизационного анализа оценивается длительность работы всей последовательности в целом. Одна из причин эффективности этой идеи заключается в том, что в некоторых последовательностях операций невозможна ситуация, когда время работы всех индивидуальных операций является наихудшим. Зачастую некоторые операции

в таких последовательностях оказываются дорогостоящими в плане времени работы, в то время как многие другие — дешевыми. Заметим, что амортизационный анализ — это не просто средство анализа. Его можно рассматривать и как метод разработки алгоритмов, поскольку разработка и анализ времени работы алгоритмов часто тесно переплетаются. В главе 17 излагаются основы трех способов амортизационного анализа алгоритмов.

ГЛАВА 15

Динамическое программирование

Динамическое программирование, как и метод разбиения, позволяет решать задачи, комбинируя решения вспомогательных задач. (Термин “программирование” в данном контексте означает табличный метод, а не составление компьютерного кода.) В главе 2 уже было показано, как в алгоритмах разбиения задача делится на несколько независимых подзадач, каждая из которых решается рекурсивно, после чего из решений вспомогательных задач формируется решение исходной задачи. Динамическое программирование, напротив, находит применение тогда, когда вспомогательные задачи не являются независимыми, т.е. когда разные вспомогательные задачи используют решения одних и тех же подзадач. В этом смысле алгоритм разбиения, многократно решая задачи одних и тех же типов, выполняет больше действий, чем было бы необходимо. В алгоритме динамического программирования каждая вспомогательная задача решается только один раз, после чего ответ сохраняется в таблице. Это позволяет избежать одних и тех же повторных вычислений каждый раз, когда встречается данная подзадача.

Динамическое программирование, как правило, применяется к ***задачам оптимизации*** (optimization problems). В таких задачах возможно наличие многих решений. Каждому варианту решения можно сопоставить какое-то значение, и нам нужно найти среди них решение с оптимальным (минимальным или максимальным) значением. Назовем такое решение *одним из возможных* оптимальных решений. В силу того, что таких решений с оптимальным значением может быть несколько, следует отличать их от *единственного* оптимального решения.

Процесс разработки алгоритмов динамического программирования можно разбить на четыре перечисленных ниже этапа.

1. Описание структуры оптимального решения.

2. Рекурсивное определение значения, соответствующего оптимальному решению.
3. Вычисление значения, соответствующего оптимальному решению, с помощью метода восходящего анализа.
4. Составление оптимального решения на основе информации, полученной на предыдущих этапах.

Этапы 1–3 составляют основу метода динамического программирования. Этап 4 может быть опущен, если требуется узнать только значение, соответствующее оптимальному решению. На четвертом этапе иногда используется дополнительная информация, полученная на третьем этапе, что облегчает процесс конструирования оптимального решения.

В последующих разделах метод динамического программирования используется для решения некоторых задач оптимизации. В разделе 15.1 исследуется задача по составлению графика работы двух автосборочных конвейеров. По завершении каждого этапа сборки автомобиль либо остается на том же конвейере, либо перемещается на другой. В разделе 15.2 исследуется вопрос о том, в каком порядке следует выполнять перемножение нескольких матриц, чтобы свести к минимуму общее количество операций перемножения скаляров. На этих двух примерах в разделе 15.3 обсуждаются две основные характеристики, которыми должны обладать задачи, для которых подходит метод динамического программирования. Далее, в разделе 15.4 показано, как найти самую длинную общую подпоследовательность двух последовательностей. Наконец, в разделе 15.5 с помощью динамического программирования конструируется дерево бинарного поиска, оптимальное для заданного распределения ключей, по которым ведется поиск.

15.1 Расписание работы конвейера

В качестве первого примера динамического программирования рассмотрим решение производственной задачи. Некая автомобильная корпорация производит автомобили на заводе, оснащенном двумя конвейерами (рис. 15.1). На оба конвейера, на каждом из которых предусмотрено по несколько рабочих мест, поступают для сборки автомобильные шасси, после чего на каждом рабочем месте конвейера добавляются все новые детали. Наконец, собранный автомобиль покидает конвейер. На каждом конвейере имеется n рабочих мест, пронумерованных от 1 до n. Обозначим рабочее место под номером j на i-м конвейере (где i принимает значения 1 или 2) через $S_{i,j}$. На обоих конвейерах на рабочих местах с одинаковыми номерами выполняются один и те же операции. Однако конвейеры создавались в разное время и по разным технологиям, поэтому время выполнения одних и тех же операций на разных конвейерах различается. Обозначим время сборки на рабочем месте $S_{i,j}$ через $a_{i,j}$. Как видно из рис. 15.1, шасси поступает на первое

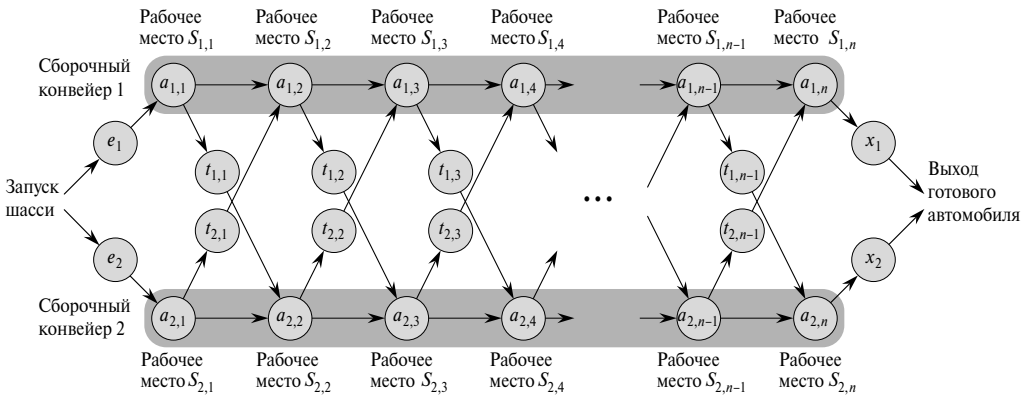
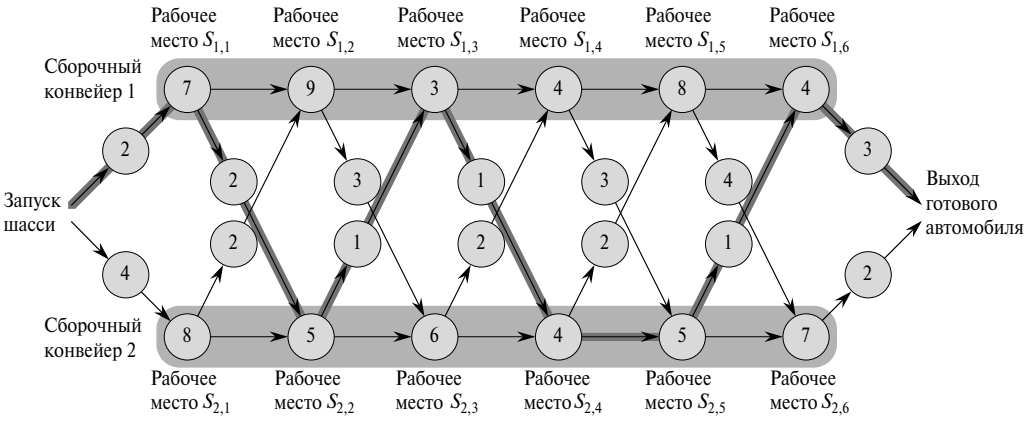


Рис. 15.1. Производственная задача по определению оптимального способа сборки

рабочее место одного из конвейеров, а затем переходит от одного рабочего места к другому. Постановка шасси на конвейер занимает время e_i, а снятие готового автомобиля с конвейера — время x_i.

Обычно поступившее на конвейер шасси проходит все этапы сборки на одном и том же конвейере. Временем перехода от одного рабочего места к другому, если этот переход выполняется в пределах одного конвейера, можно пренебречь. Однако иногда заказчик требует, чтобы автомобиль был собран за кратчайшее время, и такой порядок приходится нарушить. Для ускорения сборки шасси по-прежнему проходит все n рабочих мест в обычном порядке, однако менеджер может дать указание перебросить частично собранный автомобиль с одного конвейера на другой, причем такая переброска возможна на любом этапе сборки. Время, которое требуется для перемещения шасси с конвейера i на соседний после прохождения рабочего места $S_{i,j}$, равно $t_{i,j}$, $i = 1, 2, \ldots, n-1$ (поскольку после прохождения n-го рабочего места сборка завершается). Задача заключается в том, чтобы определить, какие рабочие места должны быть выбраны на первом конвейере, а какие — на втором, чтобы минимизировать полное время, затраченное на заводе на сборку одного автомобиля. В примере, проиллюстрированном на рис. 15.2 *а*, полное время получается наименьшим, если на первом конвейере выбираются рабочие места 1, 3 и 6, а на втором — места 2, 4 и 5. На рисунке указаны величины e_i, $a_{i,j}$, $t_{i,j}$ и x_i. Самый быстрый путь через фабрику выделен жирной линией. В части *б* рисунка приведены величины $f_i[j]$, f^*, $l_i[j]$ и l^*, соответствующие примеру, изображенному в части *а* (о том, что означают эти величины, будет сказано немного позже).

Очевидный метод перебора, позволяющий минимизировать время сборки на конвейерах с малым числом рабочих мест, становится неприменимым для большого количества рабочих мест. Если заданы списки рабочих мест, которые исполь-



а)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

б)

Рис. 15.2. Пример задачи на составление расписания сборочного конвейера

зуются на первом и втором конвейерах, то время полного прохождения шасси по конвейеру легко вычислить за время $\Theta(n)$. К сожалению, всего существует 2^n возможных комбинаций. Это легко понять, рассматривая множество рабочих мест первого конвейера как подмножество множества $\{1, 2, \ldots, n\}$; очевидно, что всего существует 2^n таких подмножеств. Таким образом, способ, при котором наиболее рациональное прохождение по конвейеру определяется методом перебора всех возможных вариантов, в результате чего определяется время сборки для каждого варианта, занял бы время, равное $\Omega(2^n)$. Очевидно, что при больших n этот способ неприемлем.

Первый этап: структура самой быстрой сборки

В парадигме динамического программирования первый этап заключается в том, чтобы охарактеризовать структуру оптимального решения. В задаче по составлению расписания работы конвейера этот шаг можно выполнить следующим образом. Рассмотрим способ, при котором шасси, поступившее на первый конвейер, попадет на рабочее место $S_{1,j}$ за кратчайшее время. Если $j = 1$, существует всего один способ прохождения такого отрезка пути, поэтому легко определить время достижения рабочего места $S_{1,j}$. Однако если $j = 2, 3, \ldots, n$, возможен один из двух вариантов: на рабочее место $S_{1,j}$ шасси могло попасть непосредственно с рабочего места $S_{1,j-1}$ или с $S_{2,j-1}$. В первом случае временем перехода

от одного рабочего места к другому можно пренебречь, а во втором переход займет время $t_{2,j-1}$. Рассмотрим обе эти возможности отдельно, хотя впоследствии станет ясно, что у них много общего.

Сначала предположим, что самый быстрый путь, проходящий через рабочее место $S_{1,j}$, проходит также через рабочее место $S_{1,j-1}$. Основной вывод, который можно сделать в этой ситуации, заключается в том, что отрезок пути от начальной точки до $S_{1,j-1}$ тоже должен быть самым быстрым. Почему? Если бы на это рабочее место можно было бы попасть быстрее, то при подстановке данного отрезка в полный путь получился бы более быстрый путь к рабочему месту $S_{1,j}$, а это противоречит начальному предположению.

Теперь предположим, что самый быстрый путь, проходящий через рабочее место $S_{1,j}$, проходит также через рабочее место $S_{2,j-1}$. В этом случае можно прийти к выводу, что на рабочее место $S_{2,j-1}$ шасси должно попасть за кратчайшее время. Объяснение аналогично предыдущему: если бы существовал более быстрый способ добраться до рабочего места $S_{2,j-1}$, то при подстановке данного отрезка в полный путь получился бы более быстрый путь к рабочему месту $S_{1,j}$, а это снова противоречит сделанному предположению.

Обобщая эти рассуждения, можно сказать, что задача по составлению оптимального расписания (определение самого быстрого пути до рабочего места $S_{i,j}$) содержит в себе оптимальное решение подзадач (нахождение самого быстрого пути до рабочего места $S_{1,j-1}$ или $S_{2,j-1}$). Назовем это свойство ***оптимальной подструктурой*** (optimal substructure). В разделе 15.3 мы убедимся, что наличие этого свойства — один из признаков применимости динамического программирования.

С помощью свойства оптимальной подструктуры можно показать, что определение оптимального решения задачи сводится к определению оптимального решения ее подзадач. В задаче по составлению расписания работы конвейера рассуждения проводятся следующим образом. Наиболее быстрый путь, проходящий через рабочее место $S_{1,j}$, должен также проходить через рабочее место под номером $j-1$, расположенное на первом или втором конвейере. Таким образом, если самый быстрый путь проходит через рабочее место $S_{1,j}$, то справедливо одно из таких утверждений:

- этот путь проходит через рабочее место $S_{1,j-1}$, после чего шасси попадает непосредственно на рабочее место $S_{1,j}$;
- этот путь проходит через рабочее место $S_{2,j-1}$, после чего автомобиль перебрасывается из второго конвейера на первый, а затем попадает на рабочее место $S_{1,j}$.

Из соображений симметрии для самого быстрого пути, проходящего через рабочее место $S_{2,j}$, справедливо одно из следующих утверждений:

- этот путь проходит через рабочее место $S_{2,j-1}$, после чего шасси попадает непосредственно на рабочее место $S_{2,j}$;
- этот путь проходит через рабочее место $S_{1,j-1}$, после чего автомобиль перебрасывается из первого конвейера на второй, а затем попадает на рабочее место $S_{2,j}$.

Чтобы решить задачу определения наиболее быстрого пути до рабочего места j, которое находится на любом из конвейеров, нужно решить вспомогательную задачу определения самого быстрого пути до рабочего места $j-1$ (также на любом из конвейеров).

Таким образом, оптимальное решение задачи об оптимальном расписании работы конвейера можно найти путем поиска оптимальных решений подзадач.

Второй этап: рекурсивное решение

Второй этап в парадигме динамического программирования заключается в том, чтобы рекурсивно определить оптимальное решение в терминах оптимальных решений подзадач. В задаче по составлению расписания работы конвейера роль подзадач играют задачи по определению самого быстрого пути, проходящего через j-е рабочее место на любом из двух конвейеров для $j = 2, 3, \ldots, n$. Обозначим через $f_i[j]$ минимально возможное время, в течение которого шасси проходит от стартовой точки до рабочего места $S_{i,j}$.

Конечная цель заключается в том, чтобы определить кратчайшее время f^*, в течение которого шасси проходит по всему конвейеру. Для этого автомобиль, который находится в сборке, должен пройти весь путь до рабочего места n, находящегося на первом или втором конвейере, после чего он покидает фабрику. Поскольку самый быстрый из этих путей является самым быстрым путем прохождения всего конвейера, справедливо следующее соотношение:

$$f^* = \min\left(f_1[n] + x_1, f_2[n] + x_2\right). \tag{15.1}$$

Легко также сделать заключение по поводу величин $f_1[1]$ и $f_2[1]$. Чтобы за кратчайшее время пройти первое рабочее место на любом из конвейеров, шасси должно попасть на это рабочее место непосредственно. Таким образом, можно записать уравнения

$$f_1[1] = e_1 + a_{1,1}, \tag{15.2}$$

$$f_2[1] = e_2 + a_{2,1}. \tag{15.3}$$

А теперь рассмотрим, как вычислить величину $f_i[j]$ при $j = 2, 3, \ldots, n$ (и $i = 1, 2$). Рассуждая по поводу $f_1[j]$, мы пришли к выводу, что самый быстрый путь до рабочего места $S_{1,j}$ является также либо самым быстрым путем до рабочего

места $S_{1,j-1}$, после чего шасси попадает прямо на рабочее место $S_{1,j}$, либо самым быстрым путем до рабочего места $S_{2,j-1}$, с последующим переходом со второго конвейера на первый. В первом случае можно записать соотношение $f_1[j] = f_1[j-1] + a_{1,j}$, а во втором — соотношение $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$. Таким образом, при $j = 2, 3, \ldots, n$ выполняется уравнение

$$f_1[j] = \min\left(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}\right). \tag{15.4}$$

Аналогично можно записать уравнение

$$f_2[j] = \min\left(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}\right). \tag{15.5}$$

Комбинируя уравнения (15.2)–(15.5), получим рекуррентные соотношения

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{при } j = 1, \\ \min\left(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}\right) & \text{при } j \geqslant 2. \end{cases} \tag{15.6}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{при } j = 1, \\ \min\left(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}\right) & \text{при } j \geqslant 2. \end{cases} \tag{15.7}$$

На рис. 15.2*б* приведены величины $f_i[j]$, соответствующие примеру, изображенному на рис. 15.2*а*, и вычисленные по уравнениям (15.6) и (15.7), а также величина f^*.

Величины $f_i[j]$ являются значениями, соответствующими оптимальным решениям подзадач. Чтобы было легче понять, как конструируется оптимальное решение, обозначим через $l_i[j]$ номер конвейера (1 или 2), содержащего рабочее место под номером $j-1$, через которое проходит самый быстрый путь к рабочему месту $S_{i,j}$. Индексы i и j принимают следующие значения: $i = 1, 2$; $j = 2, 3, \ldots, n$. (Величины $l_i[1]$ не определяются, поскольку на обоих конвейерах отсутствуют рабочие места, предшествующие рабочему месту под номером 1.) Кроме того, обозначим через l^* номер конвейера, через n-е рабочее место которого проходит самый быстрый полный путь сборки. Величины $l_i[j]$ облегчат определение самого быстрого пути. С помощью приведенных на рис. 15.2*б* величин l^* и $l_i[j]$ самый быстрый способ сборки на заводе, изображенном на рис. 15.2*а*, выясняется путем таких рассуждений. Начнем с $l^* = 1$; при этом используется рабочее место $S_{1,6}$. Теперь посмотрим на величину $l_1[6]$, которая равна 2, из чего следует, что используется рабочее место $S_{2,5}$. Продолжая рассуждения, смотрим на величину $l_2[5] = 2$ (используется рабочее место $S_{2,4}$), величину $l_2[4] = 1$ (рабочее место $S_{1,3}$), величину $l_1[3] = 2$ (рабочее место $S_{2,2}$) и величину $l_2[2] = 1$ (рабочее место $S_{1,1}$).

Третий этап: вычисление минимальных промежутков времени

На данном этапе на основе уравнения (15.1) и рекуррентных соотношений (15.6) и (15.7) легко было бы записать рекурсивный алгоритм определения самого быстрого пути сборки автомобиля на заводе. Однако с таким алгоритмом связана одна проблема: время его работы экспоненциально зависит от n. Чтобы понять, почему это так, введем значения $r_i(j)$, обозначающие количество ссылок в рекурсивном алгоритме на величины $f_i[j]$. Из уравнения (15.1) получаем:

$$r_1(n) = r_2(n) = 1. \tag{15.8}$$

Из рекуррентных уравнений (15.6) и (15.7) следует соотношение

$$r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1), \tag{15.9}$$

справедливое при $j = 1, 2, \ldots, n-1$. В упражнении 15.1-2 предлагается показать, что $r_i(j) = 2^{n-j}$. Таким образом, к величине $f_1[1]$ алгоритм обращается 2^{n-1} раз! В упражнении 15.1-3 предлагается показать, что полное количество обращений ко всем величинам $f_i[j]$ равно $\Theta(2^n)$.

Намного лучших результатов можно достичь, если вычислять величины $f_i[j]$ в порядке, отличном от рекурсивного. Обратите внимание, что при $j \geqslant 2$ каждое значение $f_i[j]$ зависит только от величин $f_1[j-1]$ и $f_2[j-1]$. Вычисляя значения $f_i[j]$ в порядке *увеличения* номеров рабочих мест j — слева направо (см. рис. 15.2*б*), — можно найти самый быстрый путь (и время его прохождения) по заводу в течение времени $\Theta(n)$. Приведенная ниже процедура FASTEST_WAY принимает в качестве входных данных величины $a_{i,j}$, $t_{i,j}$, e_i и x_i, а также количество рабочих мест на каждом конвейере n:

```
FASTEST_WAY(a, t, e, x, n)
 1  f_1[1] ← e_1 + a_{1,1}
 2  f_2[1] ← e_2 + a_{2,1}
 3  for j ← 2 to n
 4      do if f_1[j − 1] + a_{1,j} ≤ f_2[j − 1] + t_{2,j−1} + a_{1,j}
 5            then f_1[j] ← f_1[j − 1] + a_{1,j}
 6                 l_1[j] ← 1
 7            else f_1[j] ← f_2[j − 1] + t_{2,j−1} + a_{1,j}
 8                 l_1[j] ← 2
 9         if f_2[j − 1] + a_{2,j} ≤ f_1[j − 1] + t_{1,j−1} + a_{2,j}
10            then f_2[j] ← f_2[j − 1] + a_{2,j}
11                 l_2[j] ← 2
12            else f_2[j] ← f_1[j − 1] + t_{1,j−1} + a_{2,j}
13                 l_2[j] ← 1
```

```
14  if f_1[n] + x_1 ≤ f_2[n] + x_2
15      then f* = f_1[n] + x_1
16           l* = 1
17      else f* = f_2[n] + x_2
18           l* = 2
```

Процедура FASTEST_WAY работает следующим образом. В строках 1–2 с помощью уравнений (15.2) и (15.3) вычисляются величины $f_1[1]$ и $f_2[1]$. Затем в цикле **for** в строках 3–13, вычисляются величины $f_i[j]$ и $l_i[j]$ при $i = 1, 2$ и $j = 2, 3, \ldots, n$. В строках 4–8 с помощью уравнения (15.4) вычисляются величины $f_1[j]$ и $l_1[j]$, а в строках 9–13, с помощью уравнения (15.5), величины $f_2[j]$ и $l_2[j]$. Наконец, в строках 14–18 с помощью уравнения (15.1) вычисляются величины f^* и l^*. Поскольку строки 1–2 и 14–18 выполняются в течение фиксированного времени, и выполнение каждой из $n-1$ итераций цикла **for**, заданного в строках 3–13, тоже длится в течение фиксированного времени, на выполнение всей процедуры требуется время, равное $\Theta(n)$.

Один из способов вычисления значений $f_i[j]$ и $l_i[j]$ — заполнение ячеек соответствующей таблицы. Как видно из рис. 15.2*б*, таблицы для величин $f_i[j]$ и $l_i[j]$ заполняются слева направо (и сверху вниз в каждом столбце). Для вычисления величины $f_i[j]$ понадобятся значения $f_1[j-1]$ и $f_2[j-1]$. Зная, что эти величины уже вычислены и занесены в таблицу, их можно просто извлечь из соответствующих ячеек таблицы, когда они понадобятся.

Четвертый этап: построение самого быстрого пути

После вычисления величин $f_i[j]$, f^*, $l_i[j]$ и l^* нужно построить последовательность решений, используемых в самом быстром пути сборки. Ранее было рассказано, как это можно сделать для примера, проиллюстрированного на рис. 15.2.

Приведенная ниже процедура выводит в порядке убывания номера использующихся рабочих мест. В упражнении 15.1-1 предлагается модифицировать эту процедуру так, чтобы номера рабочих мест выводились в ней в порядке возрастания.

```
PRINT_STATIONS(l, n)
1  i ← l*
2  print "Конвейер " i ", рабочее место " n
3  for j ← n downto 2
4       do i ← l_i[j]
5          print "Конвейер " i ", рабочее место " j − 1
```

В примере, приведенном на рис. 15.2, процедура PRINT_STATIONS выведет такую последовательность рабочих мест:

Конвейер 1, рабочее место 6
Конвейер 2, рабочее место 5
Конвейер 2, рабочее место 4
Конвейер 1, рабочее место 3
Конвейер 2, рабочее место 2
Конвейер 1, рабочее место 1

Упражнения

15.1-1. Покажите, как модифицировать процедуру PRINT_STATIONS, чтобы она выводила данные в порядке возрастания номеров рабочих мест. (*Указание*: используйте рекурсию.)

15.1-2. С помощью уравнений (15.8) и (15.9) и метода подстановки покажите, что количество обращений $r_i(j)$ к величинам $f_i[j]$ в рекурсивном алгоритме равно 2^{n-j}.

15.1-3. Воспользовавшись результатом, полученном при выполнении упражнения 15.1-2, покажите, что полное число обращений ко всем величинам $f_i[j]$, выражающееся двойной суммой $\sum_{i=1}^{2}\sum_{j=1}^{n} r_i(j)$, равно $2^{n+1}-2$.

15.1-4. В таблицах, содержащих величины $f_i[j]$ и $l_i[j]$, общее количество ячеек равно $4n-2$. Покажите, как свести количество ячеек к $2n+2$ и при этом иметь возможность вычисления величины f^* и вывода информации о всех рабочих местах, составляющих самый быстрый путь прохождения завода.

15.1-5. Профессор предположил, что могут существовать значения e_i, $a_{i,j}$ и $t_{i,j}$, для которых процедура FASTEST_WAY дает такие значения $l_i[j]$, что для некоторого рабочего места j $l_1[j]=2$ и $l_2[j]=1$. Принимая во внимание, что все переходы выполняются в течение положительных промежутков времени $t_{i,j}$, покажите, что предположение профессора ложно.

15.2 Перемножение цепочки матриц

Следующий пример динамического программирования — алгоритм, позволяющий решить задачу о перемножении цепочки матриц. Пусть имеется последовательность (цепочка), состоящая из n матриц, и нужно вычислить их произведение

$$A_1 A_2 \cdots A_n. \tag{15.10}$$

Выражение (15.10) можно вычислить, используя в качестве подпрограммы стандартный алгоритм перемножения пар матриц. Однако сначала нужно расставить скобки, чтобы устранить все неоднозначности в порядке перемножения. Порядок

произведения матриц ***полностью определен скобками*** (fully parenthesized), если произведение является либо отдельной матрицей, либо взятым в скобки произведением двух подпоследовательностей матриц, в котором порядок перемножения полностью определен скобками. Матричное умножение обладает свойством ассоциативности, поэтому результат не зависит от расстановки скобок. Например, если задана последовательность матриц $\langle A_1, A_2, A_3, A_4 \rangle$, то способ вычисления их произведения можно полностью определить с помощью скобок пятью разными способами:

$$(A_1(A_2(A_3A_4)))\ ,$$
$$(A_1((A_2A_3)A_4))\ ,$$
$$((A_1A_2)(A_3A_4))\ ,$$
$$((A_1(A_2A_3))A_4)\ ,$$
$$(((A_1A_2)A_3)A_4)\ .$$

От того как расставлены скобки при перемножении последовательности матриц, может сильно зависеть время, затраченное на вычисление произведения. Сначала рассмотрим, как определить стоимость произведения двух матриц. Ниже приводится псевдокод стандартного алгоритма. Атрибуты $rows$ и $columns$ означают количество строк и столбцов матрицы.

```
MATRIX_MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2     then error "Несовместимые размеры"
3     else  for i ← 1 to rows[A]
4                 do for j ← 1 to columns[B]
5                       do C[i, j] ← 0
6                          for k ← 1 to columns[A]
7                              do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8           return C
```

Матрицы A и B можно перемножать, только если они ***совместимы***: количество столбцов матрицы A должно совпадать с количеством строк матрицы B. Если A — это матрица размера $p \times q$, а B — матрица размера $q \times r$, то в результате их перемножения получится матрица C размера $p \times r$. Время вычисления матрицы C преимущественно определяется количеством произведений скаляров (далее в главе для краткости будем называть эту операцию скалярным умножением — *Прим. перев.*), которое выполняется в строке 7. Это количество равно pqr. Итак, стоимость умножения матриц будет выражаться в количестве умножений скалярных величин.

Чтобы проиллюстрировать, как расстановка скобок при перемножении нескольких матриц влияет на количество выполняемых операций, рассмотрим при-

мер, в котором перемножаются три матрицы $\langle A_1, A_2, A_3\rangle$. Предположим, что размеры этих матриц равны 10×100, 100×5 и 5×50 соответственно. Перемножая матрицы в порядке, заданном выражением $((A_1 A_2)\, A_3)$, необходимо выполнить $10 \cdot 100 \cdot 5 = 5\,000$ скалярных умножений, чтобы найти результат произведения $A_1 A_2$ (при этом получится матрица размером 10×5), а затем — еще $10 \cdot 5 \cdot 50 = 2\,500$ скалярных умножений, чтобы умножить эту матрицу на матрицу A_3. Всего получается 7 500 скалярных умножений. Если вычислять результат в порядке, заданном выражением $(A_1\,(A_2 A_3))$, то сначала понадобится выполнить $100 \cdot 5 \cdot 50 = 25\,000$ скалярных умножений (при этом будет найдена матрица $A_2 A_3$ размером 100×50), а затем еще $10 \cdot 100 \cdot 50 = 50\,000$ скалярных умножений, чтобы умножить A_1 на эту матрицу. Всего получается 75 000 скалярных умножений. Таким образом, для вычисления результата первым способом понадобится в 10 раз меньше времени.

Задачу о перемножении последовательности матриц (matrix-chain multiplication problem) можно сформулировать так: для заданной последовательности матриц $\langle A_1, A_2, \ldots, A_n\rangle$, в которой матрица A_i, $i = 1, 2, \ldots, n$ имеет размер $p_{i-1} \times p_i$, с помощью скобок следует полностью определить порядок умножений в матричном произведении $A_1 A_2 \cdots A_n$, при котором количество скалярных умножений сведется к минимуму.

Обратите внимание, что само перемножение матриц в задачу не входит. Наша цель — определить оптимальный порядок перемножения. Обычно время, затраченное на нахождение оптимального способа перемножения матриц, с лихвой окупается, когда выполняется само перемножение (как это было в рассмотренном примере, когда удалось обойтись всего 7 500 скалярными умножениями вместо 75 000).

Подсчет количества способов расстановки скобок

Прежде чем приступить к решению задачи об умножении последовательности матриц методами динамического программирования, заметим, что исчерпывающая проверка всех возможных вариантов расстановки скобок не является эффективным алгоритмом ее решения. Обозначим через $P(n)$ количество альтернативных способов расстановки скобок в последовательности, состоящей из n матриц. Если $n = 1$, то матрица всего одна, поэтому скобки в матричном произведении можно расставить всего одним способом. Если $n \geqslant 2$, то произведение последовательности матриц, в котором порядок перемножения полностью определен скобками, является произведением двух таких произведений подпоследовательностей матриц, в которых порядок перемножения тоже полностью определен скобками. Разбиение на подпоследовательности может производиться на границе k-й и $k+1$-й матриц для любого $k = 1, 2, \ldots, n-1$. В результате получаем

рекуррентное соотношение

$$P(n) = \begin{cases} 1 & \text{при } n = 1, \\ \sum\limits_{k=1}^{n-1} P(k) P(n-k) & \text{при } n \geqslant 2. \end{cases} \tag{15.11}$$

В задаче 12-4 предлагается показать, что решением аналогичного рекуррентного соотношения является последовательность ***чисел Каталана*** (Catalan numbers), возрастающая как $\Omega\left(4^n/n^{3/2}\right)$. Более простое упражнение (упражнение 15.2-3) заключается в том, чтобы показать, что решение рекуррентного соотношения (15.11) ведет себя как $\Omega(2^n)$. Таким образом, количество вариантов расстановки скобок экспоненциально увеличивается с ростом n, и метод прямого перебора всех вариантов не подходит для определения оптимальной стратегии расстановки скобок в матричном произведении.

Первый этап: структура оптимальной расстановки скобок

Первый этап применения парадигмы динамического программирования — найти оптимальную вспомогательную подструктуру, а затем с ее помощью сконструировать оптимальное решение задачи по оптимальным решениям вспомогательных задач. В рассматриваемой задаче этот этап можно осуществить следующим образом. Обозначим для удобства результат перемножения матриц $A_i A_{i+1} \cdots A_j$ через $A_{i..j}$, где $i \leqslant j$. Заметим, что если задача нетривиальна, т.е. $i < j$, то любой способ расстановки скобок в произведении $A_i A_{i+1} \cdots A_j$ разбивает это произведение между матрицами A_k и A_{k+1}, где k — целое, удовлетворяющее условию $i \leqslant k < j$. Таким образом, при некотором k сначала вычисляются матрицы $A_{i..k}$ и $A_{k+1..j}$, а затем они умножаются друг на друга, в результате чего получается произведение $A_{i..j}$. Стоимость, соответствующая этому способу расстановки скобок, равна сумме стоимости вычисления матрицы $A_{i..k}$, стоимости вычисления матрицы $A_{k+1..j}$ и стоимости вычисления их произведения.

Ниже описывается оптимальная вспомогательная подструктура для данной задачи. Предположим, что в результате оптимальной расстановки скобок последовательность $A_i A_{i+1} \ldots A_j$ разбивается на подпоследовательности между матрицами A_k и A_{k+1}. Тогда расстановка скобок в "префиксной" подпоследовательности $A_i A_{i+1} \ldots A_k$ тоже должна быть оптимальной. Почему? Если бы существовал более экономный способ расстановки скобок в последовательности $A_i A_{i+1} \ldots A_k$, то его применение позволило бы перемножить матрицы $A_i A_{i+1} \ldots A_j$ еще эффективнее, что противоречит предположению об оптимальности первоначальной расстановки скобок. Аналогично можно прийти к выводу, что расстановка скобок в подпоследовательности матриц $A_{k+1} A_{k+1} \ldots A_j$, возникающей в результате

оптимальной расстановки скобок в последовательности $A_i A_{i+1} \dots A_j$, также должна быть оптимальной.

Теперь с помощью нашей оптимальной вспомогательной подструктуры покажем, что оптимальное решение задачи можно составить из оптимальных решений вспомогательных задач. Мы уже убедились, что для решения любой нетривиальной задачи об оптимальном произведении последовательности матриц всю последовательность необходимо разбить на подпоследовательности и что каждое оптимальное решение содержит в себе оптимальные решения подзадач. Другими словами, решение полной задачи об оптимальном перемножении последовательности матриц можно построить путем разбиения этой задачи на две подзадачи — оптимальную расстановку скобок в подпоследовательностях $A_i A_{i+1} \dots A_k$ и $A_{k+1} A_{k+2} \dots A_j$. После этого находятся оптимальные решения подзадач, из которых затем составляется оптимальное решение полной задачи. Необходимо убедиться, что при поиске способа перемножения матриц учитываются все возможные разбиения — только в этом случае можно быть уверенным, что найденное решение будет глобально оптимальным.

Второй этап: рекурсивное решение

Далее, рекурсивно определим стоимость оптимального решения в терминах оптимальных решений вспомогательных задач. В задаче о перемножении последовательности матриц в качестве вспомогательной задачи выбирается задача об оптимальной расстановке скобок в подпоследовательности $A_i A_{i+1} \dots A_j$ при $1 \leqslant i \leqslant j \leqslant n$. Путь $m[i, j]$ — минимальное количество скалярных умножений, необходимых для вычисления матрицы $A_{i..j}$. Тогда в полной задаче минимальная стоимость матрицы $A_{1..n}$ равна $m[1, n]$.

Определим величину $m[i, j]$ рекурсивно следующим образом. Если $i = j$, то задача становится тривиальной: последовательность состоит всего из одной матрицы $A_{i..j} = A_j$, и для вычисления произведения матриц не нужно выполнять никаких скалярных умножений. Таким образом, при $i = 1, 2, \dots, n$ $m[i, i] = 0$. Чтобы вычислить $m[i, j]$ при $i < j$, воспользуемся свойством подструктуры оптимального решения, исследованным на первом этапе. Предположим, что в результате оптимальной расстановки скобок последовательность $A_i A_{i+1} \dots A_j$ разбивается между матрицами A_k и A_{k+1}, где $i \leqslant k < j$. Тогда величина $m[i, j]$ равна минимальной стоимости вычисления частных произведений $A_{i..k}$ и $A_{k+1..j}$ плюс стоимость умножения этих матриц друг на друга. Если вспомнить, что каждая матрица A_i имеет размеры $p_{i-1} \times p_i$, то нетрудно понять, что для вычисления произведения матриц $A_{i..k} A_{k+1..j}$ понадобится $p_{i-1} p_k p_j$ скалярных умножений. Таким образом, получаем:

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j.$$

В этом рекурсивном уравнении предполагается, что значение k известно, но на самом деле это не так. Для выбора этого значения всего имеется $j - i$ возможностей, а именно — $k = i, i + 1, \ldots, j - 1$. Поскольку в оптимальной расстановке скобок необходимо использовать одно из этих значений k, все, что нужно сделать, — проверить все возможности и выбрать среди них лучшую. Таким образом, рекурсивное определение оптимальной расстановки скобок в произведении $A_i A_{i+1} \ldots A_j$ принимает вид:

$$m[i,j] = \begin{cases} 0 & \text{при } i = j, \\ \min\limits_{i \leqslant k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{при } i < j. \end{cases} \qquad (15.12)$$

Величины $m[i,j]$ равны стоимостям оптимальных решений вспомогательных задач. Чтобы легче было проследить за процессом построения оптимального решения, обозначим через $s[i,j]$ значение k, при котором последовательность $A_i A_{i+1} \ldots A_j$ разбивается на две подпоследовательности в процессе оптимальной расстановки скобок. Таким образом, величина $s[i,j]$ равна значению k, такому что $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$.

Третий этап: вычисление оптимальной стоимости

На данном этапе не составляет труда написать на основе рекуррентного соотношения (15.12) рекурсивный алгоритм для вычисления минимальной стоимости $m[1,n]$ для произведения $A_1 A_2 \ldots A_n$. Однако в разделе 15.3 мы сможем убедиться, что время работы этого алгоритма экспоненциально зависит от n, что ничем не лучше метода прямого перебора, при котором проверяется каждый способ расстановки скобок в произведении.

Важное наблюдение, которое можно сделать на данном этапе, заключается в том, что у нас относительно мало подзадач: по одной для каждого выбора величин i и j, удовлетворяющих неравенству $1 \leqslant i \leqslant j \leqslant n$, т.е. всего $\binom{n}{2} + n = \Theta(n^2)$. В рекурсивном алгоритме каждая вспомогательная задача может неоднократно встречаться в разных ветвях рекурсивного дерева. Такое свойство перекрытия вспомогательных подзадач — вторая отличительная черта применимости метода динамического программирования (первая отличительная черта — наличие оптимальной подструктуры).

Вместо того чтобы рекурсивно решать рекуррентное соотношение (15.12), выполним третий этап парадигмы динамического программирования и вычислим оптимальную стоимость путем построения таблицы в восходящем направлении. В описанной ниже процедуре предполагается, что размеры матриц A_i равны $p_{i-1} \times p_i$ $(i = 1, 2, \ldots, n)$. Входные данные представляют собой последовательность $p = \langle p_0, p_1, \ldots, p_n \rangle$; длина данной последовательности равна $length[p] = n + 1$. В процедуре используется вспомогательная таблица $m[1..n, 1..n]$ для хранения

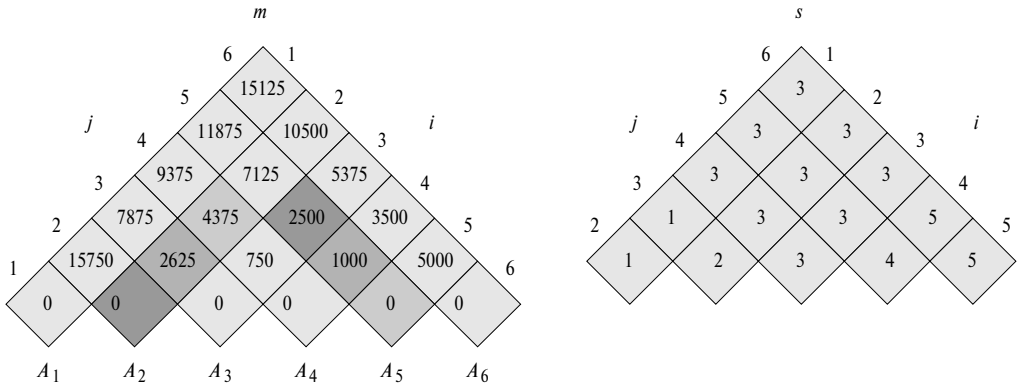
стоимостей $m[i, j]$ и вспомогательная таблица $s[1..n, 1..n]$, в которую заносятся индексы k, при которых достигаются оптимальные стоимости $m[i, j]$. Таблица s будет использоваться при построении оптимального решения.

Чтобы корректно реализовать восходящий подход, необходимо определить, с помощью каких записей таблицы будут вычисляться величины $m[i, j]$. Из уравнения (15.12) видно, что стоимость $m[i, j]$ вычисления произведения последовательности $j - i + 1$ матриц зависит только от стоимости вычисления последовательностей матриц, содержащих менее $j - i + 1$ матриц. Другими словами, при $k = i, i + 1, \ldots, j - 1$ матрица $A_{i..k}$ представляет собой произведение $k - i + 1 < j - i + 1$ матриц, а матрица $A_{k+1..j}$ — произведение $j - k < j - i + 1$ матриц. Таким образом, в ходе выполнения алгоритма следует организовать заполнение таблицы m в порядке, соответствующем решению задачи о расстановке скобок в последовательностях матриц возрастающей длины:

```
MATRIX_CHAIN_ORDER(p)
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3       do m[i, i] ← 0
 4  for l ← 2 to n              ▷ l — длина последовательности
 5       do for i ← 1 to n − l + 1
 6               do j ← i + l − 1
 7                  m[i, j] ← ∞
 8                  for k ← i to j − 1
 9                       do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                          if q < m[i, j]
11                            then m[i, j] ← q
12                                 s[i, j] ← k
13  return m и s
```

Сначала в этом алгоритме (строки 2–3) вычисляются величины $m[i, i] \leftarrow \leftarrow 0$ ($i = 1, 2, \ldots, n$), которые представляют собой минимальные стоимости для последовательностей единичной длины. Затем в первой итерации цикла в строках 4–12 с помощью рекуррентного соотношения (15.12) вычисляются величины $m[i, i + 1]$ при $i = 1, 2, \ldots, n - 1$ (минимальные стоимости для последовательностей длины $l = 2$). При втором проходе этого цикла вычисляются величины $m[i, i + 2]$ при $i = 1, 2, \ldots, n - 2$ (минимальные стоимости для последовательностей длины $l = 3$) и т.д. На каждом этапе вычисляемые в строках 9–12 величины $m[i, j]$ зависят только от уже вычисленных и занесенных в таблицу значений $m[i, k]$ и $m[k + 1, j]$.

На рис. 15.3 описанный выше процесс проиллюстрирован для цепочки, состоящей из $n = 6$ матриц, размеры которых равны: $A_1 - 30 \times 35$, $A_2 - 35 \times 15$, $A_3 - 15 \times 5$, $A_4 - 5 \times 10$, $A_5 - 10 \times 20$, $A_6 - 20 \times 25$. Поскольку величины $m[i, j]$ опре-



Матрица	Размерность
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Рис. 15.3. Таблицы m и s, вычисленные процедурой MATRIX_CHAIN_ORDER для $n = 6$

делены только при $i \leqslant j$, используется только часть таблицы m, расположенная над ее главной диагональю. То же можно сказать и о таблице s. На рис. 15.3 таблица повернута так, чтобы ее главная диагональ была расположена горизонтально. В нижней части рисунка приведен список матриц, входящих в последовательность. На этой схеме легко найти минимальную стоимость $m[i, j]$ перемножения частичной последовательности матриц $A_i A_{i+1} \ldots A_j$. Она находится на пересечении линий, идущих от матрицы A_i вправо и вверх, и от матрицы A_j — влево и вверх. В каждой горизонтальной строке таблицы содержатся стоимости перемножения частных последовательностей, состоящих из одинакового количества матриц. В процедуре MATRIX_CHAIN_ORDER строки вычисляются снизу вверх, а элементы в каждой строке — слева направо. Величина $m[i, j]$ вычисляется с помощью произведений $p_{i-1} p_k p_j$ для $k = i, i + 1, \ldots, j - 1$ и величин внизу слева и внизу справа от $m[i, j]$. Из таблицы m видно, что минимальное количество скалярных умножений, необходимых для вычисления произведения шести матриц, равно $m[1, 6] = 15\,125$. Для пояснения приведем пример. При вычислении элемента $m[5, 2]$ использовались пары элементов, идущие от матриц A_2 и A_5

и имеющие одинаковый оттенок фона:

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000 \ , \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \ , \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$
$$= 7125 \ .$$

Несложный анализ структуры вложенных циклов в процедуре MATRIX_CHAIN_ORDER показывает, что время ее работы составляет $O\left(n^3\right)$. Глубина вложения циклов равна трем, а индексы в каждом из них (l, i и k) принимают не более $n-1$ значений. В упражнении 15.2-4 предлагается показать, что время работы этого алгоритма фактически равно $\Omega\left(n^3\right)$. Для хранения таблиц m и s требуется объем, равный $\Theta\left(n^2\right)$. Таким образом, процедура MATRIX_CHAIN_ORDER намного эффективнее, чем метод перебора и проверки всевозможных способов расстановки скобок, время работы которого экспоненциально зависит от количества перемножаемых матриц.

Четвертый этап: конструирование оптимального решения

Несмотря на то, что в процедуре MATRIX_CHAIN_ORDER определяется оптимальное количество скалярных произведений, необходимых для вычисления произведения последовательности матриц, в нем не показано, как именно перемножаются матрицы. Оптимальное решение несложно построить с помощью информации, хранящейся в таблице s. В каждом элементе $s\,[i,j]$ хранится значение индекса k, где при оптимальной расстановке скобок в последовательности $A_i A_{i+1} \ldots A_j$ выполняется разбиение. Таким образом, нам известно, что оптимальное вычисление произведения матриц $A_{1..n}$ выглядит как $A_{1..s[1,n]} A_{s[1,n]+1..n}$. Эти частные произведения матриц можно вычислить рекурсивно, поскольку элемент $s\,[1, s[1,n]]$ определяет матричное умножение, выполняемое последним при вычислении $A_{1..s[1,n]}$, а $s\,[s[1,n]+1, n]$ — последнее умножение при вычислении $A_{s[1,n]+1..n}$. Приведенная ниже рекурсивная процедура выводит оптимальный способ расстановки скобок в последовательности матриц $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, по таблице s, полученной в результате работы процедуры MATRIX_CHAIN_ORDER, и индексам i и j. В результате вызова процедуры PRINT_OPTIMAL_PARENS$(s, 1, n)$ выводится оптимальная расстановка скобок в последовательности $\langle A_1, A_2, \ldots, A_n \rangle$:

```
PRINT_OPTIMAL_PARENS(s, i, j)
1  if i = j
2     then print "A"_i
3     else  print "("
```

```
4       PRINT_OPTIMAL_PARENS(s, i, s[i, j])
5       PRINT_OPTIMAL_PARENS(s, s[i, j] + 1, j)
6       print ")"
```

В примере, проиллюстрированном на рис. 15.3, в результате вызова процедуры PRINT_OPTIMAL_PARENS$(s, 1, 6)$ выводится строка $((A_1\,(A_2A_3))\,((A_4A_5)\,A_6))$.

Упражнения

15.2-1. Определите оптимальный способ расстановки скобок в произведении последовательности матриц, размеры которых равны $\langle 5, 10, 3, 12, 5, 50, 6\rangle$.

15.2-2. Разработайте рекурсивный алгоритм MATRIX_CHAIN_MULTIPLY(A, s, i, j), в котором оптимальным образом вычисляется произведение заданной последовательности матриц $\langle A_1A_2, \ldots, A_n\rangle$. На вход этого алгоритма также подаются индексы i и j, а также таблица s, вычисленная с помощью процедуры MATRIX_CHAIN_ORDER. (Начальный вызов этой процедуры выглядит следующим образом: MATRIX_CHAIN_MULTIPLY$(A, s, 1, n)$.)

15.2-3. Покажите с помощью метода подстановок, что решение рекуррентного соотношения (15.11) ведет себя как $\Omega\,(2^n)$.

15.2-4. Пусть $R\,(i, j)$ — количество обращений к элементу матрицы $m\,[i, j]$, которые выполняются в ходе вычисления других элементов этой матрицы в процедуре MATRIX_CHAIN_ORDER. Покажите, что полное количество обращений ко всем элементам равно:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} R\,(i, j) = \frac{n^3 - n}{3}.$$

(*Указание*: при решении может оказаться полезным уравнение (A.3).)

15.2-5. Покажите, что в полной расстановке скобок в n-элементном выражении используется ровно $n - 1$ пар скобок.

15.3 Элементы динамического программирования

Несмотря на рассмотренные в предыдущем разделе два примера, в которых применялся метод динамического программирования, возможно, вам все еще не совсем понятно, когда применим этот метод. В каких случаях задачу следует решать с помощью динамического программирования? В данном разделе рассматриваются два ключевых ингредиента, которые должны быть присущи задаче

оптимизации, чтобы к ней можно было применить метод динамического программирования: наличие оптимальной подструктуры и перекрывающиеся вспомогательные программы. Мы также рассмотрим разновидность метода, который называется *запоминанием* (memoization) и позволяет воспользоваться свойством перекрывающихся вспомогательных программ.

Оптимальная подструктура

Первый шаг решения задачи оптимизации методом динамического программирования состоит в том, чтобы охарактеризовать структуру оптимального решения. Напомним, что ***оптимальная подструктура*** проявляется в задаче в том случае, если в ее оптимальном решении содержатся оптимальные решения вспомогательных подзадач. Если в задаче выявлено наличие оптимальной подструктуры, это служит веским аргументом в пользу того, что к ней может быть применим метод динамического программирования. (Однако наличие этого свойства может также свидетельствовать о применимости жадных алгоритмов; см. главу 16.) В динамическом программировании оптимальное решение задачи конструируется из оптимальных решений вспомогательных задач. Следовательно, необходимо убедиться в том, что в число рассматриваемых подзадач входят те, которые используются в оптимальном решении.

Оптимальная подструктура была обнаружена в обеих задачах, которые исследовались в настоящей главе. В разделе 15.1 было установлено, что самый быстрый путь до рабочего места с номером j, расположенного на одном из двух конвейеров, включает в себя самый быстрый путь до рабочего места с номером $j-1$. В разделе 15.2 было продемонстрировано, что в оптимальной расстановке скобок, в результате которой последовательность матриц $A_i A_{i+1} \dots A_j$ разбивается на подпоследовательности между матрицами A_k и A_{k+1}, содержатся оптимальные решения задач о расстановке скобок в подпоследовательностях $A_i A_{i+1} \dots A_k$ и $A_{k+1} A_{k+2} \dots A_j$.

Можно видеть, что поиск оптимальной подструктуры происходит по общему образцу.

1. На первом этапе следует показать, что в процессе решения задачи приходится делать выбор. В рассмотренных примерах это был выбор рабочего места на конвейере или выбор индекса, при котором разбивается последовательность матриц. После выбора остается решить одну или несколько вспомогательных задач.
2. На этом этапе мы исходим из того, что для поставленной задачи делается выбор, ведущий к оптимальному решению. Пока что не рассматривается, как именно следует делать выбор, а просто предполагается, что он найден.

3. Исходя из предположения предыдущего этапа, определяется, какие вспомогательные задачи получаются и как лучше охарактеризовать получающееся в результате пространство подзадач.
4. Показывается, что решения вспомогательных задач, возникающих в ходе оптимального решения задачи, сами должны быть оптимальными. Это делается методом "от противного": предполагая, что решение каждой вспомогательной задачи не оптимально, приходим к противоречию. В частности, путем "вырезания" неоптимального решения вспомогательной задачи и "вставки" оптимального демонстрируется, что можно получить лучшее решение исходной задачи, а это противоречит предположению, что уже имеется оптимальное решение. Если вспомогательных задач несколько, они обычно настолько похожи, что описанный выше способ рассуждения, примененный к одной из вспомогательных задач, легко модифицируется для остальных.

Характеризуя пространство подзадач, постарайтесь придерживаться такого практического правила: попытайтесь, чтобы это пространство было как можно проще, а потом расширьте его до необходимых пределов. Например, пространство в подзадачах, возникающих в задаче о составлении расписания работы конвейера, было образовано самым быстрым путем прохождения шасси от начала конвейера до рабочего места $S_{1,j}$ или $S_{2,j}$. Это подпространство оказалось вполне подходящим, и не возникло необходимости его расширять.

Теперь предположим, что в задаче о перемножении цепочки матриц производится попытка ограничить пространство подзадач произведением вида $A_1 A_2 \ldots A_j$. Как и раньше, оптимальная расстановка скобок должна разбивать произведение между матрицами A_k и A_{k+1} для некоторого индекса $1 \leqslant k < j$. Если k не всегда равно $j-1$, мы обнаружим, что возникают вспомогательные задачи в виде $A_1 A_2 \ldots A_k$ и $A_{k+1} A_{k+2} \ldots A_j$ и что последняя подзадача не является подзадачей вида $A_1 A_2 \ldots A_j$. В этой задаче необходима возможность изменения обоих концов последовательности, т.е. чтобы во вспомогательной задаче $A_i A_{i+1} \ldots A_j$ могли меняться оба индекса, — и i, и j.

Оптимальная подструктура изменяется в области определения задачи в двух аспектах:

1. количество вспомогательных задач, которые используются при оптимальном решении исходной задачи;
2. количество выборов, возникающих при определении того, какая вспомогательная задача (задачи) используется в оптимальном решении.

В задаче о составлении расписания работы конвейера в оптимальном решении используется только одна вспомогательная задача, и для поиска оптимального решения необходимо рассмотрение двух вариантов. Чтобы найти самый быстрый

путь через рабочее место $S_{i,j}$, мы используем *либо* самый быстрый путь через рабочее место $S_{1,j-1}$, *либо* самый быстрый путь через рабочее место $S_{2,j-1}$. Каждый из вариантов представляет одну вспомогательную задачу, для которой необходимо найти оптимальное решение. Перемножение вспомогательной цепочки матриц $A_iA_{i+1}\ldots A_j$ — пример, в котором возникают две вспомогательные задачи и $j-i$ вариантов выбора. Если задана матрица A_k, у которой происходит разбиение произведения, то возникают две вспомогательные задачи — расстановка скобок в подпоследовательностях $A_iA_{i+1}\ldots A_k$ и $A_{k+1}A_{k+2}\ldots A_j$, причем для *каждой* из них необходимо найти оптимальное решение. Как только оптимальные решения вспомогательных задач найдены, выбирается один из $j-i$ вариантов индекса k.

Если говорить неформально, время работы алгоритма динамического программирования зависит от произведения двух множителей: общего количества вспомогательных задач и количества вариантов выбора, возникающих в каждой вспомогательной задаче. При составлении расписания работы сборочного конвейера у нас всего возникало $\Theta(n)$ вспомогательных задач и лишь два варианта выбора, которые нужно было проверить. В результате получается, что время работы алгоритма ведет себя как $\Theta(n)$. При перемножении матриц всего возникало $\Theta\left(n^2\right)$ вспомогательных задач, и в каждой из них — не более $n-1$ вариантов выбора, поэтому время работы алгоритма ведет себя как $O\left(n^3\right)$.

Оптимальная подструктура используется в динамическом программировании в восходящем направлении. Другими словами, сначала находятся оптимальные решения вспомогательных задач, после чего определяется оптимальное решение поставленной задачи. Поиск оптимального решения задачи влечет за собой необходимость выбора одной из вспомогательных задач, которые будут использоваться в решении полной задачи. Стоимость решения задачи обычно определяется как сумма стоимостей решений вспомогательных задач и стоимости, которая затрачивается на определение правильного выбора. Например, при составлении расписания работы сборочного конвейера сначала решались вспомогательные задачи по определению самого быстрого пути через рабочие места $S_{1,j-1}$ и $S_{2,j-1}$, а потом одна из них выбиралась в качестве предыдущей для рабочего места $S_{i,j}$. Стоимость, которая относится к самому выбору, зависит от того, происходит ли переход от одного конвейера к другому между рабочими местами $j-1$ и j. Если шасси остается на одном и том же конвейере, то эта стоимость равна $a_{i,j}$, а если осуществляется переход от одного конвейера к другому — она равна $t_{i',j-1}+a_{i,j}$, где $i'\neq i$. В задаче о перемножении цепочки матриц сначала был определен оптимальный способ расстановки скобок во вспомогательной цепочке $A_iA_{i+1}\ldots A_j$, а потом была выбрана матрица A_k, у которой выполняется разбиение произведения. Стоимость, которая относится к самому выбору, выражается членом $p_{i-1}p_kp_j$.

В главе 16 рассматриваются жадные алгоритмы, имеющие много общего с динамическим программированием. В частности, задачи, к которым применимы

жадные алгоритмы, обладают оптимальной подструктурой. Одно из характерных различий между жадными алгоритмами и динамическим программированием заключается в том, что в жадных алгоритмах оптимальная подструктура используется в нисходящем направлении. Вместо того чтобы находить оптимальные решения вспомогательных задач с последующим выбором одного из возможных вариантов, в жадных алгоритмах сначала делается выбор, который выглядит наилучшим на текущий момент, а потом решается возникшая в результате вспомогательная задача.

Некоторые тонкости

Следует быть особенно внимательным в отношении вопроса о применимости оптимальной подструктуры, когда она отсутствует в задаче. Рассмотрим такие две задачи, в которых имеется ориентированный граф $G = (V, E)$ и вершины $u, v \in V$.

Задача о кратчайшем невзвешенном пути[1]**.** Нужно найти путь от вершины u к вершине v, состоящий из минимального количества ребер. Этот путь обязан быть простым, поскольку в результате удаления из него цикла получается путь, состоящий из меньшего количества ребер.

Задача о самом длинном невзвешенном пути. Определите простой путь от вершины u к вершине v, состоящий из максимального количества ребер. Требование простоты весьма важно, поскольку в противном случае можно проходить по одному и тому же циклу сколько угодно раз, получая в результате путь, состоящий из произвольного количества ребер.

В задаче о кратчайшем пути возникает оптимальная подструктура. Это можно показать с помощью таких рассуждений. Предположим, что $u \neq v$ и задача является нетривиальной. В этом случае любой путь p от u к v должен содержать промежуточную вершину, скажем, w. (Заметим, что вершина w может совпадать с вершиной u или v.) Поэтому путь $u \overset{p}{\rightsquigarrow} v$ можно разложить на вспомогательные пути $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$. Очевидно, что количество ребер, входящих в путь p, равно сумме числа ребер в путях p_1 и p_2. Утверждается, что если путь p от вершины u к вершине v оптимальный (т.е. кратчайший), то p_1 должен быть кратчайшим путем от вершины u к вершине w. Почему? Аргумент такой: если бы существовал другой путь, соединяющий вершины u и v и состоящий из меньшего количества ребер, чем p_1, скажем, p'_1, можно было бы вырезать путь p_1 и вставить путь p'_1, в результате чего получился бы путь $u \overset{p'_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$, состоящий из меньшего количества ребер, чем путь p. А это противоречит предположению об оптимальности пути p. Аналогично, p_2 должен быть кратчайшим путем от вершины w

[1] Термин “невзвешенный” используется, чтобы отличать эту задачу от той, при которой находится кратчайший путь на взвешенных ребрах, с которой мы ознакомимся в главах 24 и 25. Для решения задачи о невзвешенном пути можно использовать метод поиска в ширину, описанный в главе 26.

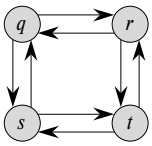


Рис. 15.4. Пример, демонстрирующий отсутствие оптимальной подструктуры в задаче о поиске самого длинного простого пути

к вершине v. Таким образом, кратчайший путь от вершины u к вершине v можно найти, рассмотрев все промежуточные вершины w, отыскав кратчайший путь от вершины u к вершине w и кратчайший путь от вершины w к вершине v, и выбрав промежуточную вершину w, через которую весь путь окажется кратчайшим. В разделе 25.2 один из вариантов этой оптимальной подструктуры используется для поиска кратчайшего пути между всеми парами вершин во взвешенном ориентированном графе.

Напрашивается предположение, что в задаче поиска самого длинного простого невзвешенного пути тоже проявляется оптимальная подструктура. В конце концов, если разложить самый длинный простой путь $u \overset{p}{\rightsquigarrow} v$ на вспомогательные пути $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$, то разве не должен путь p_1 быть самым длинным простым путем от вершины u к вершине w, а путь p_2 — самым длинным простым путем от вершины w к вершине v? Оказывается, нет! Пример такой ситуации приведен на рис. 15.4. Рассмотрим путь $q \to r \to t$, который является самым длинным простым путем от вершины q к вершине t. Является ли путь $q \to r$ самым длинным путем от вершины q к вершине r? Нет, поскольку простой путь $q \to s \to t \to r$ длиннее. Является ли путь $r \to t$ самым длинным путем от вершины r к вершине t? Снова нет, поскольку простой путь $r \to q \to s \to t$ длиннее.

Этот пример демонстрирует, что в задаче о самых длинных простых путях не только отсутствует оптимальная подструктура, но и не всегда удается составить "законное" решение задачи из решений вспомогательных задач. Если сложить самые длинные простые пути $q \to s \to t \to r$ и $r \to q \to s \to t$, то получим путь $q \to s \to t \to r \to q \to s \to t$, который не является простым. Итак, на деле оказывается, что в задаче о поиске самого длинного невзвешенного пути не возникает никаких оптимальных подструктур. Для этой задачи до сих пор не найдено ни одного эффективного алгоритма, работающего по принципу динамического программирования. Фактически, это NP-полная задача, что означает — как будет показано в главе 34 — что вряд ли ее можно решить в течение полиномиального времени.

Что можно сказать по поводу структуры самого длинного простого пути, так отличающейся от структуры самого короткого пути? Несмотря на то, что

в решениях задач о поиске и самого короткого, и самого длинного пути возникают две вспомогательные задачи, вспомогательные задачи при определении самого длинного пути не являются ***независимыми***, в то время как в задаче о кратчайшем пути они независимы. Что подразумевается под независимостью вспомогательных задач? Подразумевается, что решение одной вспомогательной задачи не влияет на решение другой вспомогательной задачи, возникающей в той же задаче. В примере, проиллюстрированном на рис. 15.4, рассматривается задача о поиске самого длинного простого пути между вершинами q и t, в которой возникают две вспомогательные задачи: определение самых длинных простых путей между вершинами q и r и между вершинами r и t. Для первой из этих вспомогательных задач выбирается путь $q \to s \to t \to r$, в котором используются вершины s и t. Во второй вспомогательной задаче мы не сможем больше использовать эти вершины, поскольку в процессе комбинирования решений этих двух вспомогательных задач получился бы путь, который не является простым. Если во второй задаче нельзя использовать вершину t, то ее вообще невозможно будет решить, поскольку эта вершина должна быть в найденном пути, и это не та вершина, в которой "соединяются" решения вспомогательных задач (этой вершиной является r). Использование вершин s и t в решении одной вспомогательной задачи приводит к невозможности их применения в решении другой вспомогательной задачи. Однако для ее решения необходимо использовать хотя бы одну из них, а в оптимальное решение данной вспомогательной задачи входят обе эти вершины. Поэтому эти вспомогательные задачи не являются независимыми. Другими словами, использование ресурсов в решении одной вспомогательной задачи (в качестве ресурсов выступают вершины) делают их недоступными в другой вспомогательной задаче.

Почему же вспомогательные задачи остаются независимыми при поиске самого короткого пути? Ответ такой: по самой природе поставленной задачи возникающие в ней вспомогательные задачи не используют одни и те же ресурсы. Утверждается, что если вершина w находится на кратчайшем пути p от вершины u к вершине v, то можно соединить *любой* кратчайший путь $u \overset{p_1}{\rightsquigarrow} w$ с *любым* кратчайшим путем $w \overset{p_2}{\rightsquigarrow} v$, получив в результате самый короткий путь от вершины u к вершине v. Мы уверены в том, что пути p_1 и p_2 не содержат ни одной общей вершины, кроме w. Почему? Предположим, что имеется еще некоторая вершина $x \neq w$, принадлежащая путям p_1 и p_2, так что путь p_1 можно разложить как $u \overset{p_{ux}}{\rightsquigarrow} x \rightsquigarrow w$, а путь p_2 — как $w \rightsquigarrow x \overset{p_{xv}}{\rightsquigarrow} v$. В силу оптимальной подструктуры этой задачи количество ребер в пути p равно сумме количеств ребер в путях p_1 и p_2. Предположим, что путь p содержит e ребер. Теперь построим путь $u \overset{p_{ux}}{\rightsquigarrow} x \overset{p_{xv}}{\rightsquigarrow} v$ от вершины u к вершине v. В этом пути содержится не более $e - 2$ вершин, что противоречит предположению о том, что путь p — кратчайший. Таким образом,

мы убедились, что вспомогательные задачи, возникающие в задаче поиска кратчайшего пути, являются независимыми.

Подзадачи, возникающие в задачах, которые рассматриваются в разделах 15.1 и 15.2, являются независимыми. При перемножении цепочки матриц, вспомогательные задачи заключались в перемножении подцепочек $A_i A_{i+1} \dots A_k$ и $A_{k+1} A_{k+2} \dots A_j$. Это непересекающиеся подцепочки, которые не могут содержать общих матриц. При составлении расписания конвейера для определения самого быстрого пути через рабочее место $S_{i,j}$ осуществлялся поиск самого быстрого пути через рабочие места $S_{1,j-1}$ и $S_{2,j-1}$. Поскольку решение (т.е. самый быстрый путь через рабочее место $S_{i,j}$) содержит только одно из решений этих подзадач, данная подзадача автоматически независима от всех других подзадач, использующихся в этом решении.

Перекрытие вспомогательных задач

Вторая составляющая часть, наличие которой необходимо для применения динамического программирования, заключается в том, что пространство вспомогательных задач должно быть "небольшим" в том смысле, что в результате выполнения рекурсивного алгоритма одни и те же вспомогательные задачи решаются снова и снова, а новые вспомогательные задачи не возникают. Обычно полное количество различающихся вспомогательных задач выражается как полиномиальная функция от объема входных данных. Когда рекурсивный алгоритм снова и снова обращается к одной и той же задаче, говорят, что задача оптимизации содержит ***перекрывающиеся вспомогательные задачи*** (overlapping subproblems)[2]. В отличие от описанной выше ситуации, в задачах, решаемых с помощью алгоритма разбиения, на каждом шаге рекурсии обычно возникают полностью новые задачи. В алгоритмах динамического программирования обычно используется преимущество, заключающееся в наличии перекрывающихся вспомогательных задач. Это достигается путем однократного решения каждой вспомогательной задачи с последующим сохранением результатов в таблице, где при необходимости их можно будет найти за фиксированное время.

В разделе 15.1 было показано, что в рекурсивном решении задачи о составлении расписания работы конвейера осуществляется 2^{n-j} обращений к $f_i[j]$ для $j = 1, 2, \dots, n$. Путем использования таблицы, содержащей результаты решений вспомогательных задач, экспоненциальное время работы алгоритма удается свести к линейному.

[2]Может показаться странным, что вспомогательные задачи, использующиеся в динамическом программировании, являются и независимыми, и перекрывающимися. Эти требования могут показаться противоречащими друг другу, однако это не так, поскольку они относятся к разным понятиям. Две вспомогательные задачи одной и той же задачи независимы, если в них не используются общие ресурсы. Две вспомогательные задачи перекрываются, если на самом деле речь идет об одной и той же вспомогательной задаче, возникающей в разных задачах.

Чтобы подробнее проиллюстрировать свойство перекрывания вспомогательных задач, еще раз обратимся к задаче о перемножении цепочки матриц. Возвратимся к рис. 15.3. Обратите внимание, что процедура MATRIX_CHAIN_ORDER в процессе решения вспомогательных задач в более высоких строках постоянно обращается к решениям вспомогательных задач в более низких строках. Например, к элементу $m[3,4]$ осуществляется 4 обращения: при вычислении элементов $m[2,4]$, $m[1,4]$, $m[3,5]$ и $m[3,6]$. Если бы элемент $m[3,4]$ каждый раз приходилось каждый раз вычислять заново, а не просто находить в таблице, это привело бы к значительному увеличению времени работы. Чтобы продемонстрировать это, рассмотрим приведенную ниже (неэффективную) рекурсивную процедуру, в которой определяется величина $m[i,j]$, т.е. минимальное количество скалярных умножений, необходимых для вычисления произведения цепочки матриц $A_{i..j} = A_i A_{i+1} \ldots A_j$. Эта процедура основана непосредственно на рекуррентном соотношении (15.12):

```
RECURSIVE_MATRIX_CHAIN(p, i, j)
1  if i = j
2     then return 0
3  m[i, j] ← ∞
4  for k ← i to j − 1
5        do q ← RECURSIVE_MATRIX_CHAIN(p, i, k)
                  + RECURSIVE_MATRIX_CHAIN(p, k + 1, j)
                  + p_{i−1} p_k p_j
6           if q < m[i, j]
7              then m[i, j] ← q
8  return m[i, j]
```

На рис. 15.5 показано рекурсивное дерево, полученное в результате вызова процедуры RECURSIVE_MATRIX_CHAIN$(p, 1, 4)$. Каждый его узел обозначен величинами параметров i и j. Вычисления, которые производятся в части дерева, выделенной серым фоном, заменяются в процедуре MEMOIZED_MATRIX_CHAIN$(p, 1, 4)$. Обратите внимание, что некоторые пары значений встречаются много раз.

Можно показать, что время вычисления величины $m[1,n]$ в этой рекурсивной процедуре, как минимум, экспоненциально по n. Обозначим через $T(n)$ время, которое потребуется процедуре RECURSIVE_MATRIX_CHAIN для вычисления оптимального способа расстановки скобок в цепочке, состоящей из n матриц. Если считать, что выполнение каждой из строк 1–2 и 6–7 требует как минимум единичного интервала времени, то мы получим такое рекуррентное соотношение:

$$T(1) \geqslant 1,$$

$$T(n) \geqslant 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{при } n > 1.$$

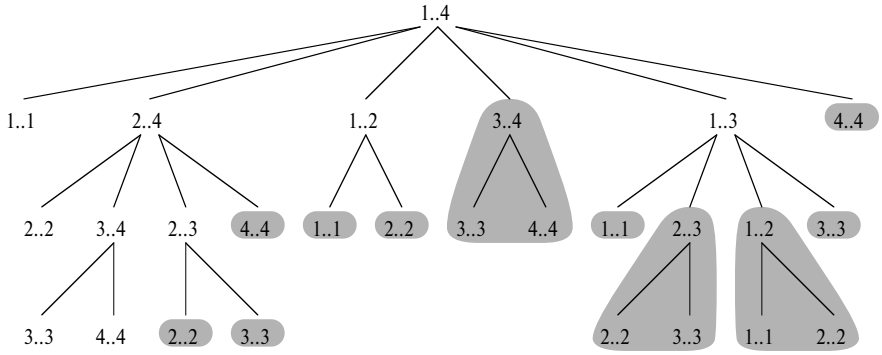


Рис. 15.5. Рекурсивное дерево, соответствующее вызову процедуры RECURSIVE_MATRIX_CHAIN$(p, 1, 4)$

Если заметить, что при $i = 1, 2, \ldots, n-1$ каждое слагаемое $T(i)$ один раз появляется как $T(k)$ и один раз как $T(n-k)$, и просуммировать $n-1$ единиц с той, которая стоит слева от суммы, то рекуррентное соотношение можно переписать в виде

$$T(n) \geqslant 2\sum_{i=1}^{n-1} T(i) + n. \tag{15.13}$$

Докажем с помощью метода подстановок, что $T(n) = \Omega(2^n)$. В частности, покажем, что для всех $n \geqslant 1$ справедливо соотношение $T(n) \geqslant 2^{n-1}$. Очевидно, что базисное соотношение индукции выполняется, поскольку $T(1) \geqslant 1 = 2^0$. Далее, по методу математической индукции для $n \geqslant 2$ имеем

$$T(n) \geqslant 2\sum_{i=1}^{n-1} 2^{i-1} + n = 2\sum_{i=0}^{n-2} 2^i + n = 2\left(2^{n-1} - 1\right) + n = (2^n - 2) + n \geqslant 2^{n-1}.$$

На этом доказательство завершено. Таким образом, полное количество вычислений, которые выполняются при вызове процедуры RECURSIVE_MATRIX_CHAIN$(p, 1, n)$, как минимум, экспоненциально зависит от n.

Сравним этот нисходящий рекурсивный алгоритм с восходящим алгоритмом, построенным по принципу динамического программирования. Последний работает эффективнее, поскольку в нем используется свойство перекрывающихся вспомогательных задач. Всего возникает $\Theta\left(n^2\right)$ различных вспомогательных задач, и в алгоритме, основанном на принципах динамического программирования, каждая из них решается ровно по одному разу. В рекурсивном же алгоритме каждую вспомогательную задачу необходимо решать всякий раз, когда она возникает в рекурсивном дереве. Каждый раз, когда рекурсивное дерево для обычного рекурсивного решения задачи несколько раз включает в себя одну и ту же вспомогательную

задачу, полезно проверить, нельзя ли воспользоваться динамическим программированием.

Построение оптимального решения

На практике зачастую мы сохраняем сведения о том, какой выбор делается в каждой вспомогательной программе, так что впоследствии нам не надо дополнительно решать задачу восстановления этой информации. В задаче о составлении расписания работы сборочного конвейера в элементе $l_i[j]$ сохранялся номер рабочего места, предшествующего рабочему месту $S_{i,j}$ на самом быстром пути через это рабочее место. Другая возможность — заполнить всю таблицу $f_i[j]$. В этом случае можно было бы определить, какое рабочее место предшествует рабочему месту $S_{1,j}$ на самом быстром пути, проходящем через рабочее место $S_{1,j}$, но на это потребовалось бы больше усилий. Если $f_1[j] = f_1[j-1] + a_{1,j}$, то на самом быстром пути через рабочее место $S_{1,j}$ ему предшествует рабочее место $S_{1,j-1}$. В противном случае должно выполняться соотношение $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$, и рабочему месту $S_{1,j}$ предшествует рабочее место $S_{2,j-1}$. В задаче о составлении расписания работы сборочного конвейера на восстановление информации о предшествующем рабочем месте необходимо время, равное $O(1)$ (на каждое рабочее место), даже если отсутствует таблица $l_i[j]$.

Однако в задаче о перемножении цепочки матриц в процессе воссоздания оптимального решения, благодаря таблице $s[i,j]$ удается значительно уменьшить объем работы. Предположим, что таблица $s[i,j]$ не поддерживается, а заполняется только таблица $m[i,j]$, в которой содержатся оптимальные стоимости вспомогательных задач. При выборе вспомогательной задачи, использующейся в оптимальном решении задачи о расстановке скобок в произведении $A_iA_{i+1}\ldots A_j$, всего имеется $j-i$ вариантов выбора, и это количество не является константой. Поэтому на восстановление сведений о том, какие вспомогательные задачи выбираются в процессе решения данной задачи, потребуется время, равное $\Theta(j-i) = \omega(1)$. Сохраняя в элементе $s[i,j]$ индекс, где выполняется разбиение произведения $A_iA_{i+1}\ldots A_j$, данные о каждом выборе можно восстановить за время $O(1)$.

Запоминание

Одна из вариаций динамического программирования часто обеспечивает ту же степень эффективности обычного подхода динамического программирования, но при соблюдении нисходящей стратегии. Идея заключается в том, чтобы ***запомнить*** (memoize[3]) обычный неэффективный рекурсивный алгоритм. Как и при

[3] В данном случае это не опечатка, по-английски этот термин пишется именно так. Как поясняет в примечании автор книги, смысл не просто в том, чтобы запомнить информацию (memorize), а в том, чтобы вскоре ею воспользоваться как памяткой (memo). — *Прим. ред.*

обычном динамическом программировании, поддерживается таблица, содержащая решения вспомогательных задач. Однако структура, управляющая заполнением таблицы, больше напоминает рекурсивный алгоритм.

В рекурсивном алгоритме с запоминанием поддерживается элемент таблицы для решения каждой вспомогательной задачи. Изначально в каждом таком элементе таблицы содержится специальное значение, указывающее на то, что данный элемент еще не заполнен. Если в процессе выполнения рекурсивного алгоритма вспомогательная задача встречается впервые, ее решение вычисляется и заносится в таблицу. Каждый раз, когда снова будет встречаться эта вспомогательная задача, в таблице находится и возвращается ее решение[4].

Ниже приведена версия процедуры RECURSIVE_MATRIX_CHAIN с запоминанием:

```
MEMOIZED_MATRIX_CHAIN(p)
1  n ← length[p] − 1
2  for i ← 1 to n
3        do for j ← i to n
4              do m[i, j] ← ∞
5  return LOOKUP_CHAIN(p, 1, n)

LOOKUP_CHAIN(p, i, j)
1  if m[i, j] < ∞
2     then return m[i, j]
3  if i = j
4     then m[i, j] ← 0
5     else  for k ← i to j − 1
6              do q ← LOOKUP_CHAIN(p, i, k)
                        + LOOKUP_CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
7                 if q < m[i, j]
8                   then m[i, j] ← q
9  return m[i, j]
```

В процедуре MEMOIZED_MATRIX_CHAIN, как и в процедуре MATRIX_CHAIN_ORDER, поддерживается таблица $m\,[1..n, 1..n]$, состоящая из вычисленных значений $m\,[i, j]$, которые представляют собой минимальное количество скалярных умножений, необходимых для вычисления матрицы $A_{i..j}$. Изначально каждый элемент таблицы содержит значение ∞, указывающее на то, что данный элемент еще должен быть заполнен. Если при вызове процедуры LOOKUP_CHAIN(p, i, j) вы-

[4]В этом подходе предполагается, что известен набор параметров всех возможных вспомогательных задач и установлено соответствие между ячейками таблицы и вспомогательными задачами. Другой подход состоит в том, чтобы организовать запоминание с помощью хеширования, в котором в качестве ключей выступают параметры вспомогательных задач.

полняется условие $m[i,j] < \infty$ (строка 1), эта процедура просто возвращает ранее вычисленную стоимость $m[i,j]$ (строка 2). В противном случае эта стоимость вычисляется так же, как и в процедуре RECURSIVE_MATRIX_CHAIN, сохраняется в элементе $m[i,j]$, после чего возвращается. (Удобство использования значения ∞ для незаполненных элементов таблицы объясняется тем, что это же значение используется для инициализации элементов $m[i,j]$ в строке 3 процедуры RECURSIVE_MATRIX_CHAIN.) Таким образом, процедура LOOKUP_CHAIN(p,i,j) всегда возвращает значение $m[i,j]$, но оно вычисляется лишь в том случае, если это первый вызов данной процедуры с параметрами i и j.

Рис. 15.5 иллюстрирует, насколько эффективнее процедура MEMOIZED_MATRIX_CHAIN расходует время по сравнению с процедурой RECURSIVE_MATRIX_CHAIN. Затененные поддеревья представляют значения, которые вычисляются только один раз. При возникновении повторной потребности в этих значениях осуществляется их поиск в хранилище.

Время работы процедуры MEMOIZED_MATRIX_CHAIN, как и время выполнения алгоритма MATRIX_CHAIN_ORDER, построенного по принципу динамического программирования, равно $O(n^3)$. Каждая ячейка таблицы (а всего их $\Theta(n^2)$) однократно инициализируется в строке 4 процедуры MEMOIZED_MATRIX_CHAIN. Все вызовы процедуры LOOKUP_CHAIN можно разбить на два типа:

1. вызовы, в которых справедливо соотношение $m[i,j] = \infty$ и выполняются строки 3–9;
2. вызовы, в которых выполняется неравенство $m[i,j] < \infty$ и просто происходит возврат в строке 2.

Всего вызовов первого типа насчитывается $\Theta(n^2)$, по одному на каждый элемент таблицы. Все вызовы второго типа осуществляются как рекурсивные обращения из вызовов первого типа. Всякий раз, когда в некотором вызове процедуры LOOKUP_CHAIN выполняются рекурсивные обращения, общее их количество равно $O(n)$. Поэтому в общем итоге производится $O(n^3)$ вызовов второго типа, причем на каждый из них расходуется время $O(1)$. На каждый вызов первого типа требуется время $O(n)$ плюс время, затраченное на рекурсивные обращения в данном вызове первого типа. Поэтому общее время равно $O(n^3)$. Таким образом, запоминание преобразует алгоритм, время работы которого равно $\Omega(2^n)$, в алгоритм со временем работы $O(n^3)$.

В итоге получается, что задачу об оптимальном способе перемножения цепочки матриц можно решить либо с помощью алгоритма с запоминанием, работающего в нисходящем направлении, либо с помощью динамического программирования в восходящем направлении. При этом в обоих случаях потребуется время, равное $O(n^3)$. Оба метода используют преимущество, возникающее благодаря перекрыванию вспомогательных задач. Всего возникает только $\Theta(n^2)$ различных вспомогательных задач, и в каждом из описанных выше методов решение каждой

из них вычисляется один раз. Если не применять запоминание, то время работы обычного рекурсивного алгоритма станет экспоненциальным, поскольку уже решенные задачи придется неоднократно решать заново.

В общем случае, если все вспомогательные задачи необходимо решить хотя бы по одному разу, восходящий алгоритм, построенный по принципу динамического программирования, обычно работает быстрее, чем нисходящий алгоритм с запоминанием. Причины — отсутствие непроизводительных затрат на рекурсию и их сокращение при поддержке таблицы. Более того, в некоторых задачах после определенных исследований удается сократить время доступа к таблице или занимаемый ею объем. Если же можно обойтись без решения некоторых вспомогательных задач, содержащихся в пространстве подзадач данной задачи, запоминание результатов решений обладает тем преимуществом, что решаются только необходимые вспомогательные задачи.

Упражнения

15.3-1. Как эффективнее определить оптимальное количество скалярных умножений в задаче о перемножении цепочки матриц: перечислить все способы расстановки скобок в произведении матриц и вычислить количество скалярных умножений в каждом из них или запустить процедуру RECURSIVE_MATRIX_CHAIN? Обоснуйте ответ.

15.3-2. Нарисуйте рекурсивное дерево для процедуры MERGE_SORT, описанной в разделе 2.3.1, работающей с массивом из 16 элементов. Объясните, почему для повышения производительности работы хорошего алгоритма разбиения, такого как MERGE_SORT, использование запоминания не будет эффективным.

15.3-3. Рассмотрим разновидность задачи о перемножении цепочки матриц, цель которой — так расставить скобки в последовательности матриц, чтобы количество скалярных умножений стало не минимальным, а максимальным. Проявляется ли в этой задаче оптимальная подструктура?

15.3-4. Опишите, какие перекрывающиеся вспомогательные задачи возникают при составлении расписания работы конвейера.

15.3-5. Согласно сделанному выше утверждению, в динамическом программировании сначала решаются вспомогательные задачи, а потом выбираются те из них, которые будут использованы в оптимальном решении задачи. Профессор утверждает, что не всегда необходимо решать все вспомогательные задачи, чтобы найти оптимальное решение. Он выдвигает предположение, что оптимальное решение задачи о перемножении цепочки матриц можно найти, всегда выбирая матрицу A_k, после которой следует разбивать произведение $A_i A_{i+1} \dots A_j$ (индекс k выбирается

так, чтобы минимизировать величину $p_{i-1}p_kp_j$) *перед* решением вспомогательных задач. Приведите пример задачи о перемножении цепочки матриц, в котором такой жадный подход приводит к решению, отличному от оптимального.

15.4 Самая длинная общая подпоследовательность

В биологических приложениях часто возникает необходимость сравнить ДНК двух (или большего количества) различных организмов. Стандартная ДНК состоит из последовательности молекул, которые называются ***основаниями*** (bases). К этим молекулам относятся: аденин (adenine), гуанин (guanine), цитозин (cytosine) и тимин (thymine). Если обозначить каждое из перечисленных оснований его начальной буквой латинского алфавита, то стандартную ДНК можно представить в виде строки, состоящей из конечного множества элементов $\{A, C, G, T\}$. (Определение строки см. в приложении В.) Например, ДНК одного организма может иметь вид $S_1 = \texttt{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$, а ДНК другого — $S_2 = \texttt{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$. Одна из целей сравнения двух ДНК состоит в том, чтобы выяснить степень их сходства. Это один из показателей того, насколько тесно связаны между собой два организма. Степень подобия можно определить многими различными способами. Например, можно сказать, что два кода ДНК подобны, если один из них является подстрокой другого. (Алгоритмы, с помощью которых решается эта задача, исследуются в главе 32.) В нашем примере ни S_1, ни S_2 не является подстрокой другой ДНК. В этом случае можно сказать, что две цепочки молекул подобны, если для преобразования одной из них в другую потребовались бы только небольшие изменения. (Такой подход рассматривается в задаче 15-3.) Еще один способ определения степени подобия последовательностей S_1 и S_2 заключается в поиске третьей последовательности S_3, основания которой имеются как в S_1, так и в S_2; при этом они следуют в одном и том же порядке, но не обязательно одно за другим. Чем длиннее последовательность S_3, тем более схожи последовательности S_1 и S_2. В рассматриваемом примере самая длинная последовательность S_3 имеет вид $\texttt{GTCGTCGGAAGCCGGCCGAA}$.

Последнее из упомянутых выше понятий подобия формализуется в виде задачи о самой длинной общей подпоследовательности. Подпоследовательность данной последовательности — это просто данная последовательность, из которой удалили ноль или больше элементов. Формально последовательность $Z = \langle z_1, z_2, \ldots, z_k \rangle$ является ***подпоследовательностью*** (subsequence) последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$, если существует строго возрастающая последовательность $\langle i_1, i_2, \ldots, i_k \rangle$ индексов X, такая что для всех $j = 1, 2, \ldots, k$ выполняется соотношение $x_{i_j} = z_j$. Например, $Z = \langle B, C, D, B \rangle$ — подпоследовательность

последовательности $X = \langle A, B, C, B, D, A, B\rangle$, причем соответствующая ей последовательность индексов имеет вид $\langle 2, 3, 5, 7\rangle$.

Говорят, что последовательность Z является ***общей подпоследовательностью*** (common subsequence) последовательностей X и Y, если Z является подпоследовательностью как X, так и Y. Например, если $X = \langle A, B, C, B, D, A, B\rangle$ и $Y = \langle B, D, C, A, B, A\rangle$, то последовательность $\langle B, C, A\rangle$ — общая подпоследовательность X и Y. Однако последовательность $\langle B, C, A\rangle$ — не *самая длинная* общая подпоследовательность X и Y, поскольку ее длина равна 3, а длина последовательности $\langle B, C, B, A\rangle$, которая тоже является общей подпоследовательностью X и Y, равна 4. Последовательность $\langle B, C, B, A\rangle$ — *самая длинная общая подпоследовательность* (longest common subsequence, LCS) последовательностей X и Y, как и последовательность $\langle B, D, A, B\rangle$, поскольку не существует общей подпоследовательности длиной 5 элементов или более.

В ***задаче о самой длинной общей подпоследовательности*** даются две последовательности $X = \langle x_1, x_2, \ldots, x_m\rangle$ и $Y = \langle y_1, y_2, \ldots, y_n\rangle$, и требуется найти общую подпоследовательность X и Y максимальной длины. В данном разделе показано, что эта задача эффективно решается методом динамического программирования.

Этап 1: характеристика самой длинной общей подпоследовательности

Решение задачи поиска самой длинной общей подпоследовательности “в лоб” заключается в том, чтобы пронумеровать все подпоследовательности последовательности X и проверить, являются ли они также подпоследовательностями Y, пытаясь отыскать при этом самую длинную из них. Каждая подпоследовательность последовательности X соответствует подмножеству индексов $\{1, 2, \ldots, m\}$ последовательности X. Всего имеется 2^m подпоследовательностей последовательности X, поэтому время работы алгоритма, основанного на этом подходе, будет экспоненциально зависеть от размера задачи, и для длинных последовательностей он становится непригодным.

Однако задача поиска самой длинной общей подпоследовательности обладает оптимальной подструктурой. Этот факт доказывается в сформулированной ниже теореме. Как мы увидим, естественно возникающие классы вспомогательных задач соответствуют парам “префиксов” двух входных последовательностей. Дадим точное определение этого понятия: i-м ***префиксом*** последовательности $X = \langle x_1, x_2, \ldots, x_m\rangle$ для $i = 1, 2, \ldots, m$ является подпоследовательность $X_i = \langle x_1, x_2, \ldots, x_i\rangle$. Например, если $X = \langle A, B, C, B, D, A, B\rangle$, то $X_4 = \langle A, B, C, B\rangle$, а X_0 — пустая последовательность.

Теорема 15.1 (Оптимальная подструктура LCS). Пусть имеются последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$, а $Z = \langle z_1, z_2, \ldots, z_k \rangle$ — их самая длинная общая подпоследовательность.

1. Если $x_m = y_n$, то $z_k = x_m = y_n$ и Z_{k-1} — LCS последовательностей X_{m-1} и Y_{n-1}.
2. Если $x_m \neq y_n$, то из $z_k \neq x_m$ следует, что Z — самая длинная общая подпоследовательность последовательностей X_{m-1} и Y.
3. Если $x_m \neq y_n$, то из $z_k \neq y_n$ следует, что Z — самая длинная общая подпоследовательность последовательностей X и Y_{n-1}.

Доказательство. (1) Если бы выполнялось соотношение $z_k \neq x_m$, то к последовательности Z можно было бы добавить элемент $x_m = y_n$, в результате чего получилась бы общая подпоследовательность последовательностей X и Y длиной $k + 1$, а это противоречит предположению о том, что Z — *самая длинная* общая подпоследовательность последовательностей X и Y. Таким образом, должно выполняться соотношение $z_k = x_m = y_n$. Таким образом, префикс Z_{k-1} — общая подпоследовательность последовательностей X_{m-1} и Y_{n-1} длиной $k - 1$. Нужно показать, что это самая длинная общая подпоследовательность. Проведем доказательство методом от противного. Предположим, что имеется общая подпоследовательность W последовательностей X_{m-1} и Y_{n-1}, длина которой превышает $k - 1$. Добавив к W элемент $x_m = y_n$, получим общую подпоследовательность последовательностей X и Y, длина которой превышает k, что приводит к противоречию.

(2) Если $z_k \neq x_m$, то Z — общая подпоследовательность последовательностей X_{m-1} и Y. Если бы существовала общая подпоследовательность W последовательностей X_{m-1} и Y, длина которой превышает k, то она была бы также общей подпоследовательностью последовательностей $X_m \equiv X$ и Y, что противоречит предположению о том, что Z — самая длинная общая подпоследовательность последовательностей X и Y.

(3) Доказательство этого случая аналогично доказательству случая (2) с точностью до замены элементов последовательности X соответствующими элементами последовательности Y. ■

И теоремы 15.1 видно, что самая длинная общая подпоследовательность двух последовательностей содержит в себе самую длинную общую подпоследовательность их префиксов. Таким образом, задача о самой длинной общей подпоследовательности обладает оптимальной подструктурой. В рекурсивном решении этой задачи также возникают перекрывающиеся вспомогательные задачи, но об этом речь пойдет в следующем разделе.

Этап 2: рекурсивное решение

Из теоремы 15.1 следует, что при нахождении самой длинной общей подпоследовательности последовательностей $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$ возникает одна или две вспомогательные задачи. Если $x_m = y_n$, необходимо найти самую длинную общую подпоследовательность последовательностей X_{m-1} и Y_{m-1}. Добавив к ней элемент $x_m = y_n$, получим LCS последовательностей X и Y. Если $x_m \neq y_n$, необходимо решить две вспомогательных задачи: найти самые LSC последовательностей X_{m-1} и Y, а также последовательностей X и Y_{n-1}. Какая из этих двух подпоследовательностей окажется длиннее, та и будет самой длинной общей подпоследовательностью последовательностей X и Y.

В задаче поиска самой длинной общей подпоследовательности легко увидеть проявление свойства перекрывающихся вспомогательных задач. Чтобы найти LCS последовательностей X и Y, может понадобиться найти LCS последовательностей X_{m-1} и Y, а также LCS X и Y_{n-1}. Однако в каждой из этих задач возникает задача о поиске LCS последовательностей X_{m-1} и Y_{n-1}. Общая вспомогательная задача возникает и во многих других подзадачах.

Как и в задаче о перемножении цепочки матриц, в рекурсивном решении задачи о самой длинной общей подпоследовательности устанавливается рекуррентное соотношение для значений, характеризующих оптимальное решение. Обозначим через $c[i, j]$ длину самой длинной общей подпоследовательности последовательностей X_i и Y_j. Если $i = 0$ или $j = 0$, длина одной из этих последовательностей равна нулю, поэтому самая длинная их общая подпоследовательность имеет нулевую длину. Оптимальная вспомогательная подструктура задачи о самой длинной общей подпоследовательности определяется формулой:

$$c[i, j] = \begin{cases} 0 & \text{при } i = 0 \text{ или } j = 0, \\ c[i-1, j-1] + 1 & \text{при } i, j > 0 \text{ и } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{при } i, j > 0 \text{ и } x_i \neq y_j. \end{cases} \tag{15.14}$$

Обратите внимание, что в этой рекурсивной формулировке условия задачи ограничивают круг рассматриваемых вспомогательных задач. Если $x_i = y_j$, можно и нужно рассматривать вспомогательную задачу поиска самой длинной общей подпоследовательности последовательностей X_{i-1} и Y_{j-1}. В противном случае рассматриваются две вспомогательные задачи по поиску LCS последовательностей X_i и Y_{j-1}, а также последовательностей X_{i-1} и Y_j. В рассмотренных ранее алгоритмах, основанных на принципах динамического программирования (для задач о составлении расписания работы сборочного конвейера и о перемножении цепочки матриц), выбор вспомогательных задач не зависел от условий задачи. Задача о поиске LCS не единственная, в которой вид возникающих вспомогательных задач определяется условиями задачи. В качестве другого подобного примера можно привести задачу о расстоянии редактирования (см. задачу 15-3).

Этап 3: вычисление длины самой длинной общей подпоследовательности

На основе уравнения (15.14) легко написать рекурсивный алгоритм с экспоненциальным временем работы, предназначенный для вычисления длины LCS двух последовательностей. Однако благодаря наличию всего лишь $\Theta(mn)$ различных вспомогательных задач можно воспользоваться динамическим программированием для вычисления решения в восходящем направлении.

В процедуре LCS_LENGTH в роли входных данных выступают две последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Величины $c[i, j]$ хранятся в таблице $c[0..m, 0..n]$, элементы которой вычисляются по строкам (т.е. сначала слева направо заполняется первая строка, затем — вторая и т.д.). В процедуре также поддерживается таблица $b[1..m, 1..n]$, благодаря которой упрощается процесс построения оптимального решения. Наглядный смысл элементов $b[i, j]$ состоит в том, что каждый из них указывает на элемент таблицы, соответствующий оптимальному решению вспомогательной задачи, выбранной при вычислении элемента $c[i, j]$. Процедура возвращает таблицы b и c; в элементе $c[m, n]$ хранится длина LCS последовательностей X и Y.

```
LCS_LENGTH(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4       do c[i, 0] ← 0
 5  for j ← 0 to n
 6       do c[0, j] ← 0
 7  for i ← 1 to m
 8       do for j ← 1 to n
 9            do if x_i = y_j
10                 then c[i, j] ← c[i − 1, j − 1] + 1
11                      b[i, j] ← "↖"
12                 else if c[i − 1, j] ≥ c[i, j − 1]
13                      then c[i, j] ← c[i − 1, j]
14                           b[i, j] ← "↑"
15                      else c[i, j] ← c[i, j − 1]
16                           b[i, j] ← "←"
17  return c и b
```

На рис. 15.6 показаны таблицы, полученные в результате выполнения процедуры LCS_LENGTH с входными последовательностями $X = \langle A, B, C, B, D, A, B \rangle$ и $Y = \langle B, D, C, A, B, A \rangle$. Время выполнения процедуры равно $O(mn)$, поскольку на вычисление каждого элемента таблицы требуется время, равное $O(1)$. Квадрат,

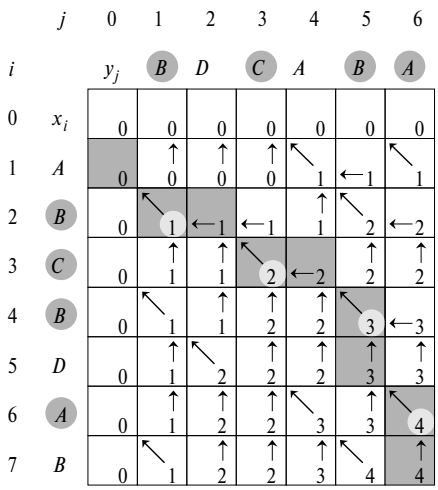


Рис. 15.6. Таблицы c и b, которые возвращаются процедурой LCS_LENGTH для входных последовательностей $X = \langle A, B, C, B, D, A, B\rangle$ и $Y = \langle B, D, C, A, B, A\rangle$

который находится на пересечении строки i и столбца j, содержит значение $c[i, j]$ и соответствующую стрелку, выступающую в роли значения $b[i, j]$. Элемент 4, который является значением содержащегося в правом нижнем углу элемента $c[7, 6]$, — длина самой длинной общей подпоследовательности последовательностей X и Y (в данном случае это $\langle B, C, B, A\rangle$). При $i, j > 0$ элемент $c[i, j]$ зависит только от того, выполняется ли соотношение $x_i = y_j$, и от значений элементов $c[i - 1, j]$, $c[i, j - 1]$ и $c[i - 1, j - 1]$, которые вычисляются перед значением $c[i, j]$. Чтобы восстановить элементы, из которых состоит самая длинная общая подпоследовательность, проследуем по стрелочкам $b[i, j]$, ведущим из правого нижнего угла. Полученный путь обозначен затенением серого цвета. Каждый элемент "↖" на этом пути соответствует (выделенному цветом) элементу, для которого $x_i = y_j$ является элементом самой длинной общей подпоследовательности.

Этап 4: построение самой длинной общей подпоследовательности

С помощью таблицы b, которая возвращается процедурой LCS_LENGTH, можно быстро сконструировать самую длинную общую подпоследовательность последовательностей $X = \langle x_1, x_2, \ldots, x_m\rangle$ и $Y = \langle y_1, y_2, \ldots, y_n\rangle$. Мы просто начинаем с элемента $b[m, n]$ и проходим таблицу по стрелкам. Если значением элемента $b[i, j]$ является "↖", то элемент $x_i = y_j$ принадлежит самой длинной

общей подпоследовательности. Элементы LCS, восстановленные этим методом, следуют в обратном порядке. Приведенная ниже рекурсивная процедура выводит элементы самой длинной общей подпоследовательности последовательностей X и Y в прямом порядке. Начальный вызов этой процедуры выглядит как PRINT_LCS$(b, X, length\,[X]\,, length\,[Y])$:

```
PRINT_LCS(b, X, i, j)
1  if i = 0 или j = 0
2     then return
3  if b[i, j] = "↖"
4     then PRINT_LCS(b, X, i − 1, j − 1)
5          print x_i
6  elseif b[i, j] = "↑"
7     then PRINT_LCS(b, X, i − 1, j)
8  else PRINT_LCS(b, X, i, j − 1)
```

Для таблицы b, изображенной на рис. 15.6, эта процедура выводит последовательность "$BCBA$". Время работы процедуры равно $O\,(m+n)$, поскольку хотя бы один из индексов i или j уменьшается на каждой стадии рекурсии.

Улучшение кода

После разработки алгоритма часто оказывается, что можно улучшить время его работы или объем требуемой им памяти. Особенно это справедливо в отношении незамысловатых алгоритмов, основанных на принципах динамического программирования. Некоторые изменения могут упростить код и уменьшить постоянный множитель, но не приводят к повышению производительности в асимптотическом пределе. Другие же могут вызвать существенное асимптотическую экономию времени и пространства.

Например, можно обойтись без таблицы b. Каждый элемент $c\,[i,j]$ зависит только от трех других элементов этой же таблицы: $c\,[i-1,j-1]$, $c\,[i-1,j]$ и $c\,[i,j-1]$. Для данного элемента $c\,[i,j]$ в течение времени $O\,(1)$ можно определить, какое из этих трех значений было использовано для вычисления $c\,[i,j]$, не прибегая к таблице b. Таким образом, самую длинную общую подпоследовательность можно восстановить в течение времени $O\,(m+n)$. Для этого понадобится процедура, подобная процедуре PRINT_LCS (такую процедуру предлагается составить в упражнении 15.4-2). Несмотря на то, что в этом методе экономится объем памяти, равный $\Theta\,(mn)$, асимптотически количество памяти, необходимой для вычисления самой длинной общей подпоследовательности, не изменяется. Это объясняется тем, что таблица c все равно требует $\Theta\,(mn)$ памяти.

Однако можно уменьшить объем памяти, необходимой для работы процедуры PRINT_LCS, поскольку одновременно нужны лишь две строки этой таблицы:

вычисляемая и предыдущая. (Фактически для вычисления длины самой длинной общей подпоследовательности можно обойтись пространством, лишь немного превышающим объем, необходимый для одной строки матрицы c — см. упражнение 15.4-4.) Это усовершенствование работает лишь в том случае, когда нужно знать только длину самой длинной общей подпоследовательности. Если же необходимо воссоздать элементы этой подпоследовательности, такая "урезанная" таблица содержит недостаточно информации для того, чтобы проследить обратные шаги в течение времени $O(m+n)$.

Упражнения

15.4-1. Определите самую длинную общую подпоследовательность последовательностей $\langle 1,0,0,1,0,1,0,1\rangle$ и $\langle 0,1,0,1,1,0,1,1,0\rangle$.

15.4-2. Покажите, как в течение времени $O(m+n)$ воссоздать самую длинную общую подпоследовательность, не используя при этом таблицу b, если имеется таблица c и исходные последовательности $X=\langle x_1,x_2,\ldots,x_m\rangle$ и $Y=\langle y_1,y_2,\ldots,y_n\rangle$.

15.4-3. Приведите версию процедуры LCS_LENGTH с запоминанием, время работы которой равно $O(mn)$.

15.4-4. Покажите, как вычислить длину самой длинной общей подпоследовательности, используя при этом всего $2\cdot\min(m,n)$ элементов таблицы c плюс $O(1)$ дополнительной памяти. Затем покажите, как это же можно сделать с помощью $\min(m,n)$ элементов таблицы и $O(1)$ дополнительной памяти.

15.4-5. Приведите алгоритм, предназначенный для вычисления самой длинной монотонно неубывающей подпоследовательности данной последовательности, состоящей из n чисел. Время работы алгоритма должно быть равно $O\left(n^2\right)$.

⋆ 15.4-6. Приведите алгоритм, предназначенный для вычисления самой длинной монотонно неубывающей подпоследовательности данной последовательности, состоящей из n чисел. Время работы алгоритма должно быть равно $O(n\lg n)$. (*Указание*: обратите внимание, что последний элемент кандидата в искомую подпоследовательность по величине не меньше последнего элемента кандидата длиной $i-1$.)

15.5 Оптимальные бинарные деревья поиска

Предположим, что разрабатывается программа, предназначенная для перевода текстов с русского языка на украинский. Для каждого русского слова необходимо найти украинский эквивалент. Один из возможных путей поиска — построение

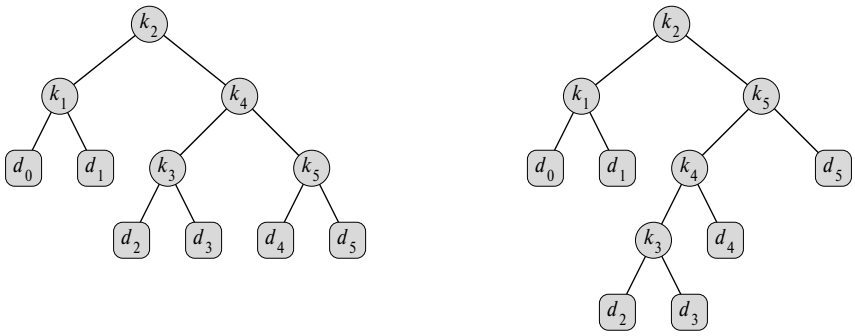
бинарного дерева поиска с n русскими словами, выступающими в роли ключей, и украинскими эквивалентами, играющими роль сопутствующих данных. Поскольку поиск с помощью этого дерева будет производиться для каждого отдельного слова из текста, полное затраченное на него время должно быть как можно меньше. С помощью красно-черного дерева или любого другого сбалансированного бинарного дерева поиска можно добиться того, что время каждого отдельного поиска будет равным $O(\lg n)$. Однако слова встречаются с разной частотой, и может получиться так, что какое-нибудь часто употребляемое слово (например, предлог или союз) находится далеко от корня, а такое редкое слово, как "контрвстреча", — возле корня. Такой способ организации привел бы к замедлению перевода, поскольку количество узлов, просмотренных в процессе поиска ключа в бинарном дереве, равно увеличенной на единицу глубине узла, содержащего данный ключ. Нужно сделать так, чтобы слова, которые встречаются в тексте часто, были размещены поближе к корню. Кроме того, в исходном тексте могут встречаться слова, для которых перевод отсутствует. Таких слов вообще не должно быть в бинарном дереве поиска. Как организовать бинарное дерево поиска, чтобы свести к минимуму количество посещенных в процессе поиска узлов, если известно, с какой частотой встречаются слова?

Необходимая нам конструкция известна как ***оптимальное бинарное дерево поиска*** (optimal binary search tree). Приведем формальное описание задачи. Имеется заданная последовательность $K = \langle k_1, k_2, \ldots, k_n \rangle$, состоящая из n различных ключей, которые расположены в отсортированном порядке (так что $k_1 < k_2 < \cdots < k_n$). Из этих ключей нужно составить бинарное дерево поиска. Для каждого ключа k_i задана вероятность p_i поиска этого ключа. Кроме того, может выполняться поиск значений, отсутствующих в последовательности K, поэтому следует предусмотреть $n+1$ фиктивных ключей $\langle d_0, d_1, \ldots, d_n \rangle$, представляющих эти значения. В частности, d_0 представляет все значения, меньшие k_1, а d_n — все значения, превышающие k_n. Фиктивный ключ d_i ($i = 1, 2, \ldots, n-1$) представляет все значения, которые находятся между k_i и k_{i+1}. Для каждого фиктивного ключа d_i задана соответствующая ей вероятность q_i. На рис. 15.7 показаны два бинарных дерева поиска для множества, состоящего из $n = 5$ ключей. Каждый ключ k_i представлен внутренним узлом, а каждый фиктивный ключ d_i является листом. Поиск может быть либо успешным (найден какой-то ключ k_i), либо неудачным (возвращается какой-то фиктивный ключ d_i), поэтому справедливо соотношение

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1. \tag{15.15}$$

Вероятности, соответствующие внутренним узлам p_i и листьям q_i, приведены в табл. 15.1.

Поскольку вероятность поиска каждого обычного и фиктивного ключа считается известной, можно определить математическое ожидание стоимости поиска



а) б)

Рис. 15.7. Два бинарных дерева поиска для множества из 5 элементов

Таблица 15.1. Вероятности поиска ключей в узлах бинарного дерева

i	0	1	2	3	4	5
p_i		0,15	0,10	0,05	0,10	0,20
q_i	0,05	0,10	0,05	0,05	0,05	0,10

по заданному бинарному дереву поиска T. Предположим, что фактическая стоимость поиска определяется количеством проверенных узлов, т.е. увеличенной на единицу глубиной узла на дереве T, в котором находится искомый ключ. Тогда математическое ожидание стоимости поиска в дереве T равно

$$\begin{aligned} \mathrm{E}\left[\text{Стоимость поиска в } T\right] &= \sum_{i=1}^{n} \left(depth_T\left(k_i\right)+1\right)\cdot p_i + \\ &+ \sum_{i=0}^{n} \left(depth_T\left(d_i\right)+1\right)\cdot q_i = \\ &= 1 + \sum_{i=1}^{n} depth_T\left(k_i\right)\cdot p_i + \\ &+ \sum_{i=0}^{n} depth_T\left(d_i\right)\cdot q_i, \end{aligned} \tag{15.16}$$

где величина $depth_T\left(\right)$ обозначает глубину узла в дереве T. Последнее равенство следует из уравнения (15.15). В табл. 15.2 вычисляется математическое ожидание стоимости поиска для бинарного дерева, изображенного на рис. 15.7*а*.

Наша цель — построить для данного набора вероятностей бинарное дерево поиска, математическое ожидание стоимости поиска для которого будет

Таблица 15.2. Вычисление математического ожидания стоимости поиска

Узел	Глубина	Вероятность	Вклад
k_1	1	0,15	0,30
k_2	0	0,10	0,10
k_3	2	0,05	0,15
k_4	1	0,10	0,20
k_5	2	0,20	0,60
d_0	2	0,05	0,15
d_1	2	0,10	0,30
d_2	3	0,05	0,20
d_3	3	0,05	0,20
d_4	3	0,05	0,20
d_5	3	0,10	0,40
Всего			2,80

минимальным. Такое дерево называется ***оптимальным бинарным деревом поиска***. На рис. 15.7*б* показано оптимальное бинарное дерево поиска для вероятностей, заданных в табл. 15.1. Математическое ожидание поиска в этом дереве равно 2.75. Этот пример демонстрирует, что оптимальное бинарное дерево поиска — это не обязательно дерево минимальной высоты. Кроме того, в оптимальном дереве ключ, которому соответствует максимальная вероятность, не всегда находится в корне. В данном случае вероятность имеет самую большую величину для ключа k_5, хотя в корне оптимального бинарного дерева расположен ключ k_2. (Минимальная величина математического ожидания для всевозможных бинарных деревьев поиска, в корне которых находится ключ k_5, равна 2.85.)

Как и в задаче о перемножении цепочки матриц, последовательный перебор всех возможных деревьев в данном случае оказывается неэффективным. Чтобы сконструировать бинарное дерево поиска, можно обозначить ключами $k_1, k_2, \ldots, k_n$ узлы бинарного дерева с n узлами, а затем добавить листья для фиктивных ключей. В задаче 12-4 было показано, что количество бинарных деревьев с n узлами равно $\Omega\left(4^n/n^{3/2}\right)$, так что количество бинарных деревьев, которые надо проверять при полном переборе, растет экспоненциально с ростом n. Не удивительно, что эта задача будет решаться методом динамического программирования.

Этап 1: структура оптимального бинарного дерева поиска

Чтобы охарактеризовать оптимальную подструктуру оптимального бинарного дерева поиска, исследуем его поддеревья. Рассмотрим произвольное поддерево бинарного дерева поиска. Оно должно содержать ключи, которые составляют непрерывный интервал $k_i, \ldots, k_j$ для некоторых $1 \leqslant i \leqslant j \leqslant n$. Кроме того, такое поддерево должно также содержать в качестве листьев фиктивные ключи $d_{i-1}, \ldots, d_j$.

Теперь можно сформулировать оптимальную подструктуру: если в состав оптимального бинарного дерева поиска T входит поддерево T', содержащее ключи $k_i, \ldots, k_j$, то это поддерево тоже должно быть оптимальным для вспомогательной подзадачи с ключами $k_i, \ldots, k_j$ и фиктивными ключами $d_{i-1}, \ldots, d_j$. Для доказательства этого утверждения применяется обычный метод "вырезания и вставки". Если бы существовало поддерево T'', математическое ожидание поиска в котором ниже, чем математическое ожидание поиска в поддереве T', то из дерева T можно было бы вырезать поддерево T' и подставить вместо него поддерево T''. В результате получилось бы дерево, математическое ожидание времени поиска в котором оказалось бы меньше, что противоречит оптимальности дерева T.

Покажем с помощью описанной выше оптимальной подструктуры, что оптимальное решение задачи можно воссоздать из оптимальных решений вспомогательных задач. Если имеется поддерево, содержащее ключи $k_i, \ldots, k_j$, то один из этих ключей, скажем, k_r ($i \leqslant r \leqslant j$) будет корнем этого оптимального поддерева. Поддерево, которое находится слева от корня k_r, будет содержать ключи $k_i, \ldots, k_{r-1}$ (и фиктивные ключи $d_{i-1}, \ldots d_{r-1}$), а правое поддерево — ключи $k_{r+1}, \ldots, k_j$ (и фиктивные ключи $d_r, \ldots, d_j$). Как только будут проверены все ключи k_r (где $i \leqslant r \leqslant j$), которые являются кандидатами на роль корня, и найдем оптимальные бинарные деревья поиска, содержащие элементы $k_i, \ldots, k_{r-1}$ и $k_{r+1}, \ldots, k_j$, мы гарантированно построим оптимальное бинарное дерево поиска.

Стоит сделать одно замечание по поводу "пустых" поддеревьев. Предположим, что в поддереве с ключами $k_i, \ldots, k_j$ в качестве корня выбран ключ k_i. Согласно приведенным выше рассуждениям, поддерево, которое находится слева от корня k_i, содержит ключи $k_i, \ldots, k_{i-1}$. Естественно интерпретировать эту последовательность как такую, в которой не содержится ни одного ключа. Однако следует иметь в виду, что поддеревья содержат помимо реальных и фиктивные ключи. Примем соглашение, согласно которому поддерево, состоящее из ключей $k_i, \ldots, k_{i-1}$, не содержит обычных ключей, но содержит один фиктивный ключ d_{i-1}. Аналогично, если в качестве корня выбран ключ k_j, то правое поддерево не содержит обычных ключей, но содержит один фиктивный ключ d_j.

Этап 2: рекурсивное решение

Теперь все готово для рекурсивного определения оптимального решения. В качестве вспомогательной задачи выберем задачу поиска оптимального бинарного дерева поиска, содержащего ключи $k_i, \ldots, k_j$, где $i \geqslant 1$, $j \leqslant n$ и $j \geqslant i-1$ (если $j = i-1$, то фактических ключей не существует, имеется только фиктивный ключ d_{i-1}). Определим величину $e[i, j]$ как математическое ожидание стоимости поиска в оптимальном бинарном дереве поиска с ключами $k_i, \ldots, k_j$. В конечном итоге нужно вычислить величину $e[1, n]$.

Если $j = i-1$, то все просто. В этом случае имеется всего один фиктивный ключ d_{i-1}, и математическое ожидание стоимости поиска равно $e[i, i-1] = q_{i-1}$.

Если $j \geqslant i$, то среди ключей $k_i, \ldots, k_j$ нужно выбрать корень k_r, а потом из ключей $k_i, \ldots, k_{r-1}$ составить левое оптимальное бинарное дерево поиска, а из ключей $k_{r+1}, \ldots, k_j$ — правое оптимальное бинарное дерево поиска. Что происходит с математическим ожиданием стоимости поиска в поддереве, когда оно становится поддеревом какого-то узла? Глубина каждого узла в поддереве возрастает на единицу. Согласно уравнению (15.16), математическое ожидание стоимости поиска в этом поддереве возрастает на величину суммы по всем вероятностям поддерева. Обозначим эту сумму вероятностей, вычисленную для поддерева с ключами $k_i, \ldots, k_j$, так:

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l. \tag{15.17}$$

Таким образом, если k_r — корень оптимального поддерева, содержащего ключи $k_i, \ldots, k_j$, то выполняется соотношение

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j)).$$

Заметив, что

$$w(i, j) = w(i, r-1) + p_r + w(r+1, j),$$

выражение для величины $e[i, j]$ можно переписать так:

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j). \tag{15.18}$$

Рекурсивное соотношение (15.18) предполагает, что нам известно, какой узел k_r используется в качестве корня. На эту роль выбирается ключ, который приводит к минимальному значению математического ожидания стоимости поиска. С учетом этого получаем окончательную рекурсивную формулу:

$$e[i, j] = \begin{cases} q_{i-1} & \text{при } j = i-1, \\ \min\limits_{i \leqslant r \leqslant j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{при } i \leqslant j. \end{cases} \tag{15.19}$$

Величины $e[i, j]$ — это математическое ожидание стоимостей поиска в оптимальных бинарных деревьях поиска. Чтобы было легче следить за структурой оптимального бинарного дерева поиска, обозначим через $root[i, j]$ (где $1 \leqslant i \leqslant j \leqslant n$) индекс r узла k_r, который является корнем оптимального бинарного дерева поиска, содержащего ключи $k_i, \ldots, k_j$. Скоро мы узнаем, как вычисляются величины $root[i, j]$, а способ восстановления из этих величин оптимального бинарного дерева поиска оставим до того момента, когда придет время выполнить упражнение 15.5-1.

Этап 3: вычисление математического ожидания стоимости поиска в оптимальном бинарном дереве поиска

На данном этапе некоторые читатели, возможно, заметили некоторое сходство между характеристиками задач об оптимальных бинарных деревьях поиска и о перемножении цепочки матриц. Во вспомогательных задачах обеих задач индексы элементов изменяются последовательно. Прямая рекурсивная реализация уравнения (15.19) может оказаться такой же неэффективной, как и прямая рекурсивная реализация алгоритма в задаче о перемножении цепочки матриц. Вместо этого будем сохранять значения $e[i, j]$ в таблице $e[1..n+1, 0..n]$. Первый индекс должен пробегать не n, а $n+1$ значений. Это объясняется тем, что для получения поддерева, в который входит только фиктивный ключ d_n, понадобится вычислить и сохранить значение $e[n+1, n]$. Второй индекс должен начинаться с нуля, поскольку для получения поддерева, содержащего лишь фиктивный ключ d_0, нужно вычислить и сохранить значение $e[1, 0]$. Мы будем использовать только те элементы $e[i, j]$, для которых $j \geqslant i - 1$. Кроме того, будет использована таблица $root[i, j]$, в которую будут заноситься корни поддеревьев, содержащих ключи $k_i, \ldots, k_j$. В этой таблице задействованы только те записи, для которых $1 \leqslant i \leqslant j \leqslant n$.

Для повышения эффективности понадобится еще одна таблица. Вместо того чтобы каждый раз при вычислении $e[i, j]$ вычислять значения $w(i, j)$ "с нуля", для чего потребуется $\Theta(j - i)$ операций сложения, будем сохранять эти значения в таблице $w[1..n+1, 0..n]$. В базовом случае вычисляются величины $w[i, i-1] = q_{i-1}$ для $1 \leqslant i \leqslant n+1$. Для $j \geqslant i$ вычисляются величины

$$w[i, j] = w[i, j-1] + p_j + q_j. \tag{15.20}$$

Таким образом, каждое из $\Theta\left(n^2\right)$ значений матрицы $w[i, j]$ можно вычислить за время $\Theta(1)$.

Ниже приведен псевдокод, который принимает в качестве входных данных вероятности $p_1, \ldots, p_n$ и $q_0, \ldots, q_n$ и размер n и возвращает таблицы e и $root$.

```
OPTIMAL_BST(p, q, n)
 1  for i ← 1 to n + 1
 2       do e[i, i − 1] ← q_{i−1}
 3          w[i, i − 1] ← q_{i−1}
 4  for l ← 1 to n
 5       do for i ← 1 to n − l + 1
 6             do j ← i + l − 1
 7                e[i, j] ← ∞
 8                w[i, j] ← w[i, j − 1] + p_j + q_j
 9                for r ← i to j
10                     do t ← e[i, r − 1] + e[r + 1, j] + w[i, j]
11                        if t < e[i, j]
12                          then e[i, j] ← t
13                               root[i, j] ← r
14  return e и root
```

Благодаря приведенному выше описанию и аналогии с процедурой MATRIX_CHAIN_ORDER, описанной в разделе 15.2, работа представленной выше процедуры должна быть понятной. Цикл **for** в строках 1–3 инициализирует значения $e[i, i-1]$ и $w[i, i-1]$. Затем в цикле **for** в строках 4–13 с помощью рекуррентных соотношений (15.19) и (15.20) вычисляются элементы матриц $e[i, j]$ и $w[i, j]$ для всех индексов $1 \leqslant i \leqslant j \leqslant n$. В первой итерации, когда $l = 1$, в этом цикле вычисляются элементы $e[i, i]$ и $w[i, i]$ для $i = 1, 2, \ldots, n$. Во второй итерации, когда $l = 2$, вычисляются элементы $e[i, i+1]$ и $w[i, i+1]$ для $i = 1, 2, \ldots, n-1$ и т.д. Во внутреннем цикле **for** (строки 9–13) каждый индекс r апробируется на роль индекса корневого элемента k_r оптимального бинарного дерева поиска с ключами $k_i, \ldots, k_j$. В этом цикле элементу $root[i, j]$ присваивается то значение индекса r, которое подходит лучше всего.

На рис. 15.8 показаны таблицы $e[i, j]$, $w[i, j]$ и $root[i, j]$, вычисленные с помощью процедуры OPTIMAL_BST для распределения ключей из табл. 15.1. Как и в примере с перемножением цепочки матриц, таблицы повернуты так, чтобы диагонали располагались горизонтально. В процедуре OPTIMAL_BST строки вычисляются снизу вверх, а в каждой строке заполнение элементов производится слева направо.

Время выполнения процедуры OPTIMAL_BST, как и время выполнения процедуры MATRIX_CHAIN_ORDER, равно $\Theta\left(n^3\right)$. Легко увидеть, что время работы составляет $O\left(n^3\right)$, поскольку циклы **for** этой процедуры трижды вложены друг в друга, и индекс каждого цикла принимает не более n значений. Далее, индексы циклов в процедуре OPTIMAL_BST изменяются не в тех же пределах, что и индексы циклов в процедуре MATRIX_CHAIN_ORDER, но во всех направлениях они принимают по крайней мере одно значение. Таким образом, процедура OPTIMAL_BST, как и процедура MATRIX_CHAIN_ORDER, выполняется в течение времени $\Omega\left(n^3\right)$.

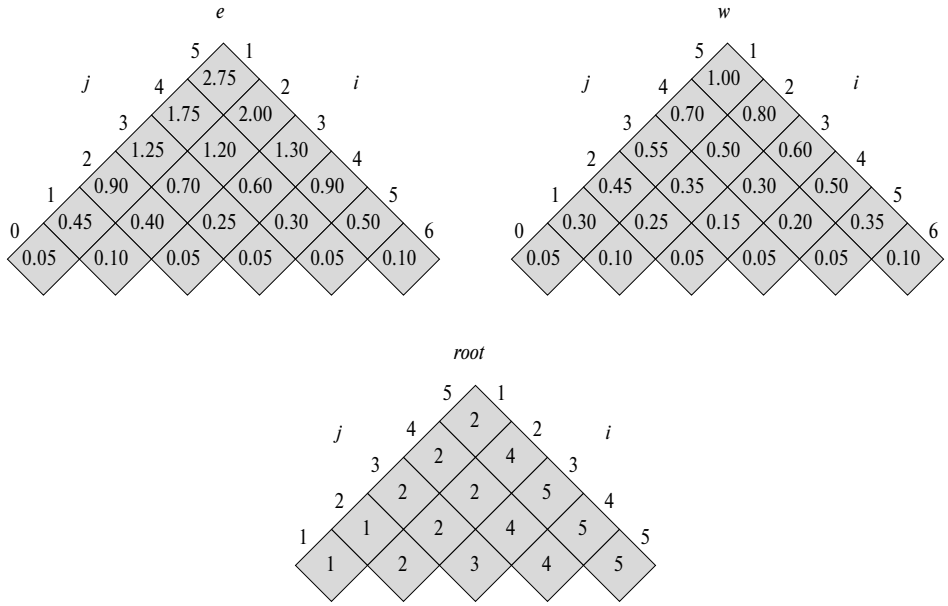


Рис. 15.8. Таблицы $e[i, j]$, $w[i, j]$ и $root[i, j]$, вычисленные процедурой OPTIMAL_BST для распределения ключей из табл. 15.1

Упражнения

15.5-1. Напишите псевдокод для процедуры CONSTRUCT_OPTIMAL_BST($root$), которая по заданной таблице $root$ выдает структуру оптимального бинарного дерева поиска. Для примера, приведенного на рис. 15.8, процедура должна выводить структуру, соответствующую оптимальному бинарному дереву поиска, показанному на рис. 15.7*б*:

k_2 — корень
k_1 — левый дочерний элемент k_2
d_0 — левый дочерний элемент k_1
d_1 — правый дочерний элемент k_1
k_5 — правый дочерний элемент k_2
k_4 — левый дочерний элемент k_5
k_3 — левый дочерний элемент k_4
d_2 — левый дочерний элемент k_3
d_3 — правый дочерний элемент k_3
d_4 — правый дочерний элемент k_4
d_5 — правый дочерний элемент k_5

15.5-2. Определите стоимость и структуру оптимального бинарного дерева поиска для множества, состоящего из $n = 7$ ключей, которым соответствуют следующие вероятности:

i	0	1	2	3	4	5	6	7
p_i		0,04	0,06	0,08	0,02	0,10	0,12	0,14
q_i	0,06	0,06	0,06	0,06	0,05	0,05	0,05	0,05

15.5-3. Предположим, что вместо того, чтобы поддерживать таблицу $w[i, j]$, значение $w(i, j)$ вычисляется в строке 8 процедуры OPTIMAL_BST непосредственно из уравнения (15.17) и используется в строке 10. Как это изменение повлияет на асимптотическое поведение времени выполнения этой процедуры?

⋆ 15.5-4. Кнут [184] показал, что всегда существуют корни оптимальных поддеревьев, такие что $root[i, j-1] \leqslant root[i, j] \leqslant root[i+1, j]$ для всех $1 \leqslant i \leqslant j \leqslant n$. Используйте этот факт для модификации процедуры OPTIMAL_BST, при которой время ее выполнения станет равным $\Theta(n^2)$.

Задачи

15-1. **Битоническая евклидова задача о коммивояжере**

Евклидова задача о коммивояжере (euclidean travelling-salesman problem) — это задача, в которой определяется кратчайший замкнутый путь, соединяющий заданное множество, которое состоит из n точек на плоскости. На рис. 15.9*а* приведено решение задачи, в которой имеется семь точек. В общем случае задача является NP-полной, поэтому считается, что для ее решения требуется время, превышающее полиномиальное (см. главу 34).

Бентли (J.L. Bentley) предположил, что задача упрощается благодаря ограничению интересующих нас маршрутов ***битоническими*** (bitonic tour), т.е. такими, которые начинаются в крайней левой точке, проходят слева направо, а затем — справа налево, возвращаясь прямо к исходной точке. На рис. 15.9*б* показан кратчайший битонический маршрут, проходящий через те же семь точек. В этом случае возможна разработка алгоритма, время работы которого является полиномиальным.

Напишите алгоритм, предназначенный для определения оптимального битонического маршрута, время работы которого будет равным $O(n^2)$. Предполагается, что не существует двух точек, координаты x которых совпадают. (*Указание*: перебор вариантов следует организовать слева направо, поддерживая оптимальные возможности для двух частей, из которых состоит маршрут.)

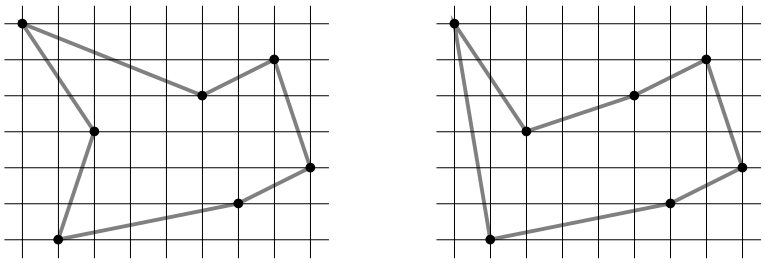


Рис. 15.9. *а*) кратчайший (не битонический) замкнутый маршрут длиной 24.89; *б*) кратчайший битонический маршрут длиной 25.58

15-2. **Вывод на печать с форматированием**

Рассмотрим задачу о форматировании параграфа текста, предназначенного для вывода на печать. Входной текст представляет собой последовательность, состоящую из n слов, длины которых равны $l_1, l_2, \ldots, l_n$ (длина слов измеряется в символах). При выводе на печать параграф должен быть аккуратно отформатирован и размещен в несколько строк, в каждой из которых помещается по M символов. Сформулируем критерий "аккуратного" форматирования. Если данная строка содержит слова от i-го до j-го и мы оставляем между словами ровно по одному пробелу, количество оставшихся пробелов, равное $M - j + i - \sum_{k=i}^{j} l_k$, должно быть неотрицательно, чтобы все слова поместились в строке. Мы хотим минимизировать сумму возведенных в куб остатков по всем строкам, кроме последней. Сформулируйте алгоритм, основанный на принципах динамического программирования, который аккуратно выводит на принтер параграф, состоящий из n слов. Проанализируйте время работы этого алгоритма и требуемое для его работы количество памяти.

15-3. **Расстояние редактирования**

Для того чтобы преобразовать исходную строку текста $x\,[1..m]$ в конечную строку $y\,[1..n]$, можно воспользоваться различными операциями преобразования. Наша цель заключается в том, чтобы для данных строк x и y определить последовательность преобразований, превращающих x в y. Для хранения промежуточных результатов используется массив z (предполагается, что он достаточно велик для хранения необходимых промежуточных результатов). Изначально массив z пуст, а в конце работы $z\,[j] = y\,[j]$ для всех $j = 1, 2, \ldots, n$. Поддерживается текущий индекс i массива x и индекс j массива z, и в ходе выполнения операций преобразования разрешено изменять элементы массива z и эти индексы. В начале работы алгоритма $i = j = 1$. В процессе преобразования необходимо

проверить каждый символ массива x. Это означает, что в конце выполнения последовательности операций значение i должно быть равно $m+1$.

Всего имеется шесть перечисленных ниже операций преобразования.

Копирование символа из массива x в массив z путем присвоения $z[j] \leftarrow x[i]$ с последующим увеличением индексов i и j. В этой операции проверяется элемент $x[i]$.

Замена символа из массива x другим символом c путем присвоения $z[j] \leftarrow c$ с последующим увеличением индексов i и j. В этой операции проверяется элемент $x[i]$.

Удаление символа из массива x путем увеличения на единицу индекса i и сохранения индекса j прежним. В этой операции проверяется элемент $x[i]$.

Вставка символа c в массив z путем присвоения $z[j] \leftarrow c$ и увеличения на единицу индекса j при сохранении индекса i прежним. В этой операции не проверяется ни один элемент массива x.

Перестановка двух следующих символов путем копирования их из массива x в массив z в обратном порядке. Это делается путем присваиваний $z[j] \leftarrow x[i+1]$ и $z[j+1] \leftarrow x[i]$ с последующим изменением значений индексов $i \leftarrow i+2$ и $j \leftarrow j+2$. В этой операции проверяются элементы $x[i]$ и $x[i+1]$.

Удаление остатка массива x путем присвоения $i \leftarrow m+1$. В этой операции проверяются все элементы массива x, которые не были проверены до сих пор. Если эта операция выполняется, то она должна быть последней.

В качестве примера приведем один из способов преобразования исходной строки `algorithm` в конечную строку `altruistic`. В ходе этого преобразования выполняется перечисленная ниже последовательность операций, в которых символы подчерка — это элементы $x[i]$ и $z[j]$ после выполнения очередной операции:

Операция	x	z
Исходные строки	a_lgorithm	_
Копирование	al_gorithm	a_
Копирование	alg_orithm	al_
Замена символом t	alg_rithm	alt_
Удаление	algo_ithm	alt_
Копирование	algor_thm	altr_
Вставка символа u	algor_thm	altru_

Вставка символа i	algori<u>i</u>thm	altrui_
Вставка символа s	algori<u>i</u>thm	altruis_
Перестановка	algorit<u>h</u>m	altruisti_
Вставка символа c	algorit<u>h</u>m	altruistic_
Удаление остатка	algorithm_	altruistic_

Заметим, что существует несколько других последовательностей операций, позволяющих преобразовать строку `algorithm` в строку `altruistic`.

С каждой операцией преобразования связана своя стоимость, которая зависит от конкретного приложения. Однако мы предположим, что стоимость каждой операции — известная константа. Кроме того, предполагается, что стоимости отдельно взятых операций копирования и замены меньше суммарной стоимости операций удаления и вставки, иначе применять эти операции было бы нерационально. Стоимость данной последовательности операций преобразования представляет собой сумму стоимостей отдельных операций последовательности. Стоимость приведенной выше последовательности преобразований строки `algorithm` в строку `altruistic` равна сумме трех стоимостей копирования, стоимости замены, стоимости удаления, четырем стоимостям вставки, стоимости перестановки и удаления остатка.

а) Пусть имеется две последовательности $x[1..m]$ и $y[1..n]$, а также множество, элементами которого являются стоимости операций преобразования. ***Расстоянием редактирования*** (edit distance) от x до y называется минимальная стоимость последовательности операций преобразования x в y. Опишите основанный на принципах динамического программирования алгоритм, в котором определяется расстояние редактирования от $x[1..m]$ до $y[1..n]$ и выводится оптимальная последовательность операций редактирования. Проанализируйте время работы этого алгоритма и требуемую для него память.

Задача о расстоянии редактирования — это обобщение задачи об анализе двух ДНК (см., например, книгу Сетубала (Setubal) и Мейданиса (Meidanis) [272, раздел 3.2]). Имеется два метода, позволяющих оценить подобие двух ДНК путем их выравнивания. Один способ выравнивания двух последовательностей x и y заключается в том, чтобы вставлять пробелы в произвольных местах этих двух последовательностей (включая позиции, расположенные в конце одной из них). Получившиеся в результате последовательности x' и y' должны иметь одинаковую длину, однако они не могут содержать пробелы в одинаковых позициях (т.е. не существует

такого индекса j, чтобы и $x'[j]$, и $y'[j]$ были пробелами). Далее каждой позиции присваивается определенное количество баллов в соответствии со сформулированными ниже правилами:

- $+1$, если $x'[j] = y'[j]$ и ни один из этих элементов не является пробелом;
- -1, если $x'[j] \neq y'[j]$ и ни один из этих элементов не является пробелом;
- -2, если элемент $x'[j]$ или элемент $y'[j]$ является пробелом.

Стоимость выравнивания определяется как сумма баллов, полученных при сравнении отдельных позиций. Например, если заданы последовательности $x =$ GATCGGCAT и $y =$ CAATGTGAATC, то одно из возможных выравниваний имеет вид:

```
G ATCG GCAT
CAAT GTGAATC
-*++*+*+-++*
```

Символ + под позицией указывает на то, что ей присваивается балл $+1$, символ - означает балл -1, а символ * — балл -2. Таким образом, полная стоимость равна $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$.

б) Объясните, как свести задачу о поиске оптимального выравнивания к задаче о поиске расстояния редактирования с помощью подмножества операций преобразования, состоящего из копирования, замены, удаления элемента, вставки, перестановки и удаления остатка.

15-4. **Вечеринка в фирме**

Профессор Тамада является консультантом президента корпорации по вопросам проведения вечеринок компании. Взаимоотношения между сотрудниками компании описываются иерархической структурой. Это означает, что отношение “начальник-подчиненный” образует дерево, во главе (в корне) которого находится президент. В отделе кадров каждому служащему присвоили рейтинг дружелюбия, представленный действительным числом. Чтобы на вечеринке все чувствовали себя раскованно, президент выдвинул требование, согласно которому ее не должны одновременно посещать сотрудник и его непосредственный начальник.

Профессору Тамаде предоставили дерево, описывающее структуру корпорации в представлении с левым дочерним и правым сестринским узлами, описанном в разделе 10.4. Каждый узел дерева, кроме указателей, содержит имя сотрудника и его рейтинг дружелюбия. Опишите алгоритм,

предназначенный для составления списка гостей, который бы давал максимальное значение суммы рейтингов дружелюбия гостей. Проанализируйте время работы этого алгоритма.

15-5. **Алгоритм Витерби**

Подход динамического программирования можно использовать в ориентированном графе $G = (V, E)$ для распознавания речи. Каждое ребро $(u, v) \in E$ графа помечено звуком $\sigma(u, v)$, который является членом конечного множества звуков Σ. Граф с метками представляет собой формальную модель речи человека, который разговаривает на языке, состоящем из ограниченного множества звуков. Все пути в графе, которые начинаются в выделенной вершине $v_0 \in V$, соответствуют возможной последовательности звуков, допустимых в данной модели. Метка направленного пути по определению является объединением меток, соответствующих ребрам, из которых состоит путь.

а) Разработайте эффективный алгоритм, который для заданного графа G с помеченными ребрами и выделенной вершиной v_0, и последовательности $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ символов из множества Σ возвращал бы путь в графе G, который начинается в вершине v_0 и имеет метку s (если таковой существует). В противном случае алгоритм должен возвращать строку NO-SUCH-PATH (нет такого пути). Проанализируйте время работы алгоритма. (*Указание*: могут оказаться полезными концепции, изложенные в главе 22.)

Теперь предположим, что каждому ребру $(u, v) \in E$ также сопоставляется неотрицательная вероятность $p(u, v)$ перехода по этому ребру из вершины u. При этом издается соответствующий звук. Сумма вероятностей по всем ребрам, исходящим из произвольной вершины, равна 1. По определению вероятность пути равна произведению вероятностей составляющих его ребер. Вероятность пути, берущего начало в вершине v_0, можно рассматривать как вероятность "случайной прогулки", которая начинается в этой вершине и продолжается по случайному пути. Выбор каждого ребра на этом пути в вершине u производится в соответствии с вероятностями ребер, исходящих из этой вершины.

б) Обобщите сформулированный в части *а*) ответ так, чтобы возвращаемый путь (если он существует) был *наиболее вероятным* из всех путей, начинающихся в вершине v_0 и имеющих метку s. Проанализируйте время работы этого алгоритма.

15-6. **Передвижение по шахматной доске**

Предположим, имеется шахматная доска размером $n \times n$ клеток и шашка. Необходимо провести шашку от нижнего края доски к верхнему. Ходы

можно делать в соответствии со сформулированным далее правилом. На каждом ходу шашка передвигается в одну из следующих клеток:

а) на соседнюю клетку, расположенную непосредственно над текущей;

б) на ближайшую клетку, расположенную по диагонали в направлении вверх и влево (если шашка не находится в крайнем левом столбце);

в) на ближайшую клетку, расположенную по диагонали в направлении вверх и вправо (если шашка не находится в крайнем правом столбце).

Каждый ход из клетки x в клетку y имеет стоимость $p(x, y)$, которая задается для всех пар (x, y), допускающих ход. При этом величины $p(x, y)$ не обязательно положительны.

Сформулируйте алгоритм, определяющий последовательность ходов, для которой стоимость полного пути от нижнего края доски к верхнему краю будет максимальной. В качестве начальной может быть выбрана любая клетка в нижнем ряду. Аналогично, маршрут может заканчиваться в любой клетке верхнего ряда. Чему равно время работы этого алгоритма?

15-7. **Расписание для получения максимального дохода**

Предположим, что имеется один компьютер и множество, состоящее из n заданий $a_1, a_2, \ldots, a_n$, которые нужно выполнить на этом компьютере. Каждому заданию a_j соответствует время обработки t_j, прибыль p_j и конечный срок выполнения d_j. Компьютер не способен выполнять несколько заданий одновременно. Кроме того, если запущено задание a_j, то оно должно выполняться без прерываний в течение времени t_j. Если задание a_j завершается до того, как истечет конечный срок его выполнения d_j, вы получаете доход p_j. Если же это произойдет после момента времени d_j, то полученный доход равен нулю. Сформулируйте алгоритм, который бы выдавал расписание для получения максимальной прибыли. При этом предполагается, что любое время выполнения задания выражается целым числом в интервале от 1 до n. Чему равно время работы этого алгоритма?

Заключительные замечания

Беллман (R. Bellman) приступил к систематическому изучению динамического программирования в 1955 году. Как в названии “динамическое программирование”, так и в термине “линейное программирование” слово “программирование” относится к методу табличного решения. Методы оптимизации, включающие в себя элементы динамического программирования, были известны и раньше, однако именно Беллман дал строгое математическое обоснование этой области [34].

Ху (Hu) и Шинг (Shing) [159,160] сформулировали алгоритм, позволяющий решить задачу о перемножении матриц в течение времени $O(n \lg n)$.

По-видимому, алгоритм решения задачи о самой длинной общей подпоследовательности, время работы которого равно $O(mn)$, — плод "народного" творчества. Кнут (Knuth) [63] поставил вопрос о том, существует ли алгоритм решения этой задачи, время работы которого возрастало бы медленнее, чем квадрат размера. Масек (Masek) и Патерсон (Paterson) ответили на этот вопрос утвердительно, разработав алгоритм, время работы которого равно $O(mn/\lg n)$, где $n \leqslant m$, а последовательности составлены из элементов множества ограниченного размера. Шиманski (Szimanski) [288] показал, что в особом случае, когда ни один элемент не появляется во входной последовательности более одного раза, эту задачу можно решить в течение времени $O((n+m)\lg(n+m))$. Многие из этих результатов были обобщены для задачи о вычислении расстояния редактирования (задача 15-3).

Ранняя работа Гильберта (Gilbert) и Мура (Moore) [114], посвященная бинарному кодированию переменной длины, нашла применение при конструировании оптимального бинарного дерева поиска для случая, когда все вероятности p_i равны 0. В этой работе приведен алгоритм, время работы которого равно $O\left(n^3\right)$. Ахо (Aho), Хопкрофт (Hopcroft) и Ульман (Ullman) [5] представили алгоритм, описанный в разделе 15.5. Упражнение 15.5-4 предложено Кнутом [184]. Ху и Такер (Tucker) [161] разработали алгоритм для случая, когда все вероятности p_i равны 0; время работы этого алгоритма равно $O\left(n^2\right)$, а количество необходимой для него памяти — $O(n)$. Впоследствии Кнуту [185] удалось уменьшить это время до величины $O(n \lg n)$.

ГЛАВА 16

Жадные алгоритмы

Алгоритмы, предназначенные для решения задач оптимизации, обычно представляют собой последовательность шагов, на каждом из которых предоставляется некоторое множество выборов. Определение наилучшего выбора, руководствуясь принципами динамического программирования, во многих задачах оптимизации напоминает стрельбу из пушки по воробьям; другими словами, для этих задач лучше подходят более простые и эффективные алгоритмы. В ***жадном алгоритме*** (greedy algorithm) всегда делается выбор, который кажется самым лучшим в данный момент — т.е. производится локально оптимальный выбор в надежде, что он приведет к оптимальному решению глобальной задачи. В этой главе рассматриваются задачи оптимизации, пригодные для решения с помощью жадных алгоритмов. Перед освоением главы следует ознакомиться с динамическим программированием, изложенным в главе 15, в частности, с разделом 15.3.

Жадные алгоритмы не всегда приводят к оптимальному решению, но во многих задачах они дают нужный результат. В разделе 16.1 рассматривается простая, но нетривиальная задача о выборе процессов, эффективное решение которой можно найти с помощью жадного алгоритма. Чтобы прийти к жадному алгоритму, сначала будет рассмотрено решение, основанное на парадигме динамического программирования, после чего будет показано, что оптимальное решение можно получить, исходя из принципов жадных алгоритмов. В разделе 16.2 представлен обзор основных элементов подхода, в соответствии с которым разрабатываются жадные алгоритмы. Это позволит упростить обоснование корректности жадных алгоритмов, не используя при этом сравнение с динамическим программированием, как это делается в разделе 16.1. В разделе 16.3 приводится важное приложение методов жадного программирования: разработка кодов Хаффмана (Huffman) для

сжатия данных. В разделе 16.4 исследуются теоретические положения, на которых основаны комбинаторные структуры, известные под названием “матроиды”, для которых жадный алгоритм всегда дает оптимальное решение. Наконец, в разделе 16.5 матроиды применяются для решения задачи о составлении расписания заданий равной длительности с предельным сроком и штрафами.

Жадный метод обладает достаточной мощью и хорошо подходит для довольно широкого класса задач. В последующих главах представлены многие алгоритмы, которые можно рассматривать как применение жадного метода, включая алгоритмы поиска минимальных остовных деревьев (minimum-spanning-tree) (глава 23), алгоритм Дейкстры (Dijkstra) для определения кратчайших путей из одного источника (глава 24) и эвристический жадный подход Чватала (Chvatal) к задаче о покрытии множества (set-covering) (глава 35). Алгоритмы поиска минимальных остовных деревьев являются классическим примером применения жадного метода. Несмотря на то, что эту главу и главу 23 можно читать независимо друг от друга, может оказаться полезно читать их одна за другой.

16.1 Задача о выборе процессов

В качестве первого примера рассмотрим задачу о составлении расписания для нескольких конкурирующих процессов, каждый из которых безраздельно использует общий ресурс. Цель этой задачи — выбор набора взаимно совместимых процессов, образующих множество максимального размера. Предположим, имеется множество $S = \{a_1, a_2, \ldots, a_n\}$, состоящее из n ***процессов*** (activities). Процессам требуется некоторый ресурс, который одновременно может использоваться лишь одним процессом. Каждый процесс a_i характеризуется ***начальным моментом*** s_i и ***конечным моментом*** f_i, где $0 \leqslant s_i < f_i < \infty$. Будучи выбран, процесс a_i длится в течение полуоткрытого интервала времени $[s_i, f_i)$. Процессы a_i и a_j ***совместимы*** (compatible), если интервалы $[s_i, f_i)$ и $[s_j, f_j)$ не перекрываются (т.е. если $s_i \geqslant f_j$ или $s_j \geqslant f_i$). ***Задача о выборе процессов*** (activity-selection problem) заключается в том, чтобы выбрать подмножество взаимно совместимых процессов, образующих множество максимального размера. Например, рассмотрим описанное ниже множество S процессов, отсортированных в порядке возрастания моментов окончания:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

(Вскоре станет понятно, почему удобнее рассматривать процессы, расположенные именно в таком порядке.) В данном примере из взаимно совместимых процессов можно составить подмножество $\{a_3, a_9, a_{11}\}$. Однако оно не является максималь-

ным, поскольку существует подмножество $\{a_1, a_4, a_8, a_{11}\}$, состоящее из большего количества элементов. Еще одно такое подмножество — $\{a_2, a_4, a_9, a_{11}\}$.

Разобьем решение этой задачи на несколько этапов. Начнем с того, что сформулируем решение рассматриваемой задачи, основанное на принципах динамического программирования. Это оптимальное решение исходной задачи получается путем комбинирования оптимальных решений подзадач. Рассмотрим несколько вариантов выбора, который делается в процессе определения подзадачи, использующейся в оптимальном решении. Впоследствии станет понятно, что заслуживает внимания лишь один выбор — жадный — и что когда делается этот выбор, одна из подзадач гарантированно получается пустой. Остается лишь одна непустая подзадача. Исходя из этих наблюдений, мы разработаем рекурсивный жадный алгоритм, предназначенный для решения задачи о составлении расписания процессов. Процесс разработки жадного алгоритма будет завершен его преобразованием из рекурсивного в итерационный. Описанные в этом разделе этапы сложнее, чем те, что обычно имеют место при разработке жадных алгоритмов, однако они иллюстрируют взаимоотношение между жадными алгоритмами и динамическим программированием.

Оптимальная подструктура задачи о выборе процессов

Как уже упоминалось, мы начнем с того, что разработаем решение для задачи о выборе процессов по методу динамического программирования. Как и в главе 15, первый шаг будет состоять в том, чтобы найти оптимальную подструктуру и с ее помощью построить оптимальное решение задачи, пользуясь оптимальными решениями подзадач.

В главе 15 мы убедились, что для этого нужно определить подходящее пространство подзадач. Начнем с определения множеств

$$S_{ij} = \{a_k \in S : f_i \leqslant s_k < f_k \leqslant s_j\}\,.$$

S_{ij} — подмножество процессов из множества S, которые можно успеть выполнить в промежутке времени между завершением процесса a_i и началом процесса a_j. Фактически множество S_{ij} состоит из всех процессов, совместимых с процессами a_i и a_j, а также теми, которые оканчиваются не позже процесса a_i и теми, которые начинаются не ранее процесса a_j. Для представления всей задачи добавим фиктивные процессы a_0 и a_{n+1} и примем соглашение, что $f_0 = 0$ и $s_{n+1} = \infty$. Тогда $S = S_{0,n+1}$, а индексы i и j находятся в диапазоне $0 \leqslant i, j \leqslant n+1$.

Дополнительные ограничения диапазонов индексов i и j можно получить с помощью приведенных ниже рассуждений. Предположим, что процессы отсортированы в порядке возрастания соответствующих им конечных моментов времени:

$$f_0 \leqslant f_1 \leqslant \cdots \leqslant f_n \leqslant f_{n+1}. \tag{16.1}$$

Утверждается, что в этом случае $S_{ij} = \emptyset$, если $i \geqslant j$. Почему? Предположим, что существует процесс $a_k \in S_{ij}$ для некоторого $i \geqslant j$, так что процесс a_i следует после процесса a_j, если расположить их в порядке сортировки. Однако из определения S_{ij} следует соотношение $f_i \leqslant s_k < f_k \leqslant s_j < f_j$, откуда $f_i < f_j$, что противоречит предположению о том, что процесс a_i следует после процесса a_j, если расположить их в порядке сортировки. Можно прийти к выводу, что если процессы отсортированы в порядке монотонного возрастания времени их окончания, то пространство подзадач образуется путем выбора максимального подмножества взаимно совместимых процессов из множества S_{ij} для $0 \leqslant i < j \leqslant n + 1$ (мы знаем, что все прочие S_{ij} — пустые).

Чтобы понять подструктуру задачи о выборе процессов, рассмотрим некоторую непустую подзадачу S_{ij}[1] и предположим, что ее решение включает некоторый процесс a_k, так что $f_i \leqslant s_k < f_k \leqslant s_j$. С помощью процесса a_k генерируются две подзадачи, S_{ik} (процессы, которые начинаются после окончания процесса a_i и заканчиваются перед началом процесса a_k) и S_{kj} (процессы, которые начинаются после окончания процесса a_k и заканчиваются перед началом процесса a_j), каждая из которых состоит из подмножества процессов, входящих в S_{ij}. Решение задачи S_{ij} представляет собой объединение решений задач S_{ik}, S_{kj} и процесса a_k. Таким образом, количество процессов, входящее в состав решения задачи S_{ij}, — это сумма количества процессов в решении задачи S_{ik}, количества процессов в решении задачи S_{kj} и еще одного процесса (a_k).

Опишем оптимальную подструктуру этой задачи. Предположим, что оптимальное решение A_{ij} задачи S_{ij} включает в себя процесс a_k. Тогда решения A_{ik} задачи S_{ik} и A_{kj} задачи S_{kj} в рамках оптимального решения S_{ij} тоже должны быть оптимальными. Как обычно, это доказывается с помощью рассуждений типа "вырезания и вставки". Если бы существовало решение A'_{ik} задачи S_{ik}, включающее в себя большее количество процессов, чем A_{ik}, в решение A_{ij} можно было бы вместо A_{ik} подставить A'_{ik}, что привело бы к решению задачи S_{ij}, в котором содержится больше процессов, чем в A_{ij}. Поскольку предполагается, что A_{ij} — оптимальное решение, мы приходим к противоречию. Аналогично, если бы существовало решение A'_{kj} задачи S_{kj}, содержащее больше процессов, чем A_{kj}, то путем замены A_{kj} на A'_{kj} можно было бы получить решение задачи S_{ij}, в которое входит больше процессов, чем в A_{ij}.

Теперь с помощью сформулированной выше оптимальной подструктуры покажем, что оптимальное решение задачи можно составить из оптимальных решений подзадач. Мы уже знаем, что в любое решение непустой задачи S_{ij} входит некоторый процесс a_k и что любое оптимальное решение задачи содержит в себе

[1] Иногда о множествах S_{ij} мы будем говорить не как о множествах процессов, а как о вспомогательных задачах. Из контекста всегда будет понятно, что именно обозначает S_{ij} — множество процессов или вспомогательную задачу, входными данными для которой является это множество.

оптимальные решения подзадач S_{ik} и S_{kj}. Таким образом, максимальное подмножество взаимно совместимых процессов множества S_{ij} можно составить путем разбиения задачи на две подзадачи (определение подмножеств максимального размера, состоящих из взаимно совместимых процессов в задачах S_{ik} и S_{kj}) — собственно нахождения максимальных подмножеств A_{ik} и A_{kj} взаимно совместимых процессов, являющихся решениями этих подзадач, и составления подмножества A_{ij} максимального размера, включающего в себя взаимно совместимые задачи:

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}. \tag{16.2}$$

Оптимальное решение всей задачи представляет собой решение задачи $S_{0,n+1}$.

Рекурсивное решение

Второй этап разработки решения по методу динамического программирования — определение значения, соответствующего оптимальному решению. Пусть в задаче о выборе процесса $c[i,j]$ — количество процессов в подмножестве максимального размера, состоящем из взаимно совместимых процессов в задаче S_{ij}. Эта величина равна нулю при $S_{ij} = \emptyset$; в частности, $c[i,j] = 0$ при $i \geqslant j$.

Теперь рассмотрим непустое подмножество S_{ij}. Мы уже знаем, что если процесс a_k используется в максимальном подмножестве, состоящем из взаимно совместимых процессов задачи S_{ij}, то для подзадач S_{ik} и S_{kj} также используются подмножества максимального размера. С помощью уравнения (16.2) получаем рекуррентное соотношение

$$c[i,j] = c[i,k] + c[k,j] + 1.$$

В приведенном выше рекурсивном уравнении предполагается, что значение k известно, но на самом деле это не так. Это значение можно выбрать из $j - i - 1$ возможных значений: $k = i+1, \ldots, j-1$. Поскольку в подмножестве максимального размера для задачи S_{ij} должно использоваться одно из этих значений k, нужно проверить, какое из них подходит лучше других. Таким образом, полное рекурсивное определение величины $c[i,j]$ принимает вид:

$$c[i,j] = \begin{cases} 0 & \text{при } S_{ij} = \emptyset, \\ \max\limits_{\substack{i<k<j \text{ и} \\ a_k \in S_{ij}}} \{c[i,k] + c[k,j] + 1\} & \text{при } S_{ij} \neq \emptyset. \end{cases} \tag{16.3}$$

Преобразование решения динамического программирования в жадное решение

На данном этапе не составляет труда написать восходящий алгоритм динамического программирования, основанный на рекуррентном уравнении (16.3). Это

предлагается сделать читателю в упражнении (16.1-1). Однако можно сделать два наблюдения, позволяющие упростить полученное решение.

Теорема 16.1. Рассмотрим произвольную непустую задачу S_{ij} и пусть a_m — процесс, который оканчивается раньше других:

$$f_m = \min\{f_k : a_k \in S_{ij}\}.$$

В этом случае справедливы такие утверждения.

1. Процесс a_m используется в некотором подмножестве максимального размера, состоящем из взаимно совместимых процессов задачи S_{ij}.
2. Подзадача S_{im} пустая, поэтому в результате выбора процесса a_m непустой остается только подзадача S_{mj}.

Доказательство. Сначала докажем вторую часть теоремы, так как она несколько проще. Предположим, что подзадача S_{im} непустая и существует некоторый процесс a_k, такой что $f_i \leqslant s_k < f_k \leqslant s_m < f_m$. Тогда процесс a_k также входит в решение подзадачи S_{im}, причем он оканчивается раньше, чем процесс a_m, что противоречит выбору процесса a_m. Таким образом, мы приходим к выводу, что решение подзадачи S_{im} — пустое множество.

Чтобы доказать первую часть, предположим, что A_{ij} — подмножество максимального размера взаимно совместимых процессов задачи S_{ij}, и упорядочим процессы этого подмножества в порядке возрастания времени их окончания. Пусть a_k — первый процесс множества A_{ij}. Если $a_k = a_m$, то доказательство завершено, поскольку мы показали, что процесс a_m используется в некотором подмножестве максимального размера, состоящем из взаимно совместимых процессов задачи S_{ij}. Если же $a_k \neq a_m$, то составим подмножество $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$. Процессы в A'_{ij} не перекрываются, поскольку процессы во множестве A_{ij} не перекрываются, a_k — процесс из множества A_{ij}, который оканчивается раньше всех, и $f_m \leqslant f_k$. Заметим, что количество процессов во множестве A'_{ij} совпадает с количеством процессов во множестве A_{ij}, поэтому A'_{ij} — подмножество максимального размера, состоящее из взаимно совместимых процессов задачи S_{ij} и включающее в себя процесс a_m. ■

Почему теорема 16.1 имеет такое большое значение? Из раздела 15.3 мы узнали, что оптимальные подструктуры различаются количеством подзадач, использующихся в оптимальном решении исходной задачи, и количеством возможностей, которые следует рассмотреть при определении того, какие подзадачи нужно выбрать. В решении, основанном на принципах динамического программирования, в оптимальном решении используется две подзадачи, а также до $j-i-1$ вариантов выбора для подзадачи S_{ij}. Благодаря теореме 16.1 обе эти величины значительно

уменьшаются: в оптимальном решении используется лишь одна подзадача (вторая подзадача гарантированно пустая), а в процессе решения подзадачи S_{ij} достаточно рассмотреть только один выбор — тот процесс, который оканчивается раньше других. К счастью, его легко определить.

Кроме уменьшения количества подзадач и количества выборов, теорема 16.1 дает еще одно преимущество: появляется возможность решать каждую подзадачу в нисходящем направлении, а не в восходящем, как это обычно происходит в динамическом программировании. Чтобы решить подзадачу S_{ij}, в ней выбирается процесс a_m, который оканчивается раньше других, после чего в это решение добавляется множество процессов, использующихся в оптимальном решении подзадачи S_{mj}. Поскольку известно, что при выбранном процессе a_m в оптимальном решении задачи S_{ij} безусловно используется решение задачи S_{mj}, нет необходимости решать задачу S_{mj} *до* задачи S_{ij}. Чтобы решить задачу S_{ij}, можно *сначала* выбрать a_m, который представляет собой процесса из S_{ij}, который заканчивается раньше других, а *потом* решать задачу S_{mj}.

Заметим также, что существует единый шаблон для всех подзадач, которые подлежат решению. Наша исходная задача — $S = S_{0,n+1}$. Предположим, что в качестве процесса задачи $S_{0,n+1}$, который оканчивается раньше других, выбран процесс a_{m_1}. (Поскольку процессы отсортированы в порядке монотонного возрастания времени их окончания, и $f_0 = 0$, должно выполняться равенство $m_1 = 1$.) Следующая подзадача — $S_{m_1,n+1}$. Теперь предположим, что в качестве процесса задачи $S_{m_1,n+1}$, который оканчивается раньше других, выбран процесс a_{m_2} (не обязательно, чтобы $m_2 = 2$). Следующая на очереди подзадача — $S_{m_2,n+1}$. Продолжая рассуждения, нетрудно убедиться в том, что каждая подзадача будет иметь вид $S_{m_i,n+1}$ и ей будет соответствовать некоторый номер процесса m_i. Другими словами, каждая подзадача состоит из некоторого количества процессов, завершающихся последними, и это количество меняется от одной подзадачи к другой.

Для процессов, которые первыми выбираются в каждой подзадаче, тоже существует свой шаблон. Поскольку в задаче $S_{m_i,n+1}$ всегда выбирается процесс, который оканчивается раньше других, моменты окончания процессов, которые последовательно выбираются во всех подзадачах, образуют монотонно возрастающую последовательность. Более того, каждый процесс в процессе решения задачи можно рассмотреть лишь один раз, воспользовавшись сортировкой в порядке возрастания времен окончания процессов.

В ходе решения подзадачи всегда выбирается процесс a_m, который оканчивается раньше других. Таким образом, выбор является жадным в том смысле, что интуитивно для остальных процессов он оставляет как можно больше возможностей войти в расписание. Таким образом, жадный выбор — это такой выбор, который максимизирует количество процессов, пока что не включенных в расписание.

Рекурсивный жадный алгоритм

После того как стал понятен путь упрощения решения, основанного на принципах динамического программирования, и преобразования его в нисходящий метод, можно перейти к рассмотрению алгоритма, работающего исключительно в жадной нисходящей манере. Здесь приводится простое рекурсивное решение, реализованное в виде процедуры RECURSIVE_ACTIVITY_SELECTOR. На ее вход подаются значения начальных и конечных моментов процессов, представленные в виде массивов s и f, а также индексы i и n, определяющие подзадачу $S_{i,n+1}$, которую требуется решить. (Параметр n идентифицирует последний реальный процесс a_n в подзадаче, а не фиктивный процесс a_{n+1}, который тоже входит в эту подзадачу.) Процедура возвращает множество максимального размера, состоящее из взаимно совместимых процессов задачи $S_{i,n+1}$. В соответствии с уравнением (16.1), предполагается, что все n входных процессов расположены в порядке монотонного возрастания времени их окончания. Если это не так, их можно отсортировать в указанном порядке за время $O(n \lg n)$. Начальный вызов этой процедуры имеет вид RECURSIVE_ACTIVITY_SELECTOR$(s, f, 0, n)$.

```
RECURSIVE_ACTIVITY_SELECTOR(s, f, i, n)
1  m ← i + 1
2  while m ≤ n и s_m < f_i          ▷ Поиск первого процесса в S_{i,n+1}.
3        do m ← m + 1
4  if m ≤ n
5     then return {a_m} ∪ RECURSIVE_ACTIVITY_SELECTOR(s, f, m, n)
6     else return ∅
```

Работа представленного алгоритма для рассматривавшихся в начале этого раздела 11 процессов проиллюстрирована на рис. 16.1. Процессы, которые рассматриваются в каждом рекурсивном вызове, разделены горизонтальными линиями. Фиктивный процесс a_0 оканчивается в нулевой момент времени, и в начальном вызове, который имеет вид RECURSIVE_ACTIVITY_SELECTOR$(s, f, 0, 11)$, выбирается процесс a_1. В каждом рекурсивном вызове заштрихованы процессы, которые уже были выбраны, а белым цветом обозначен рассматриваемый процесс. Если начальный момент процесса наступает до конечного момента процесса, который был добавлен последним (т.е. стрелка, проведенная от первого процесса ко второму, направлена влево), такой процесс отвергается. В противном случае (стрелка направлена прямо вверх или вправо) процесс выбирается. В последнем рекурсивном вызове процедуры, имеющем вид RECURSIVE_ACTIVITY_SELECTOR$(s, f, 11, 11)$, возвращается $\emptyset$. Результирующее множество выбранных процессов — $\{a_1, a_4, a_8, a_{11}\}$.

В рекурсивном вызове процедуры RECURSIVE_ACTIVITY_SELECTOR(s, f, i, n) в цикле **while** в строках 2–3 осуществляется поиск первого процесса задачи $S_{i,n+1}$. В цикле проверяются процессы $a_{i+1}, a_{i+2}, \ldots, a_n$ до тех пор, пока не будет найден

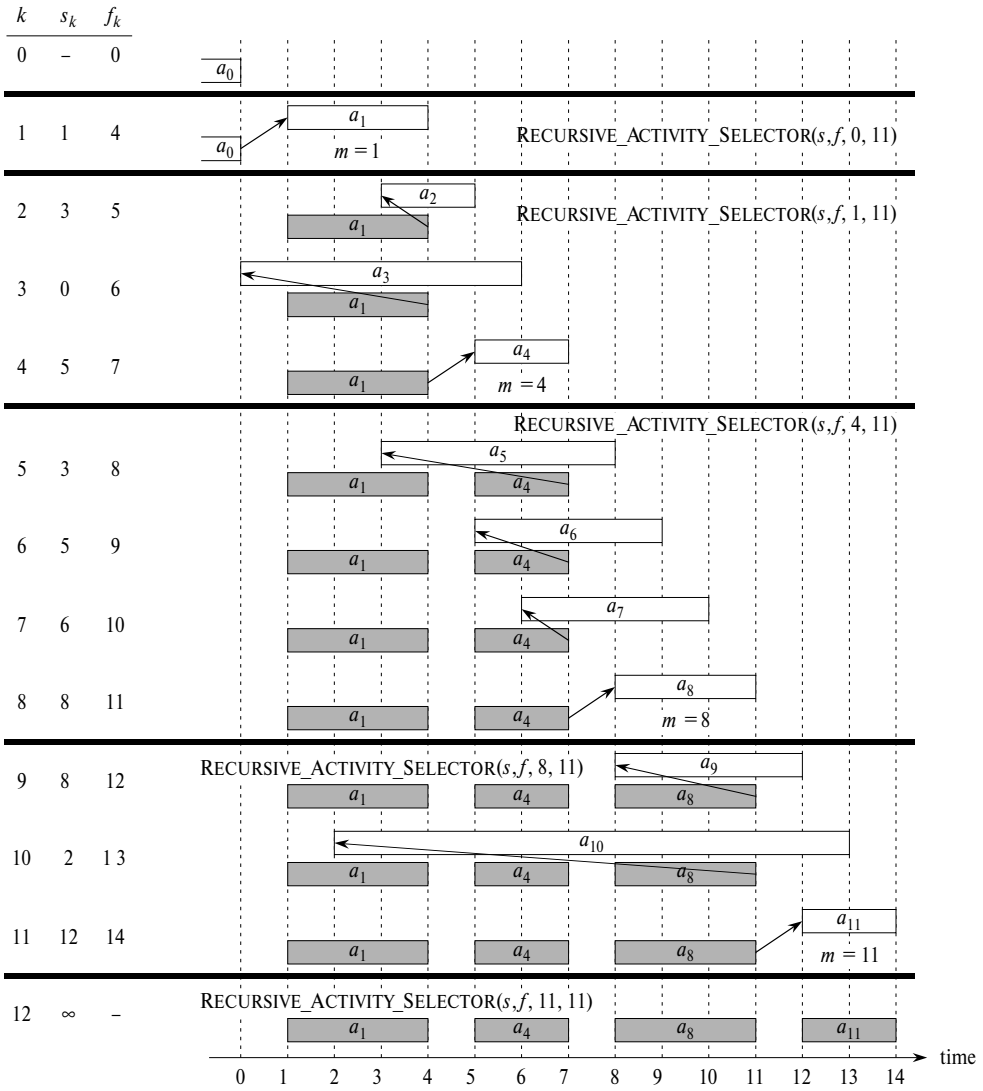


Рис. 16.1. Обработка алгоритмом RECURSIVE_ACTIVITY_SELECTOR множества из одиннадцати процессов

первый процесс a_m, совместимый с процессом a_i; для такого процесса справедливо соотношение $s_m \geqslant f_i$. Если цикл завершается из-за того, что такой процесс найден, то происходит возврат из процедуры в строке 5, в которой процесс $\{a_m\}$ объединяется с подмножеством максимального размера для задачи $S_{m,n+1}$, которое возвращается рекурсивным вызовом RECURSIVE_ACTIVITY_SELECTOR(s, f, m, n). Еще одна возможная причина завершения процесса — достижение условия $m > n$.

В этом случае проверены все процессы, но среди них не найден такой, который был бы совместим с процессом a_i. Таким образом, $S_{i,n+1} = \emptyset$ и процедура возвращает значение $\emptyset$ в строке 6.

Если процессы отсортированы в порядке, заданном временем их окончания, время, которое затрачивается на вызов процедуры RECURSIVE_ACTIVITY_SELECTOR$(s, f, 0, n)$, равно $\Theta(n)$. Это можно показать следующим образом. Сколько бы ни было рекурсивных вызовов, каждый процесс проверяется в цикле **while** в строке 2 ровно по одному разу. В частности, процесс a_k проверяется в последнем вызове, когда $i < k$.

Итерационный жадный алгоритм

Представленную ранее рекурсивную процедуру легко преобразовать в итеративную. Процедура RECURSIVE_ACTIVITY_SELECTOR почти подпадает под определение оконечной рекурсии (см. задачу 7-4): она оканчивается рекурсивным вызовом самой себя, после чего выполняется операция объединения. Преобразование процедуры, построенной по принципу оконечной рекурсии, в итерационную — обычно простая задача. Некоторые компиляторы разных языков программирования выполняют эту задачу автоматически. Как уже упоминалось, процедура RECURSIVE_ACTIVITY_SELECTOR выполняется для подзадач $S_{i,n+1}$, т.е. для подзадач, состоящих из процессов, которые оканчиваются позже других.

Процедура GREEDY_ACTIVITY_SELECTOR — итеративная версия процедуры RECURSIVE_ACTIVITY_SELECTOR. В ней также предполагается, что входные процессы расположены в порядке монотонного возрастания времен окончания. Выбранные процессы объединяются в этой процедуре в множество A, которое и возвращается процедурой после ее окончания.

```
GREEDY_ACTIVITY_SELECTOR(s, f)
1  n ← length[s]
2  A ← {a_1}
3  i ← 1
4  for m ← 2 to n
5       do if s_m ≥ f_i
6            then A ← A ∪ {a_m}
7                 i ← m
8  return A
```

Процедура работает следующим образом. Переменная i индексирует самое последнее добавление к множеству A, соответствующее процессу a_i в рекурсивной версии. Поскольку процессы рассматриваются в порядке монотонного возрастания моментов их окончания, f_i — всегда максимальное время окончания всех

процессов, принадлежащих множеству A:

$$f_i = \max\{f_k : a_k \in A\} \tag{16.4}$$

В строках 2–3 выбирается процесс a_1, инициализируется множество A, содержащее только этот процесс, а переменной i присваивается индекс этого процесса. В цикле **for** в строках 4–7 происходит поиск процесса задачи $S_{i,n+1}$, оканчивающегося раньше других. В этом цикле по очереди рассматривается каждый процесс a_m, который добавляется в множество A, если он совместим со всеми ранее выбранными процессами; этот процесс оканчивается раньше других в задаче $S_{i,n+1}$. Чтобы узнать, совместим ли процесс a_m с процессами, которые уже содержатся во множестве A, в соответствии с уравнением (16.4) достаточно проверить (строка 5), что его начальный момент s_m наступает не раньше момента f_i окончания последнего из добавленных в множество A процессов. Если процесс a_m удовлетворяет сформулированным выше условиям, то в строках 6–7 он добавляется в множество A и переменной i присваивается значение m. Множество A, возвращаемое процедурой GREEDY_ACTIVITY_SELECTOR(s, f), совпадает с тем, которое возвращается процедурой RECURSIVE_ACTIVITY_SELECTOR$(s, f, 0, n)$.

Процедура GREEDY_ACTIVITY_SELECTOR, как и ее рекурсивная версия, составляет расписание для n-элементного множества в течение времени $\Theta(n)$. Это утверждение справедливо в предположении, что процессы уже отсортированы в порядке возрастания времени их окончания.

Упражнения

16.1-1. Разработайте алгоритм динамического программирования для решения задачи о выборе процессов, основанный на рекуррентном соотношении (16.3). В этом алгоритме должны вычисляться определенные выше размеры $c[i, j]$, а также выводиться подмножество процессов A максимального размера. Предполагается, что процессы отсортированы в порядке, заданном уравнением (16.1). Сравните время работы найденного алгоритма со временем работы процедуры GREEDY_ACTIVITY_SELECTOR.

16.1-2. Предположим, что вместо того, чтобы все время выбирать процесс, который оканчивается раньше других, выбирается процесс, который начинается позже других и совместим со всеми ранее выбранными процессами. Опишите этот подход как жадный алгоритм и докажите, что он позволяет получить оптимальное решение.

16.1-3. Предположим, что имеется множество процессов, для которых нужно составить расписание при наличии большого количества ресурсов. Цель — включить в расписание все процессы, использовав при этом как можно меньше ресурсов. Сформулируйте эффективный жадный алгоритм,

позволяющий определить, какой ресурс должен использоваться тем или иным процессом.

(Эта задача известна также как задача о ***раскрашивании интервального графа*** (interval-graph colouring problem). Можно создать интервальный граф, вершины которого сопоставляются заданным процессам, а ребра соединяют несовместимые процессы. Минимальное количество цветов, необходимых для раскрашивания всех вершин таким образом, чтобы никакие две соединенные вершины не имели один и тот же цвет, будет равно минимальному количеству ресурсов, необходимых для работы всех заданных процессов.)

16.1-4. Не все жадные подходы к задаче о выборе процессов позволяют получить множество, состоящее из максимального количества взаимно совместимых процессов. Приведите пример, иллюстрирующий, что выбор самых коротких процессов среди тех, что совместимы с ранее выбранными, не дает правильного результата. Выполните такое же задание для подходов, в одном из которых всегда выбирается процесс, совместимый с ранее выбранными и перекрывающийся с минимальным количеством оставшихся, а в другом — совместимый с ранее выбранными процесс, который начинается раньше других.

16.2 Элементы жадной стратегии

Жадный алгоритм позволяет получить оптимальное решение задачи путем осуществления ряда выборов. В каждой точке принятия решения в алгоритме делается выбор, который в данный момент выглядит самым лучшим. Эта эвристическая стратегия не всегда дает оптимальное решение, но все же решение может оказаться и оптимальным, в чем мы смогли убедиться на примере задачи о выборе процессов. В настоящем разделе обсуждаются некоторые общие свойства жадных методов.

Процесс разработки жадного алгоритма, рассмотренный в разделе 16.1, несколько сложнее, чем обычно. Были пройдены перечисленные ниже этапы.

1. Определена оптимальная подструктура задачи.
2. Разработано рекурсивное решение.
3. Доказано, что на любом этапе рекурсии один из оптимальных выборов является жадным. Из этого следует, что всегда можно делать жадный выбор.
4. Показано, что все возникающие в результате жадного выбора подзадачи, кроме одной, — пустые.
5. Разработан рекурсивный алгоритм, реализующий жадную стратегию.
6. Рекурсивный алгоритм преобразован в итеративный.

В ходе выполнения этих этапов мы во всех подробностях смогли рассмотреть, как динамическое программирование послужило основой для жадного алгоритма. Однако обычно на практике при разработке жадного алгоритма эти этапы упрощаются. Мы разработали подструктуру так, чтобы в результате жадного выбора оставалась только одна подзадача, подлежащая оптимальному решению. Например, в задаче о выборе процессов сначала определяются подзадачи S_{ij}, в которых изменяются оба индекса, — и i, и j. Потом мы выяснили, что если всегда делается жадный выбор, то подзадачи можно было бы ограничить видом $S_{i,n+1}$.

Можно предложить альтернативный подход, в котором оптимальная подструктура приспосабливалась бы специально для жадного выбора — т.е. второй индекс можно было бы опустить и определить подзадачи в виде $S_i = \{a_k \in S : f_i \leqslant s_k\}$. Затем можно было бы доказать, что жадный выбор (процесс, который оканчивается первым в задаче S_i) в сочетании с оптимальным решением для множества S_m остальных совместимых между собой процессов приводит к оптимальному решению задачи S_i. Обобщая сказанное, опишем процесс разработки жадных алгоритмов в виде последовательности перечисленных ниже этапов.

1. Привести задачу оптимизации к виду, когда после сделанного выбора остается решить только одну подзадачу.
2. Доказать, что всегда существует такое оптимальное решение исходной задачи, которое можно получить путем жадного выбора, так что такой выбор всегда допустим.
3. Показать, что после жадного выбора остается подзадача, обладающая тем свойством, что объединение оптимального решения подзадачи со сделанным жадным выбором приводит к оптимальному решению исходной задачи.

Описанный выше упрощенный процесс будет использоваться в последующих разделах данной главы. Тем не менее, заметим, что в основе каждого жадного алгоритма почти всегда находится более сложное решение в стиле динамического программирования.

Как определить, способен ли жадный алгоритм решить стоящую перед нами задачу оптимизации? Общего пути здесь нет, однако можно выделить две основные составляющие — свойство жадного выбора и оптимальную подструктуру. Если удается продемонстрировать, что задача обладает двумя этими свойствами, то с большой вероятностью для нее можно разработать жадный алгоритм.

Свойство жадного выбора

Первый из названных выше основных составляющих жадного алгоритма — ***свойство жадного выбора***: глобальное оптимальное решение можно получить, делая локальный оптимальный (жадный) выбор. Другими словами, рассуждая по поводу того, какой выбор следует сделать, мы делаем выбор, который кажется

самым лучшим в текущей задаче; результаты возникающих подзадач при этом не рассматриваются.

Рассмотрим отличие жадных алгоритмов от динамического программирования. В динамическом программировании на каждом этапе делается выбор, однако обычно этот выбор зависит от решений подзадач. Следовательно, методом динамического программирования задачи обычно решаются в восходящем направлении, т.е. сначала обрабатываются более простые подзадачи, а затем — более сложные. В жадном алгоритме делается выбор, который выглядит в данный момент наилучшим, после чего решается подзадача, возникающая в результате этого выбора. Выбор, сделанный в жадном алгоритме, может зависеть от сделанных ранее выборов, но он не может зависеть от каких бы то ни было выборов или решений последующих подзадач. Таким образом, в отличие от динамического программирования, где подзадачи решаются в восходящем порядке, жадная стратегия обычно разворачивается в нисходящем порядке, когда жадный выбор делается один за другим, в результате чего каждый экземпляр текущей задачи сводится к более простому.

Конечно же, необходимо доказать, что жадный выбор на каждом этапе приводит к глобальному оптимальному решению, и здесь потребуются определенные интеллектуальные усилия. Обычно, как это было в теореме 16.1, в таком доказательстве исследуется глобальное оптимальное решение некоторой подзадачи. Затем демонстрируется, что решение можно преобразовать так, чтобы в нем использовался жадный выбор, в результате чего получится аналогичная, но более простая подзадача.

Свойство жадного выбора часто дает определенное преимущество, позволяющее повысить эффективность выбора в подзадаче. Например, если в задаче о выборе процессов предварительно отсортировать процессы в порядке монотонного возрастания моментов их окончания, то каждый из них достаточно рассмотреть всего один раз. Зачастую оказывается, что благодаря предварительной обработке входных данных или применению подходящей структуры данных (нередко это очередь с приоритетами) можно ускорить процесс жадного выбора, что приведет к повышению эффективности алгоритма.

Оптимальная подструктура

Оптимальная подструктура проявляется в задаче, если в ее оптимальном решении содержатся оптимальные решения подзадач. Это свойство является основным признаком применимости как динамического программирования, так и жадных алгоритмов. В качестве примера оптимальной подструктуры напомним результаты раздела 16.1. В нем было продемонстрировано, что если оптимальное решение подзадачи S_{ij} содержит процесс a_k, то оно также содержит оптимальные решение подзадач S_{ik} и S_{kj}. После выявления этой оптимальной подструктуры

было показано, что если известно, какой процесс используется в качестве процесса a_k, то оптимальное решение задачи S_{ij} можно построить путем выбора процесса a_k и объединения его со всеми процессами в оптимальных решениях подзадач S_{ik} и S_{kj}. На основе этого наблюдения удалось получить рекуррентное соотношение (16.3), описывающее оптимальное решение.

Обычно при работе с жадными алгоритмами применяется более простой подход. Как уже упоминалось, мы воспользовались предположением, что подзадача получилась в результате жадного выбора в исходной задаче. Все, что осталось сделать, — это обосновать, что оптимальное решение подзадачи в сочетании с ранее сделанным жадным выбором приводит к оптимальному решению исходной задачи. В этой схеме для доказательства того, что жадный выбор на каждом шаге приводит к оптимальному решению, неявно используется индукция по вспомогательным задачам.

Сравнение жадных алгоритмов и динамического программирования

Поскольку свойство оптимальной подструктуры применяется и в жадных алгоритмах, и в стратегии динамического программирования, может возникнуть соблазн разработать решение в стиле динамического программирования для задачи, в которой достаточно применить жадное решение. Не исключена также возможность ошибочного вывода о применимости жадного решения в той ситуации, когда необходимо решать задачу методом динамического программирования. Чтобы проиллюстрировать тонкие различия между этими двумя методами, рассмотрим две разновидности классической задачи оптимизации.

Дискретная задача о рюкзаке (0-1 knapsack problem) формулируется следующим образом. Вор во время ограбления магазина обнаружил n предметов. Предмет под номером i имеет стоимость v_i грн. и вес w_i кг, где и v_i, и w_i — целые числа. Нужно унести вещи, суммарная стоимость которых была бы как можно большей, однако грузоподъемность рюкзака ограничивается W килограммами, где W — целая величина. Какие предметы следует взять с собой?

Ситуация в ***непрерывной задаче о рюкзаке*** (fractional knapsack problem) та же, но теперь тот или иной товар вор может брать с собой частично, а не делать каждый раз бинарный выбор — брать или не брать (0–1). В дискретной задаче о рюкзаке в роли предметов могут выступать, например, слитки золота, а для непрерывной задачи больше подходят такие товары, как золотой песок.

В обеих разновидностях задачи о рюкзаке проявляется свойство оптимальной подструктуры. Рассмотрим в целочисленной задаче наиболее ценную загрузку, вес которой не превышает W кг. Если вынуть из рюкзака предмет под номером j, то остальные предметы должны быть наиболее ценными, вес которых не превышает $W - w_j$ и которые можно составить из $n - 1$ исходных предметов, из множества

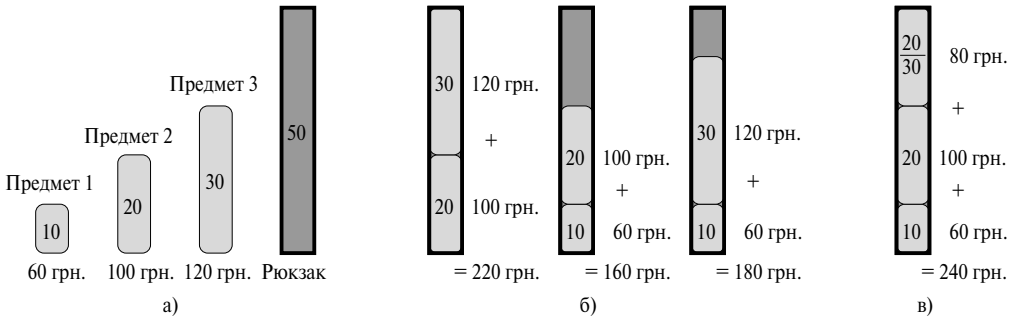


Рис. 16.2. Жадная стратегия не работает для дискретной задачи о рюкзаке

которых исключен предмет под номером j. Для аналогичной непрерывной задачи можно привести такие же рассуждения. Если удалить из оптимально загруженного рюкзака товар с индексом j, который весит w кг, остальное содержимое рюкзака будет наиболее ценным, состоящим из $n - 1$ исходных товаров, вес которых не превышает величину $W - w$ плюс $w_j - w$ кг товара с индексом j.

Несмотря на сходство сформулированных выше задач, непрерывная задача о рюкзаке допускает решение на основе жадной стратегии, а дискретная — нет. Чтобы решить непрерывную задачу, вычислим сначала стоимость единицы веса v_i/w_i каждого товара. Придерживаясь жадной стратегии, вор сначала загружает как можно больше товара с максимальной удельной стоимостью (за единицу веса). Если запас этого товара исчерпается, а грузоподъемность рюкзака — нет, он загружает как можно больше товара, удельная стоимость которого будет второй по величине. Так продолжается до тех пор, пока вес товара не достигает допустимого максимума. Таким образом, вместе с сортировкой товаров по их удельной стоимости время работы алгоритма равно $O(n \lg n)$. Доказательство того, что непрерывная задача о рюкзаке обладает свойством жадного выбора, предлагается провести в упражнении 16.2-1.

Чтобы убедиться, что подобная жадная стратегия не работает в целочисленной задаче о рюкзаке, рассмотрим пример, проиллюстрированный на рис. 16.2*а*. Имеется 3 предмета и рюкзак, способный выдержать 50 кг. Первый предмет весит 10 кг и стоит 60 грн. Второй предмет весит 20 кг и стоит 100 грн. Третий предмет весит 30 кг и стоит 120 грн. Таким образом, один килограмм первого предмета стоит 6 грн, что превышает аналогичную величину для второго (5 грн/кг) и третьего (4 грн/кг) предметов. Поэтому жадная стратегия должна состоять в том, чтобы сначала взять первый предмет. Однако, как видно из рис. 16.2*б*, оптимальное решение — взять второй и третий предмет, а первый — оставить. Оба возможных решения, включающих в себя первый предмет, не являются оптимальными.

Однако для аналогичной непрерывной задачи жадная стратегия, при которой сначала загружается первый предмет, позволяет получить оптимальное решение,

как видно из рис. 16.2*в*. Если же сначала загрузить первый предмет в дискретной задаче, то невозможно будет загрузить рюкзак до отказа и оставшееся пустое место приведет к снижению эффективной стоимости единицы веса при такой загрузке. Принимая в дискретной задаче решение о том, следует ли положить тот или иной предмет в рюкзак, необходимо сравнить решение подзадачи, в которую входит этот предмет, с решением подзадачи, в которой он отсутствует, и только после этого можно делать выбор. В сформулированной таким образом задаче возникает множество перекрывающихся подзадач, что служит признаком применимости динамического программирования. И в самом деле, целочисленную задачу о рюкзаке можно решить с помощью динамического программирования (см. упражнение 16.2-2).

Упражнения

16.2-1. Докажите, что непрерывная задача о рюкзаке обладает свойством жадного выбора.

16.2-2. Приведите решение дискретной задачи о рюкзаке с помощью динамического программирования. Время работы вашего алгоритма должно быть равно $O(nW)$, где n — количество элементов, а W — максимальный вес предметов, которые вор может положить в свой рюкзак.

16.2-3. Предположим, что в дискретной задаче о рюкзаке порядок сортировки по увеличению веса совпадает с порядком сортировки по уменьшению стоимости. Сформулируйте эффективный алгоритм для поиска оптимального решения этой разновидности задачи о рюкзаке и обоснуйте его корректность.

16.2-4. Профессор едет из Киева в Москву на автомобиле. У него есть карта, где обозначены все заправки с указанием расстояния между ними. Известно, что если топливный бак заполнен, то автомобиль может проехать n километров. Профессору хочется как можно реже останавливаться на заправках. Сформулируйте эффективный метод, позволяющий профессору определить, на каких заправках следует заливать топливо, чтобы количество остановок оказалось минимальным. Докажите, что разработанная стратегия дает оптимальное решение.

16.2-5. Разработайте эффективный алгоритм, который по заданному множеству $\{x_1, x_2, \ldots, x_n\}$ действительных точек на числовой прямой позволил бы найти минимальное множество закрытых интервалов единичной длины, содержащих все эти точки. Обоснуйте корректность алгоритма.

⋆ 16.2-6. Покажите, как решить непрерывную задачу о рюкзаке за время $O(n)$.

16.2-7. Предположим, что имеется два множества, A и B, каждое из которых состоит из n положительных целых чисел. Порядок элементов каждого

множества можно менять произвольным образом. Пусть a_i — i-й элемент множества A, а b_i — i-й элемент множества B после переупорядочения. Составим величину $\prod_{i=1}^{n} a_i^{b_i}$. Сформулируйте алгоритм, который бы максимизировал эту величину. Докажите, что ваш алгоритм действительно максимизирует указанную величину, и определите время его работы.

16.3 Коды Хаффмана

Коды Хаффмана (Huffman codes) — широко распространенный и очень эффективный метод сжатия данных, который, в зависимости от характеристик этих данных, обычно позволяет сэкономить от 20% до 90% объема. Мы рассматриваем данные, представляющие собой последовательность символов. В жадном алгоритме Хаффмана используется таблица, содержащая частоты появления тех или иных символов. С помощью этой таблицы определяется оптимальное представление каждого символа в виде бинарной строки.

Предположим, что имеется файл данных, состоящий из 100 000 символов, который требуется сжать. Символы в этом файле встречаются с частотой, представленной в табл. 16.1. Таким образом, всего файл содержит шесть различных символов, а, например, символ `a` встречается в нем 45 000 раз.

Таблица 16.1. Задача о кодировании последовательности символов

	`a`	`b`	`c`	`d`	`e`	`f`
Частота (в тысячах)	45	13	12	16	9	5
Кодовое слово фиксированной длины	000	001	010	011	100	101
Кодовое слово переменной длины	0	101	100	111	1101	1100

Существует множество способов представить подобный файл данных. Рассмотрим задачу по разработке ***бинарного кода*** символов (binary character code; или для краткости просто ***кода***), в котором каждый символ представляется уникальной бинарной строкой. Если используется ***код фиксированной длины***, или ***равномерный код*** (fixed-length code), то для представления шести символов понадобится 3 бита: $a = 000$, $b = 001$, ..., $f = 101$. При использовании такого метода для кодирования всего файла понадобится 300 000 битов. Можно ли добиться лучших результатов?

С помощью ***кода переменной длины***, или ***неравномерного кода*** (variable-length code), удается получить значительно лучшие результаты, чем с помощью кода фиксированной длины. Это достигается за счет того, что часто встречающимся символам сопоставляются короткие кодовые слова, а редко встречающимся — длинные. Такой код представлен в последней строке табл. 16.1. В нем символ `a`

представлен 1-битовой строкой `0`, а символ `f` — 4-битовой строкой `1100`. Для представления файла с помощью этого кода потребуется

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224\,000 \text{ битов.}$$

Благодаря этому дополнительно экономится 25% объема. Кстати, как мы вскоре убедимся, для рассматриваемого файла это оптимальная кодировка символов.

Префиксные коды

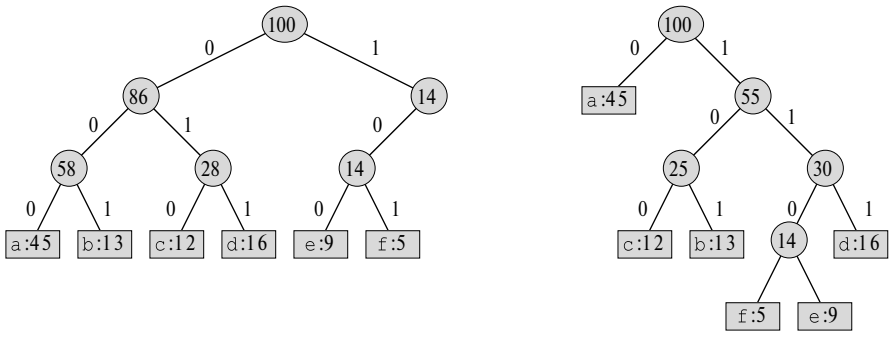
Мы рассматриваем только те коды, в которых никакое кодовое слово не является префиксом какого-то другого кодового слова. Такие коды называются ***префиксными*** (prefix codes)[2]. Можно показать (хотя здесь мы не станем этого делать), что оптимальное сжатие данных, которого можно достичь с помощью кодов, всегда достижимо при использовании префиксного кода, поэтому рассмотрение одних лишь префиксных кодов не приводит к потере общности.

Для любого бинарного кода символов кодирование текста — очень простой процесс: надо просто соединить кодовые слова, представляющие каждый символ в файле. Например, в кодировке с помощью префиксного кода переменной длины, представленного в табл. 16.1, трехсимвольный файл `abc` имеет вид $0 \cdot 101 \cdot 100 = 0101100$, где символом "$\cdot$" обозначена операция конкатенации.

Предпочтение префиксным кодам отдается из-за того, что они упрощают декодирование. Поскольку никакое кодовое слово не выступает в роли префикса другого, кодовое слово, с которого начинается закодированный файл, определяется однозначно. Начальное кодовое слово легко идентифицировать, преобразовать его в исходный символ и продолжить декодирование оставшейся части закодированного файла. В рассматриваемом примере строка 001011101 однозначно раскладывается на подстроки $0 \cdot 0 \cdot 101 \cdot 1101$, что декодируется как `aabe`.

Для упрощения процесса декодирования требуется удобное представление префиксного кода, чтобы кодовое слово можно было легко идентифицировать. Одним из таких представлений является бинарное дерево, листьями которого являются кодируемые символы. Бинарное кодовое слово, представляющее символ, интерпретируется как путь от корня к этому символу. В такой интерпретации 0 означает "перейти к левому дочернему узлу", а 1 — "перейти к правому дочернему узлу". На рис. 16.3 показаны такие деревья для двух кодов, взятых из нашего примера. Каждый лист на рисунке помечен соответствующим ему символом и частотой появления, а внутренний узел — суммой частот листьев его поддерева. В части *а* рисунка приведено дерево, соответствующее коду фиксированной длины, где $a = 000, \ldots, f = 101$. В части *б* показано дерево, соответствующее оптимальному префиксному коду $a = 0, b = 101, \ldots, f = 1100$. Заметим,

[2]Возможно, лучше подошло бы название "беспрефиксные коды", однако "префиксные коды" — стандартный в литературе термин.



а) б)

Рис. 16.3. Деревья, соответствующие схемам кодирования, приведенным в табл. 16.1

что изображенные на рисунке деревья не являются бинарными деревьями поиска, поскольку листья в них не обязательно расположены в порядке сортировки, а внутренние узлы не содержат ключей символов.

Оптимальный код файла всегда может быть представлен *полным* бинарным деревом, в котором у каждого узла (кроме листьев) имеется по два дочерних узла (см. упражнение 16.3-1). Код фиксированной длины, представленный в рассмотренном примере, не является оптимальным, поскольку соответствующее ему дерево, изображенное на рис. 16.3*а*, — неполное бинарное дерево: некоторые слова кода начинаются с `10...`, но ни одно из них не начинается с `11...`. Поскольку в нашем обсуждении мы можем ограничиться только полными бинарными деревьями, можно утверждать, что если C — алфавит, из которого извлекаются кодируемые символы, и все частоты, с которыми встречаются символы, положительны, то дерево, представляющее оптимальный префиксный код, содержит ровно $|C|$ листьев, по одному для каждого символа из множества C, и ровно $|C| - 1$ внутренних узлов (см. упражнение Б.5-3).

Если имеется дерево T, соответствующее префиксному коду, легко подсчитать количество битов, которые потребуются для кодирования файла. Обозначим через $f(c)$ частоту символа c из множества C в файле, а через $d_T(c)$ — глубину листа, представляющего этот символ в дереве. Заметим, что $d_T(c)$ является также длиной слова, кодирующего символ c. Таким образом, для кодировки файла необходимо количество битов, равное

$$B(T) = \sum_{c \in C} f(c)\, d_T(c) \tag{16.5}$$

Назовем эту величину ***стоимостью*** дерева T.

Построение кода Хаффмана

Хаффман изобрел жадный алгоритм, позволяющий составить оптимальный префиксный код, который получил название ***код Хаффмана***. В соответствии с линией, намеченной в разделе 16.2, доказательство корректности этого алгоритма основывается на свойстве жадного выбора и оптимальной подструктуре. Вместо того чтобы демонстрировать, что эти свойства выполняются, а затем разрабатывать псевдокод, сначала мы представим псевдокод. Это поможет прояснить, как алгоритм осуществляет жадный выбор.

В приведенном ниже псевдокоде предполагается, что C — множество, состоящее из n символов, и что каждый из символов $c \in C$ — объект с определенной частотой $f(c)$. В алгоритме строится дерево T, соответствующее оптимальному коду, причем построение идет в восходящем направлении. Процесс построения начинается с множества, состоящего из $|C|$ листьев, после чего последовательно выполняется $|C| - 1$ операций "слияния", в результате которых образуется конечное дерево. Для идентификации двух наименее часто встречающихся объектов, подлежащих слиянию, используется очередь с приоритетами Q, ключами в которой являются частоты f. В результате слияния двух объектов образуется новый объект, частота появления которого является суммой частот объединенных объектов:

```
HUFFMAN(C)
1  n ← |C|
2  Q ← C
3  for i ← 1 to n − 1
4       do Выделить память для узла z
5          left[z] ← x ← EXTRACT_MIN(Q)
6          right[z] ← y ← EXTRACT_MIN(Q)
7          f[z] ← f[x] + f[y]
8          INSERT(Q, z)
9  return EXTRACT_MIN(Q)              ▷ Возврат корня дерева
```

Для рассмотренного ранее примера алгоритм Хаффмана выводит результат, приведенный на рис. 16.4. На каждом этапе показано содержимое очереди, элементы которой рассортированы в порядке возрастания их частот. На каждом шаге работы алгоритма объединяются два объекта (дерева) с самыми низкими частотами. Листья изображены в виде прямоугольников, в каждом из которых указана буква и соответствующая ей частота. Внутренние узлы представлены кругами, содержащими сумму частот дочерних узлов. Ребро, соединяющее внутренний узел с левым дочерним узлом, имеет метку 0, а ребро, соединяющее его с правым дочерним узлом, — метку 1. Слово кода для буквы образуется последовательностью меток на ребрах, соединяющих корень с листом, представляющим эту букву. По-

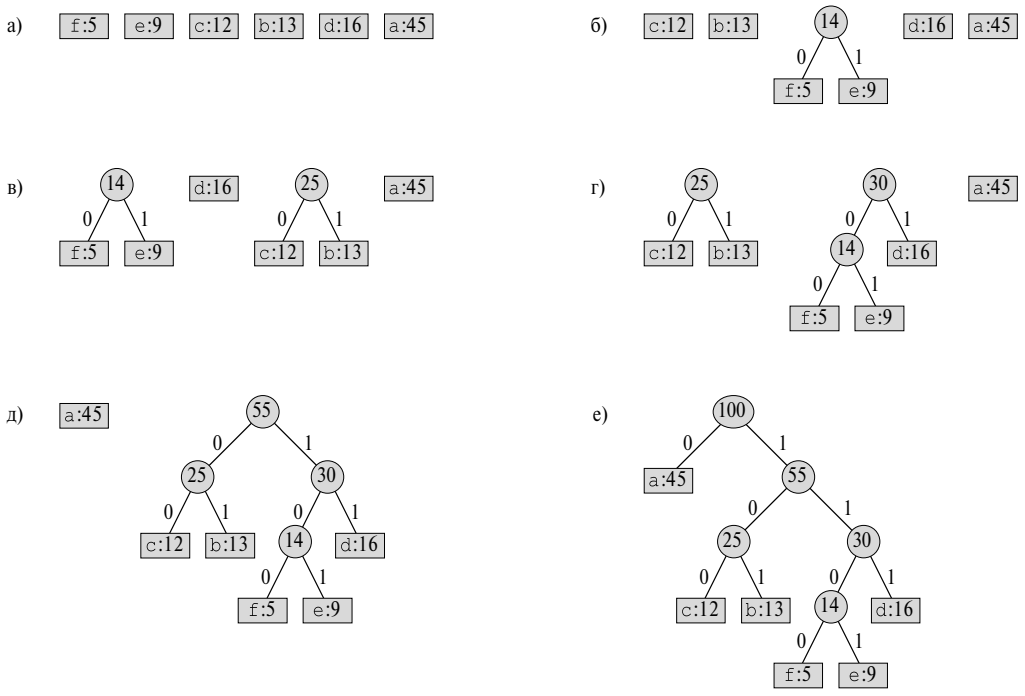


Рис. 16.4. Этапы работы алгоритма Хаффмана для частот, заданных в табл. 16.1

скольку данное множество содержит шесть букв, размер исходной очереди равен 6 (часть *а* рисунка), а для построения дерева требуется пять слияний. Промежуточные этапы изображены в частях *б*–*д*. Конечное дерево (рис. 16.4*е*) представляет оптимальный префиксный код. Как уже говорилось, слово кода для буквы — это последовательность меток на пути от корня к листу с этой буквой.

В строке 2 инициализируется очередь с приоритетами Q, состоящая из элементов множества C. Цикл **for** в строках 3–8 поочередно извлекает по два узла, x и y, которые характеризуются в очереди наименьшими частотами, и заменяет их в очереди новым узлом z, представляющим объединение упомянутых выше элементов. Частота появления z вычисляется в строке 7 как сумма частот x и y. Узел x является левым дочерним узлом z, а y — его правым дочерним узлом. (Этот порядок является произвольным; перестановка левого и правого дочерних узлов приводит к созданию другого кода с той же стоимостью.) После $n - 1$ объединений в очереди остается один узел — корень дерева кодов, который возвращается в строке 9.

При анализе времени работы алгоритма Хаффмана предполагается, что Q реализована как бинарная неубывающая пирамида (см. главу 6). Для множества C, состоящего из n символов, инициализацию очереди Q в строке 2 можно выполнить

за время $O(n)$ с помощью процедуры BUILD_MIN_HEAP из раздела 6.3. Цикл **for** в строках 3–8 выполняется ровно $n-1$ раз, и поскольку для каждой операции над пирамидой требуется время $O(\lg n)$, вклад цикла во время работы алгоритма равен $O(n \lg n)$. Таким образом, полное время работы процедуры HUFFMAN с входным множеством, состоящим из n символов, равно $O(n \lg n)$.

Корректность алгоритма Хаффмана

Чтобы доказать корректность жадного алгоритма HUFFMAN, покажем, что в задаче о построении оптимального префиксного кода проявляются свойства жадного выбора и оптимальной подструктуры. В сформулированной ниже лемме показано соблюдение свойства жадного выбора.

Лемма 16.2. Пусть C — алфавит, каждый символ $c \in C$ которого встречается с частотой $f[c]$. Пусть x и y — два символа алфавита C с самыми низкими частотами. Тогда для алфавита C существует оптимальный префиксный код, кодовые слова символов x и y в котором имеют одинаковую длину и отличаются лишь последним битом.

Доказательство. Идея доказательства состоит в том, чтобы взять дерево T, представляющее произвольный оптимальный префиксный код, и преобразовать его в дерево, представляющее другой оптимальный префиксный код, в котором символы x и y являются листьями с общим родительским узлом, причем в новом дереве эти листья находятся на максимальной глубине.

Пусть a и b — два символа, представленные листьями с общим родительским узлом, которые находятся на максимальной глубине дерева T. Предположим без потери общности, что $f[a] \leqslant f[b]$ и $f[x] \leqslant f[y]$. Поскольку $f[x]$ и $f[y]$ — две самые маленькие частоты (в указанном порядке), а $f[a]$ и $f[b]$ — две произвольные частоты, то выполняются соотношения $f[x] \leqslant f[a]$ и $f[y] \leqslant f[b]$. Как показано на рис. 16.5, в результате перестановки в дереве T листьев a и x получается дерево T', а при последующей перестановке в дереве T' листьев b и y получается дерево T''. Согласно уравнению (16.5), разность стоимостей деревьев T и T' равна

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c)\, d_T(c) - \sum_{c \in C} f(c)\, d_{T'}(c) = \\ &= f[x]\, d_T(x) + f[a]\, d_T(a) - f[x]\, d_{T'}(x) - f[a]\, d_{T'}(a) = \\ &= f[x]\, d_T(x) + f[a]\, d_T(a) - f[x]\, d_T(a) - f[a]\, d_T(x) = \\ &= (f[a] - f[x])\,(d_T(a) - d_T(x)) \geqslant 0, \end{aligned}$$

поскольку величины $f[a] - f[x]$ и $d_T(a) - d_T(x)$ неотрицательны. Величина $f[a] - f[x]$ неотрицательна, потому что x — лист с минимальной частотой, величина $d_T(a) - d_T(x)$ неотрицательна, потому что a — лист на максимальной глубине

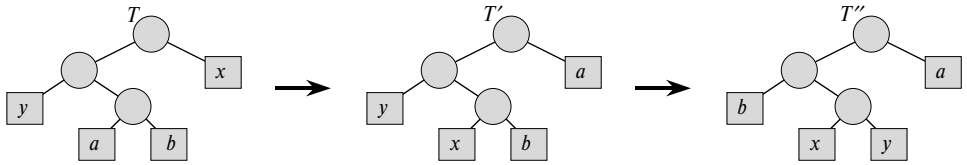


Рис. 16.5. Иллюстрация ключевых этапов доказательства леммы 16.2

в дереве T. Аналогично, перестановка листьев y и b не приведет к увеличению стоимости, поэтому величина $B(T') - B(T'')$ неотрицательна. Таким образом, выполняется неравенство $B(T'') \leqslant B(T)$, и поскольку T — оптимальное дерево, то должно также выполняться неравенство $B(T) \leqslant B(T'')$, откуда следует, что $B(T'') = B(T)$. Таким образом, T'' — оптимальное дерево, в котором x и y — находящиеся на максимальной глубине дочерние листья одного и того же узла, что и доказывает лемму. ■

Из леммы 16.2 следует, что процесс построения оптимального дерева путем объединения узлов без потери общности можно начать с жадного выбора, при котором объединению подлежат два символа с наименьшими частотами. Почему такой выбор будет жадным? Стоимость объединения можно рассматривать как сумму частот входящих в него элементов. В упражнении 16.3-3 предлагается показать, что полная стоимость сконструированного таким образом дерева равна сумме стоимостей его составляющих. Из всевозможных вариантов объединения на каждом этапе в процедуре HUFFMAN выбирается тот, в котором получается минимальная стоимость.

В приведенной ниже лемме показано, что задача о составлении оптимальных префиксных кодов обладает свойством оптимальной подструктуры.

Лемма 16.3. Пусть дан алфавит C, в котором для каждого символа $c \in C$ определены частоты $f[c]$. Пусть x и y — два символа из алфавита C с минимальными частотами. Пусть C' — алфавит, полученный из алфавита C путем удаления символов x и y и добавления нового символа z, так что $C' = C - \{x, y\} \cup \{z\}$. По определению частоты f в алфавите C' совпадают с частотами в алфавите C, за исключением частоты $f[z] = f[x] + f[y]$. Пусть T' — произвольное дерево, представляющее оптимальный префиксный код для алфавита C'. Тогда дерево T, полученное из дерева T' путем замены листа z внутренним узлом с дочерними элементами x и y, представляет оптимальный префиксный код для алфавита C.

Доказательство. Сначала покажем, что стоимость $B(T)$ дерева T можно выразить через стоимость $B(T')$ дерева T', рассматривая стоимости компонентов из уравнения (16.5). Для каждого символа $c \in C - \{x, y\}$ выполняется соотношение

$d_T(c) = d_{T'}(c)$, следовательно, $f[c]\,d_T(c) = f[c]\,d_{T'}(c)$. Поскольку $d_T(x) = d_T(y) = d_{T'}(z) + 1$, получаем соотношение

$$\begin{aligned} f[x]\,d_T(x) + f[y]\,d_T(y) &= (f[x] + f[y])\,(d_{T'}(z) + 1) = \\ &= f[z]\,d_{T'}(z) + (f[x] + f[y])\,, \end{aligned}$$

из которого следует равенство

$$B(T) = B\left(T'\right) + f[x] + f[y]\,,$$

или

$$B\left(T'\right) = B(T) - f[x] - f[y]\,.$$

Докажем лемму методом от противного. Предположим, дерево T не представляет оптимальный префиксный код для алфавита C. Тогда существует дерево T'', для которого справедливо неравенство $B(T'') < B(T)$. Согласно лемме 16.2, x и y без потери общности можно считать дочерними элементами одного и того же узла. Пусть дерево T''' получено из дерева T'' путем замены элементов x и y листом z с частотой $f[z] = f[x] + f[y]$. Тогда можно записать:

$$B\left(T'''\right) = B\left(T''\right) - f[x] - f[y] < B(T) - f[x] - f[y] = B\left(T'\right),$$

что противоречит предположению о том, что дерево T' представляет оптимальный префиксный код для алфавита C'. Таким образом, дерево T должно представлять оптимальный префиксный код для алфавита C. ■

Теорема 16.4. Процедура HUFFMAN дает оптимальный префиксный код.

Доказательство. Справедливость теоремы непосредственно следует из лемм 16.2 и 16.3. ■

Упражнения

16.3-1. Докажите, что бинарное дерево, которое не является полным, не может соответствовать оптимальному префиксному коду.

16.3-2. Определите оптимальный код Хаффмана для представленного ниже множества частот, основанного на первых восьми числах Фибоначчи.

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Попытайтесь обобщить ответ на случай первых n чисел Фибоначчи.

16.3-3. Докажите, что полную стоимость дерева, представляющего какой-нибудь код, можно также вычислить как сумму комбинаций частот двух дочерних узлов по всем внутренним узлам.

16.3-4. Докажите, что если символы отсортированы в порядке монотонного убывания частот, то существует оптимальный код, длина слов в котором — монотонно неубывающая величина.

16.3-5. Предположим, имеется оптимальный префиксный код для множества $C = \{0, 1, \ldots, n-1\}$ и мы хотим передать его, использовав как можно меньше битов. Покажите, как представить любой оптимальный префиксный код для множества C с помощью всего $2n - 1 + n \lceil \lg n \rceil$ битов. (*Указание*: для представления структуры дерева, определяемой его обходом, достаточно $2n - 1$ битов.)

16.3-6. Обобщите алгоритм Хаффмана для кодовых слов в троичной системе счисления (т.е. слов, в которых используются символы 0, 1 и 2) и докажите, что с его помощью получается оптимальный троичный код.

16.3-7. Предположим, что файл данных содержит последовательность 8-битовых символов, причем все 256 символов встречаются достаточно часто: максимальная частота превосходит минимальную менее чем в два раза. Докажите, что в этом случае кодирование Хаффмана по эффективности не превышает обычный 8-битовый код фиксированной длины.

16.3-8. Покажите, что в среднем ни одна схема сжатия не в состоянии даже на один бит сжать файл, состоящий из случайных 8-битовых символов. (*Указание*: сравните количество возможных файлов с количеством сжатых файлов.)

⋆ 16.4 Теоретические основы жадных методов

В этом разделе в общих чертах рассматривается элегантная теория жадных алгоритмов. Она оказывается полезной, когда нужно определить, когда жадный метод дает оптимальное решение. Эта теория содержит комбинаторные структуры, известные под названием “матроиды”. Несмотря на то, что рассматриваемая теория не описывает все случаи, для которых применим жадный метод (например, она не описывает задачу о выборе процессов из раздела 16.1 или задачу о кодах Хаффмана из раздела 16.3), она находит применение во многих случаях, представляющих практический интерес. Более того, эта теория быстро развивается и обобщается, находя все больше приложений. В конце данной главы приводятся ссылки на литературу, посвященную этой теме.

Матроиды

Матроид (matroid) — это упорядоченная пара $M = (S, \mathcal{I})$, удовлетворяющая сформулированным ниже условиям.

1. Множество S конечное.
2. $\mathcal{I}$ — непустое семейство подмножеств множества S (которые называются ***независимыми*** подмножествами), таких что если $B \in \mathcal{I}$ и $A \subseteq B$, то $A \in \mathcal{I}$. Если семейство $\mathcal{I}$ удовлетворяет этому свойству, то его называют ***наследственным*** (hereditary). Заметим, что пустое множество $\emptyset$ с необходимостью принадлежит семейству $\mathcal{I}$.
3. Если $A \in \mathcal{I}$, $B \in \mathcal{I}$ и $|A| < |B|$, то существует такой элемент $x \in A - B$, что $A \cup \{x\} \in \mathcal{I}$. Говорят, что структура M удовлетворяет ***свойству замены*** (exchange property).

Термин "матроид" был введен Хасслером Уитни (Hassler Whitney). Он изучал ***матричные матроиды***, элементами S которых являются строки заданной матрицы. Множество строк является независимым, если они линейно независимы в обычном смысле. Легко показать, что эта структура задает матроид (см. упражнение 16.4-2).

В качестве другого примера матроида рассмотрим графовый матроид $M_G = (S_G, \mathcal{I}_G)$, определенный ниже в терминах неориентированного графа $G = (V, E)$.

- S_G — множество E ребер графа G.
- Если A — подмножество множества E, то $A \in \mathcal{I}_G$ тогда и только тогда, когда множество A ациклическое, т.е. множество ребер A независимо тогда и только тогда, когда подграф $G_A = (V, A)$ образует лес.

Графовый матроид M_G тесно связан с задачей о минимальном остовном дереве, подробно описанном в главе 23.

Теорема 16.5. Если $G = (V, E)$ — неориентированный граф, то $M_G = (S_G, \mathcal{I}_G)$ — матроид.

Доказательство. Очевидно, что $S_G = E$ — конечное множество. Кроме того, $\mathcal{I}_G$ — наследственное семейство, поскольку подмножество леса является лесом. Другими словами, удаление ребер из ациклического множества ребер не может привести к образованию циклов.

Таким образом, осталось показать, что структура M_G удовлетворяет свойству замены. Предположим, что $G_A = (V, A)$ и $G_B = (V, B)$ — леса графа G и что $|B| > |A|$, т.е. A и B — ациклические множества ребер, и в множестве B содержится больше ребер, чем во множестве A.

Из теоремы Б.2 следует, что лес, в котором имеется k ребер, содержит ровно $|V| - k$ деревьев. (Чтобы доказать это другим путем, начнем с $|V|$ деревьев, каждое из которых содержит только одну вершину и не содержит ни одного ребра. Тогда каждое ребро, которое добавляется в лес, на единицу уменьшает

количество деревьев.) Таким образом, лес G_A содержит $|V| - |A|$ деревьев, а лес $G_B - |V| - |B|$ деревьев.

Поскольку в лесу G_B меньше деревьев, чем в лесу G_A, в нем должно содержаться некоторое дерево T, вершины которого принадлежат двум различным деревьям леса G_A. Более того, так как дерево T — связное, оно должно содержать такое ребро (u, v), что вершины u и v принадлежат разным деревьям леса G_A. Поскольку ребро (u, v) соединяет вершины двух разных деревьев леса G_A, его можно добавить в лес G_A, не образовав при этом цикла. Таким образом, структура M_G удовлетворяет свойству замены, что и завершает доказательство того, что M_G — матроид. ■

Для заданного матроида $M = (S, \mathcal{I})$ назовем элемент $x \notin A$ ***расширением*** (extension) множества $A \in \mathcal{I}$, если его можно добавить в A без нарушения независимости, т.е. x — расширение множества A, если $A \cup \{x\} \in \mathcal{I}$. В качестве примера рассмотрим графовый матроид M_G. Если A — независимое множество ребер, то ребро e является расширением множества A тогда и только тогда, когда оно не принадлежит этому множеству и его добавление в A не приведет к образованию цикла.

Если A — независимое подмножество в матроиде M, то если у него нет расширений, говорят, что A — ***максимальное*** множество. Таким образом, множество A — максимальное, если оно не содержится ни в одном большем независимом подмножестве матроида M. Сформулированное ниже свойство часто оказывается весьма полезным.

Теорема 16.6. Все максимальные независимые подмножества матроида имеют один и тот же размер.

Доказательство. Докажем теорему методом от противного. Предположим, что A — максимальное независимое подмножество матроида M и что существует другое максимальное независимое подмножество B матроида M, размер которого превышает размер подмножества A. Тогда из свойства замены следует, что множество A расширяемо до большего независимого множества $A \cup \{x\}$ за счет некоторого элемента $x \in B - A$, что противоречит предположению о максимальности множества A. ■

В качестве иллюстрации применения этой теоремы рассмотрим графовый матроид M_G связного неориентированного графа G. Каждое максимальное независимое подмножество M_G должно представлять собой свободное дерево, содержащее ровно $|V| - 1$ ребер, которые соединяют все вершины графа G. Такое дерево называется ***остовным деревом*** (spanning tree) графа G.

Говорят, что матроид $M = (S, \mathcal{I})$ взвешенный (weighted), если с ним связана весовая функция w, назначающая каждому элементу $x \in S$ строго положительный

вес $w(x)$. Весовая функция w обобщается на подмножества S путем суммирования:

$$w(A) = \sum_{x \in A} w(x)$$

для любого подмножества $A \subseteq S$. Например, если обозначить через $w(e)$ длину ребра e графового матроида M_G, то $w(A)$ — суммарная длина всех ребер, принадлежащих множеству A.

Жадные алгоритмы на взвешенном матроиде

Многие задачи, для которых жадный подход позволяет получить оптимальное решение, можно сформулировать в терминах поиска независимого подмножества с максимальным весом во взвешенном матроиде. Итак, задан взвешенный матроид $M = (S, \mathcal{I})$ и нужно найти независимое множество $A \in \mathcal{I}$, для которого величина $w(A)$ будет максимальной. Назовем такое максимальное независимое подмножество с максимально возможным весом ***оптимальным*** подмножеством матроида. Поскольку вес $w(x)$ каждого элемента $x \in S$ положителен, оптимальное подмножество всегда является максимальным независимым подмножеством, что всегда помогает сделать множество A большим, насколько это возможно.

Например, в ***задаче о минимальном остовном дереве*** (minimum-spanning-tree problem) задается связный неориентированный граф $G = (V, E)$ и функция длин w такая, что $w(e)$ — (положительная) длина ребра e. (Термин "длина" используется здесь для обозначения исходного веса, соответствующего ребру графа; термин "вес" сохранен для обозначения весов в соответствующем матроиде.) Необходимо найти подмножество ребер, которые соединяют все вершины и имеют минимальную общую длину. Чтобы представить эту проблему в виде задачи поиска оптимального подмножества матроида, рассмотрим взвешенный матроид M_G с весовой функцией w', где $w'(e) = w_0 - w(e)$, а величина w_0 превышает максимальную длину ребра. В таком взвешенном матроиде любой вес является положительным, а оптимальное подмножество представляет собой остовное дерево в исходном графе, имеющее минимальную общую длину. Точнее говоря, каждое максимальное независимое подмножество A соответствует остовному дереву, и, поскольку для любого максимального независимого подмножества A справедливо соотношение

$$w'(A) = (|V| - 1)\, w_0 - w(A),$$

то независимое подмножество, которое максимизирует величину $w'(A)$, должно минимизировать величину $w(A)$. Таким образом, любой алгоритм, позволяющий найти оптимальное подмножество A произвольного матроида, позволяет также решить задачу о минимальном остовном дереве.

В главе 23 приводится алгоритм решения задачи о минимальном остовном дереве, а здесь описывается жадный алгоритм, который работает для произвольного взвешенного матроида. В качестве входных данных в этом алгоритме выступает взвешенный матроид $M = (S, \mathcal{I})$ и связанная с ним весовая функция w. Алгоритм возвращает оптимальное подмножество A. В рассматриваемом псевдокоде компоненты матроида M обозначены как $S[M]$ и $\mathcal{I}[M]$, а весовая функция — через w. Алгоритм является жадным, поскольку все элементы $x \in S$ рассматриваются в нем в порядке монотонного убывания веса, после чего элемент добавляется в множество A, если множество $A \cup \{x\}$ — независимое.

```
GREEDY(M, w)
1  A ← ∅
2  Сортировка множества S[M] в порядке убывания w
3  for каждого x ∈ S[M] в порядке монотонного убывания веса w(x)
4       do if A ∪ {x} ∈ I[M]
5            then A ← A ∪ {x}
6  return A
```

Элементы множества S рассматриваются по очереди, в порядке монотонного убывания веса. Рассматриваемый элемент x добавляется в множество A, если он не нарушает независимость последнего. В противном случае элемент x отбрасывается. Поскольку по определению матроида пустое множество является независимым и поскольку элемент x добавляется в множество A, только если множество $A \cup \{x\}$ — независимое, подмножество A по индукции всегда независимо. Таким образом, алгоритм GREEDY всегда возвращает независимое подмножество A. Вскоре вы увидите, что A — подмножество с максимальным возможным весом, поэтому оно является оптимальным.

Время работы алгоритма GREEDY легко проанализировать. Обозначим через n величину $|S|$. Фаза сортировки длится в течение времени $O(n \lg n)$. Строка 4 выполняется ровно n раз, по одному разу для каждого элемента множества S. При каждом выполнении строки 4 требуется проверить, является ли независимым множество $A \cup \{x\}$. Если каждая подобная проверка длится в течение времени $O(f(n))$, общее время работы алгоритма составляет $O(n \lg n + nf(n))$.

Теперь докажем, что алгоритм GREEDY возвращает оптимальное подмножество.

Лемма 16.7 (Матроиды обладают свойством жадного выбора). Пусть $M = (S, \mathcal{I})$ — взвешенный матроид с весовой функцией w, и пусть множество S отсортировано в порядке монотонного убывания весов. Пусть x — первый элемент множества S, такой что множество $\{x\}$ независимо (если он существует). Если элемент x существует, то существует оптимальное подмножество A множества S, содержащее элемент x.

Доказательство. Если такого элемента x не существует, то единственным независимым подмножеством является пустое множество и доказательство закончено. В противном случае предположим, что B — произвольное непустое оптимальное подмножество. Предположим также, что $x \notin B$; в противном случае считаем, что $A = B$, и доказательство закончено.

Ни один из элементов множества B не имеет вес, больший чем $w(x)$. Чтобы продемонстрировать это, заметим, что из $y \in B$ следует, что множество $\{y\}$ независимо, поскольку $B \in \mathcal{I}$, а семейство $\mathcal{I}$ наследственное. Таким образом, благодаря выбору элемента x обеспечивается выполнение неравенства $w(x) \geqslant w(y)$ для любого элемента $y \in B$.

Построим множество A, как описано ниже. Начнем с $A = \{x\}$. В соответствии с выбором элемента x, множество A — независимое. С помощью свойства замены будем осуществлять поиск нового элемента множества B, который можно добавить в множество A, пока не станет справедливо соотношение $|A| = |B|$; при этом множество A останется независимым. Тогда $A = B - \{y\} \cup \{x\}$ для некоторого элемента $y \in B$, так что

$$w(A) = w(B) - w(y) + w(x) \geqslant w(B).$$

Поскольку множество B — оптимальное, множество A тоже должно быть оптимальным. Поскольку $x \in A$, лемма доказана. ■

Теперь покажем, что если какой-то элемент не может быть добавлен вначале, он также не может быть добавлен позже.

Лемма 16.8. Пусть $M = (S, \mathcal{I})$ — произвольный матроид. Если x — элемент S, представляющий собой расширение некоторого независимого подмножества A множества S, то x также является расширением пустого множества $\emptyset$.

Доказательство. Поскольку x — расширение множества A, то множество $A \cup \{x\}$ независимое. Так как семейство $\mathcal{I}$ является наследственным, то множество $\{x\}$ должно быть независимым. Таким образом, x — расширение пустого множества $\emptyset$. ■

Следствие 16.9. Пусть $M = (S, \mathcal{I})$ — произвольный матроид. Если x — элемент множества S, который не является расширением пустого множества $\emptyset$, то этот элемент также не является расширением любого независимого подмножества A множества S.

Доказательство. Это следствие — обращение леммы 16.8. ■

В следствии 16.9 утверждается, что любой элемент, который не может быть использован сразу, не может использоваться никогда. Таким образом, в алгоритме

GREEDY не может быть допущена ошибка, состоящая в пропуске какого-нибудь начального элемента из множества S, который не является расширением пустого множества $\emptyset$, поскольку такие элементы никогда не могут быть использованы.

Лемма 16.10 (Матроиды обладают свойством оптимальной подструктуры). Пусть x — первый элемент множества S, выбранный алгоритмом GREEDY для взвешенного матроида $M = (S, \mathcal{I})$. Оставшаяся задача поиска независимого подмножества с максимальным весом, содержащего элемент x, сводится к поиску независимого подмножества с максимальным весом для взвешенного матроида $M' = (S', \mathcal{I}')$, где

$$S' = \{y \in S : \{x, y\} \in \mathcal{I}\},$$
$$\mathcal{I}' = \{B \subseteq S - \{x\} : B \cup \{x\} \in \mathcal{I}\}, \text{ и}$$

весовая функция матроида M' совпадает с весовой функцией матроида M, ограниченной на множество S'. (Назовем матроид M' ***сужением*** (contraction) матроида M на элемент x.)

Доказательство. Если A — произвольное независимое подмножество с максимальным весом матроида M, содержащее элемент x, то $A' = A - \{x\}$ — независимое подмножество матроида M'. Справедливо также обратное: из любого независимого подмножества A' матроида M' можно получить независимое подмножество $A = A' \cup \{x\}$ матроида M. Поскольку в обоих случаях выполняется соотношение $w(A) = w(A') + w(x)$, решение с максимальным весом, содержащее элемент x, для матроида M позволяет получить решение с максимальным весом для матроида M' и наоборот. ■

Теорема 16.11 (Корректность жадного алгоритма для матроидов). Если $M = (S, \mathcal{I})$ — взвешенный матроид с весовой функцией w, то алгоритм GREEDY(M, w) возвращает оптимальное подмножество.

Доказательство. Согласно следствию 16.9, обо всех пропущенных ранее элементах, не являющихся расширениями пустого множества $\emptyset$, можно забыть, поскольку они никогда больше не понадобятся. Когда выбран первый элемент x, из леммы 16.7 следует, что процедура GREEDY не допускает ошибки, добавляя элемент x в множество A, потому что существует оптимальное подмножество, содержащее элемент x. И наконец, из леммы 16.10 следует, что в оставшейся задаче требуется найти оптимальное подмножество матроида M', представляющего собой сужение матроида M на элемент x. После того как в процедуре GREEDY множество A приобретет вид $\{x\}$, все остальные действия этой процедуры можно интерпретировать как действия над матроидом $M' = (S', \mathcal{I}')$. Это утверждение справедливо благодаря тому, что любое множество $B \in \mathcal{I}'$ — независимое подмножество матроида M' тогда и только тогда, когда множество $B \cup \{x\}$ независимо

в матроиде M. Таким образом, в ходе последующей работы процедуры GREEDY будет найдено независимое подмножество с максимальным весом для матроида M', а в результате полного выполнения этой процедуры будет найдено независимое подмножество с максимальным весом для матроида M. ■

Упражнения

16.4-1. Покажите, что если S — произвольное конечное множество, а $\mathcal{I}_k$ — множество всех подмножеств S, размер которых не превышает $k \leqslant |S|$, то $(S, \mathcal{I}_k)$ — матроид.

⋆ 16.4-2. Пусть T — матрица размером $m \times n$ над некоторым полем (например, действительных чисел). Покажите, что если S — множество столбцов T, а $A \in \mathcal{I}$, то $(S, \mathcal{I})$ является матроидом тогда и только тогда, когда столбцы матрицы A линейно независимы.

⋆ 16.4-3. Покажите, что если $(S, \mathcal{I})$ — матроид, то матроидом является и $(S, \mathcal{I}')$, где

$$\mathcal{I}' = \left\{A' : S - A' \text{ содержит некоторое максимальное } A \in \mathcal{I}\right\},$$

т.е. максимальные независимые множества матроида $(S, \mathcal{I}')$ представляют собой дополнения максимальных независимых множеств матроида $(S, \mathcal{I})$.

⋆ 16.4-4. Пусть S — конечное множество, а $S_1, S_2, \ldots, S_k$ — разбиение этого множества на непустые непересекающиеся подмножества. Определим структуру $(S, \mathcal{I})$ с помощью условия $\mathcal{I} = \{A : |A \cap S_i| \leqslant 1 \text{ для } i = 1, 2, \ldots, k\}$. Покажите, что $(S, \mathcal{I})$ — матроид. Другими словами, семейство всех множеств A, содержащих не более одного члена в каждом блоке разбиения, определяет независимые множества матроида.

16.4-5. Покажите, как можно преобразовать весовую функцию в задаче о взвешенном матроиде, в которой оптимальное решение представляет собой максимальное независимое подмножество *с минимальным весом*, чтобы преобразовать ее в стандартную задачу о взвешенном матроиде. Обоснуйте корректность преобразования.

⋆ 16.5 Планирование заданий

Интересная задача, которую можно решить с помощью матроидов, — задача по составлению оптимального расписания единичных заданий, выполняющихся на одном процессоре. Каждое задание характеризуется конечным сроком выпол-

нения, а также штрафом при пропуске этого срока. Эта задача кажется сложной, однако она на удивление просто решается с помощью жадного алгоритма.

Единичное задание (unit-time job) — это задание (например, компьютерная программа), для выполнения которого требуется единичный интервал времени. Если имеется конечное множество S таких заданий, ***расписание*** (schedule) для этого множества представляет собой перестановку элементов множества S, определяющую порядок их выполнения. Первое задание в расписании начинается в нулевой момент времени и заканчивается в момент времени 1, второе задание начинается в момент времени 1 и заканчивается в момент времени 2 и т.д.

Входные данные в задаче по ***планированию на одном процессоре единичных заданий, характеризующихся конечным сроком выполнения и штрафом***, имеют следующий вид:

- множество $S = \{a_1, a_2, \ldots, a_n\}$, состоящее из n единичных заданий;
- множество $d_1, d_2, \ldots, d_n$ ***конечных сроков выполнения***, представленных целыми числами $1 \leqslant d_i \leqslant n$; предполагается, что задание a_i должно завершиться к моменту времени d_i;
- множество из n неотрицательных весов или ***штрафных сумм*** $w_1, w_2, \ldots, w_n$; если задание a_i не будет выполнено к моменту времени d_i, изымается штраф w_i; если это задание будет выполнено в срок, штрафные санкции не применяются.

Для множества заданий S нужно найти расписание, минимизирующее суммарный штраф, который накладывается за все просроченные задания.

Рассмотрим произвольное расписание. Говорят, что задание в нем ***просрочено***, если оно завершается позже конечного срока выполнения. В противном случае задание ***своевременное***. Произвольное расписание всегда можно привести к ***виду с первоочередными своевременными заданиями*** (early-first form), когда своевременные задания выполняются перед просроченными. Чтобы продемонстрировать это, заметим, что если какое-нибудь своевременное задание a_i следует после некоторого просроченного задания a_j, то задания a_i и a_j можно поменять местами, причем задание a_i все равно останется своевременным, а задание a_j — просроченным.

Аналогично, справедливо утверждение, согласно которому произвольное расписание можно привести к ***каноническому виду*** (canonical form), в котором своевременные задания предшествуют просроченным и расположены в порядке монотонного возрастания конечных сроков выполнения. Для этого сначала приведем расписание к виду с первоочередными своевременными заданиями. После этого, до тех пор пока в расписании будут иметься своевременные задания a_i и a_j, которые заканчиваются в моменты времени k и $k + 1$ соответственно, но при этом $d_j < d_i$, мы будем менять их местами. Поскольку задание a_j до перестановки

было своевременным, $k+1 \leqslant d_j$. Таким образом, $k+1 < d_i$, и задание a_i остается своевременным и после перестановки. Задание a_j сдвигается на более раннее время, поэтому после перестановки оно тоже остается своевременным.

Приведенные выше рассуждения позволяют свести поиск оптимального расписания к определению множества A, состоящего из своевременных заданий в оптимальном расписании. Как только такое множество будет определено, можно будет создать фактическое расписание, включив в него элементы множества A в порядке монотонного возрастания моментов их окончания, а затем — перечислив просроченные задания $(S - A)$ в произвольном порядке. Таким образом будет получено канонически упорядоченное оптимальное расписание.

Говорят, что множество заданий A ***независимое***, если для него существует расписание, в котором отсутствуют просроченные задания. Очевидно, что множество своевременных заданий расписания образует независимое множество заданий. Обозначим через $\mathcal{I}$ семейство всех независимых множеств заданий.

Рассмотрим задачу, состоящую в определении того, является ли заданное множество заданий A независимым. Обозначим через $N_t(A)$ количество заданий множества A, конечный срок выполнения которых равен t или наступает раньше (величина t может принимать значения $0, 1, 2, \ldots, n$). Заметим, что $N_0(A) = 0$ для любого множества A.

Лемма 16.12. Для любого множества заданий A сформулированные ниже утверждения эквивалентны.

1. Множество A независимое.
2. Для всех $t = 0, 1, 2, \ldots, n$ выполняются неравенства $N_t(A) \leqslant t$.
3. Если в расписании задания из множества A расположены в порядке монотонного возрастания конечных сроков выполнения, то ни одно из них не является просроченным.

Доказательство. Очевидно, что если для некоторого t $N_t(A) > t$, то невозможно составить расписание таким образом, чтобы в множестве A не оказалось просроченных заданий, поскольку до наступления момента t остается более t незавершенных заданий. Таким образом, утверждение (1) предполагает выполнение утверждения (2). Если выполняется утверждение (2), то i-й по порядку срок завершения задания не превышает i, так что при расстановке заданий в этом порядке все сроки будут соблюдены. Наконец, из утверждения (3) тривиальным образом следует справедливость утверждения (1). ■

С помощью свойства (2) леммы 16.12 легко определить, является ли независимым заданное множество заданий (см. упражнение 16.5-2).

Задача по минимизации суммы штрафов за просроченные задания — это то же самое, что задача по максимизации суммы штрафов, которых удалось избежать благодаря своевременному выполнению заданий. Таким образом, приведенная ниже теорема гарантирует, что с помощью жадного алгоритма можно найти независимое множество заданий A с максимальной суммой штрафов.

Теорема 16.13. Если S — множество единичных заданий с конечным сроком выполнения, а $\mathcal{I}$ — семейство всех независимых множеств заданий, то соответствующая система $(S, \mathcal{I})$ — матроид.

Доказательство. Ясно, что любое подмножество независимого множества заданий тоже независимо. Чтобы доказать, что выполняется свойство замены, предположим, что B и A — независимые множества заданий, и что $|B| > |A|$. Пусть k — наибольшее t, такое что $N_t(B) \leqslant N_t(A)$ (такое значение t существует, поскольку $N_0(A) = N_0(B) = 0$.) Так как $N_n(B) = |B|$ и $N_n(A) = |A|$, но $|B| > |A|$, получается, что $k < n$, и для всех j в диапазоне $k+1 \leqslant j \leqslant n$ должно выполняться соотношение $N_j(B) > N_j(A)$. Таким образом, в множестве B содержится больше заданий с конечным сроком выполнения $k+1$, чем в множестве A. Пусть a_i — задание из множества $B - A$ с конечным сроком выполнения $k+1$, и пусть $A' = A \cup \{a_i\}$.

Теперь с помощью второго свойства леммы 16.12 покажем, что множество A' должно быть независимым. Поскольку множество A независимое, для любого $0 \leqslant t \leqslant k$ выполняется соотношение $N_t(A') = N_t(A) \leqslant t$. Для $k < t \leqslant n$, поскольку B — независимое множество, имеем $N_t(A') \leqslant N_t(B) \leqslant t$. Следовательно, множество A' независимое, что и завершает доказательство того, что $(S, \mathcal{I})$ — матроид. ■

С помощью теоремы 16.11 можно сформулировать жадный алгоритм, позволяющий найти независимое множество заданий A с максимальным весом. После этого можно будет создать оптимальное расписание, в котором элементы множества A будут играть роль своевременных заданий. Этот метод дает эффективный алгоритм планирования единичных заданий с конечным сроком выполнения и штрафом для одного процессора. Время работы этого алгоритма, в котором используется процедура GREEDY, равно $O(n^2)$, поскольку каждая из $O(n)$ проверок независимости, выполняющихся в этом алгоритме, требует времени $O(n)$ (см. упражнение 16.5-2). Более быструю реализацию этого алгоритма предлагается разработать в задаче 16-4.

В табл. 16.2 приводится пример задачи по планированию на одном процессоре единичных заданий с конечным сроком выполнения и штрафом. В этом примере жадный алгоритм выбирает задания a_1, a_2, a_3 и a_4, затем отвергает задания a_5 и a_6, и наконец выбирает задание a_7. В результате получается оптимальное рас-

Таблица 16.2. Пример задачи по планированию единичных заданий с конечным сроком выполнения и штрафом на одном процессоре

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

писание $\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle$, которому соответствует общая сумма штрафа, равная $w_5 + w_6 = 50$.

Упражнения

16.5-1. Решите задачу планирования, параметры которой приведены в табл. 16.2, но каждый штраф w_i в которой заменен величиной $80 - w_i$.

16.5-2. Покажите, как с помощью второго свойства леммы 16.12 в течение времени $O(|A|)$ определить, независимо ли заданное множество заданий A.

Задачи

16-1. **Размен монет**

Рассмотрим задачу по выдаче сдачи, сумма которой составляет n копеек, использовав как можно меньшее количество монет. Предполагается, что номинал каждой монеты выражается целым числом.

а) Опишите жадный алгоритм, в котором сдача выдается монетами номиналом 25 копеек, 10 копеек, 5 копеек и 1 копейка. Докажите, что алгоритм позволяет получить оптимальное решение.

б) Предположим, что номинал монет выражается степенями некоторого целого числа $c > 1$, т.е. номиналы представляют собой числа $c^0, c^1, \ldots, c^k$, где $k \geqslant 1$. Покажите, что жадный алгоритм всегда дает оптимальное решение.

в) Приведите пример множества номиналов монет, для которого жадный алгоритм не выдает оптимального решения. В это множество должна входить монета номиналом 1 копейка, чтобы решение существовало для любого значения n.

г) Разработайте алгоритм, позволяющий выдать сдачу с помощью любого набора монет, множество номиналов в котором состоит из k

различных номиналов. Предполагается, что один из номиналов обязательно равен 1 копейке. Время работы алгоритма должно быть равно $O(nk)$.

16-2. **Расписание, минимизирующее среднее время выполнения**

Предположим, у нас имеется множество заданий $S = \{a_1, a_2, \ldots, a_n\}$, в котором для выполнения задания a_i требуется p_i единиц времени. Имеется один компьютер, на котором будут выполняться эти задания, который способен одновременно выполнять не более одного задания. Пусть c_i — ***время завершения*** задания a_i, т.е. момент времени, когда прекращается его выполнение. Задача заключается в том, чтобы минимизировать среднее время завершения, т.е. величину $\sum_{i=1}^{n} c_i/n$. Например, предположим, что имеется два задания a_1 и a_2, которым соответствуют времена выполнения $p_1 = 3$ и $p_2 = 5$. Рассмотрим расписание, в котором сначала выполняется задание a_2, а затем — задание a_1. Тогда $c_2 = 5$, $c_1 = 8$, и среднее время завершения равно $(5 + 8)/2 = 6.5$.

а) Сформулируйте алгоритм, который планирует задания таким образом, чтобы минимизировать среднее время завершения. Каждое задание должно выполняться без прерываний, т.е. будучи запущено, задание a_i должно непрерывно выполняться в течение p_i единиц времени. Докажите, что ваш алгоритм минимизирует среднее время завершения, и определите время его работы.

б) Предположим, что не все задания доступны сразу. Другими словами, с каждым заданием связано ***время выпуска*** (release time) r_i, раньше которого оно недоступно для обработки. Предположим также, что допускаются ***прерывания***, т.е. что задание может быть приостановлено и возобновлено позже. Например, задание a_i, время обработки которого равно $p_i = 6$, может быть запущено в момент времени 1 и приостановлено в момент времени 4. Затем оно может быть возобновлено в момент времени 10, еще раз приостановлено в момент времени 11, снова возобновлено в момент времени 13, и наконец завершено в момент времени 15. Таким образом, задание a_i в общей сложности выполняется в течение шести единиц времени, но время его выполнения разделено на три фрагмента. Время завершения задания a_i равно при таком расписании 15. Сформулируйте алгоритм, планирующий задания таким образом, чтобы минимизировать среднее время завершения в этом новом сценарии. Докажите, что ваш алгоритм минимизирует среднее время завершения и найдите время его работы.

16-3. **Ациклические подграфы**

а) Пусть $G = (V, E)$ — неориентированный граф. Покажите с помощью определения матроида, что $(E, \mathcal{I})$ — матроид, где $A \in \mathcal{I}$ тогда и только тогда, когда A — ациклическое подмножество E.

б) Для неориентированного графа $G = (V, E)$ ***матрицей инцидентности*** (incidence matrix) называется матрица M размером $|V| \times |E|$, такая что $M_{ve} = 1$, если ребро e инцидентно вершине v, и $M_{ve} = 0$ в противном случае. Докажите, что множество столбцов матрицы M линейно независимо над полем целых чисел по модулю 2 тогда и только тогда, когда соответствующее множество ребер ациклично. Затем с помощью результата упражнения 16.4-2 приведите альтернативное доказательство того, что структура $(E, \mathcal{I})$ из части *а* задачи — матроид.

в) Предположим, с каждым ребром неориентированного графа $G = (V, E)$ связан неотрицательный вес $w(e)$. Разработайте эффективный алгоритм поиска ациклического подмножества множества E с максимальным общим весом.

г) Пусть $G = (V, E)$ — произвольный ориентированный граф, и пусть $(E, \mathcal{I})$ определено так, что $A \in \mathcal{I}$ тогда и только тогда, когда A не содержит ориентированных циклов. Приведите пример ориентированного графа G, такого что связанная с ним система $(E, \mathcal{I})$ не является матроидом. Укажите, какое условие из определения матроида не соблюдается.

д) Для ориентированного графа $G = (V, E)$ ***матрицей инцидентности*** (incidence matrix) называется матрица M размером $|V| \times |E|$, такая что $M_{ve} = -1$, если ребро e исходит из вершины v, $M_{ve} = +1$, если ребро e входит в вершину v, и $M_{ve} = 0$ в противном случае. Докажите, что если множество столбцов матрицы M линейно независимо, то соответствующее множество ребер не содержит ориентированных циклов.

е) В упражнении 16.4-2 утверждается, что семейство линейно независимых столбцов произвольной матрицы M образует матроид. Объясните, почему результаты частей *г*) и *д*) задачи не противоречат друг другу. В чем могут быть различия между понятием ациклического множества ребер и понятием множества связанных с ними линейно независимых столбцов матрицы инцидентности?

16-4. **Вариации задачи планирования**

Рассмотрим сформулированный ниже алгоритм применительно к задаче из раздела 16.5, в которой предлагается составить расписание единичных

задач с конечными сроками выполнения и штрафами. Пусть все n отрезков времени изначально являются пустыми, где i-й интервал времени — это единичный промежуток времени, который заканчивается в момент i. Задачи рассматриваются в порядке монотонного убывания штрафов. Если при рассмотрении задания a_j существуют незаполненные отрезки времени, расположенные на шкале времени не позже конечного момента выполнения d_j этого задания, то задание a_j планируется для выполнения в течение последнего из этих отрезков, заполняя данный отрезок. Если же ни одного такого отрезка времени не осталось, задание a_j планируется для выполнения в течение последнего из незаполненных отрезков.

а) Докажите, что этот алгоритм всегда позволяет получить оптимальный ответ.

б) Реализуйте эффективный алгоритм с помощью леса непересекающихся множеств, описанного в разделе 21.3. Предполагается, что множество входных заданий уже отсортировано в порядке монотонного убывания штрафов. Проанализируйте время работы алгоритма в вашей реализации.

Заключительные замечания

Намного большее количество материала по жадным алгоритмам и матроидам можно найти в книгах Лоулера (Lawler) [196], а также Пападимитриу (Papadimitriou) и Штейглица (Steiglitz) [237].

Впервые жадный алгоритм был описан в литературе по комбинаторной оптимизации в статье Эдмондса (Edmonds) [85], опубликованной в 1971 году. Появление же теории матроидов датируется 1935 годом, когда вышла статья Уитни (Whitney) [314].

Приведенное здесь доказательство корректности жадного алгоритма для решения задачи о выборе процессов основано на доказательстве, предложенном Гаврилом (Gavril) [112]. Задача о планировании заданий изучалась Лоулером [196], Горовицем (Horowitz) и Сахни (Sahni) [157], а также Брассардом (Brassard) и Брейтли (Bratley) [47].

Коды Хаффмана были разработаны в 1952 году [162]. В работе Лелевера (Lelewer) и Хиршберга (Hirschberg) [200] имеется обзор методов сжатия данных, известных к 1987 году.

Развитие теории матроидов до теории гридоидов (greedoid) впервые было предложено Кортом (Korte) и Ловасом (Lovasz) [189–192], которые значительно обобщили представленную здесь теорию.

ГЛАВА 17

Амортизационный анализ

При ***амортизационном анализе*** (amortized analysis) время, требуемое для выполнения последовательности операций над структурой данных, усредняется по всем выполняемым операциям. Этот анализ можно использовать, например, чтобы показать, что даже если одна из операций последовательности является дорогостоящей, то при усреднении по всей последовательности средняя стоимость операций будет небольшой. Амортизационный анализ отличается от анализа средних величин тем, что в нем не учитывается вероятность. При амортизационном анализе гарантируется *средняя производительность операций в наихудшем случае*.

В первых трех разделах этой главы описываются три наиболее распространенных метода, используемые при амортизационном анализе. Начало раздела 17.1 посвящено групповому анализу, в ходе которого определяется верхняя граница $T(n)$ полной стоимости последовательности n операций. Тогда средняя стоимость операции вычисляется как $T(n)/n$. Мы принимаем среднюю стоимость равной амортизированной стоимости каждой операции, так что амортизированная стоимость всех операций одинакова.

В разделе 17.2 описывается метод бухгалтерского учета, в котором определяется амортизированная стоимость каждой операции. Если имеется несколько видов операций, то каждый из них может характеризоваться своей амортизированной стоимостью. В этом методе стоимость некоторых операций, находящихся в начальной части последовательности, может переоцениваться. Эта переоценка рассматривается как своего рода “предварительный кредит” для определенных объектов структуры данных. Впоследствии такой кредит используется для “оплаты” тех операций последовательности, которые оцениваются ниже, чем стоят на самом деле.

В разделе 17.3 обсуждается метод потенциалов, напоминающий метод бухгалтерского учета в том отношении, что в нем определяется амортизированная стоимость каждой операции, причем стоимость ранних операций последовательности может переоцениваться, чтобы впоследствии компенсировать заниженную стоимость более поздних операций. В методе потенциалов кредит поддерживается как "потенциальная энергия" структуры данных в целом, а не связывается с отдельными объектами этой структуры данных.

Все три перечисленных метода будут опробованы на двух примерах. В одном из них рассматривается стек с дополнительной операцией MULTIPOP, в ходе выполнения которой со стека одновременно снимается несколько объектов. Во втором примере реализуется бинарный счетчик, ведущий счет от нуля при помощи единственной операции — INCREMENT.

В ходе чтения этой главы следует иметь в виду, что при выполнении амортизационного анализа расходы начисляются только для анализа алгоритма, и в коде в них нет никакой необходимости. Например, если при использовании метода бухгалтерского учета объекту x приписывается какой-нибудь кредит, то в коде не нужно присваивать соответствующую величину некоторому атрибуту $credit\,[x]$.

Глубокое понимание той или иной структуры данных, возникающей в ходе амортизационного анализа, может помочь оптимизировать ее устройство. Например, в разделе 17.4 с помощью метода потенциалов проводится анализ динамически увеличивающейся и уменьшающейся таблицы.

17.1 Групповой анализ

В ходе ***группового анализа*** (aggregate analysis) исследователь показывает, что в *наихудшем случае* общее время выполнения последовательности всех n операций в сумме равно $T\,(n)$. Поэтому в наихудшем случае средняя, или ***амортизированная, стоимость*** (amortized cost), приходящаяся на одну операцию, определяется соотношением $T\,(n)/n$. Заметим, что такая амортизированная стоимость применима ко всем операциям, даже если в последовательности они представлены несколькими типами. В других двух методах, которые изучаются в этой главе, — методе бухгалтерского учета и методе потенциалов — операциям различного вида могут присваиваться разные амортизированные стоимости.

Стековые операции

В качестве первого примера группового анализа рассматриваются стеки, в которых реализована дополнительная операция. В разделе 10.1 представлены две основные стековые операции, для выполнения каждой из которых требуется время $O\,(1)$:

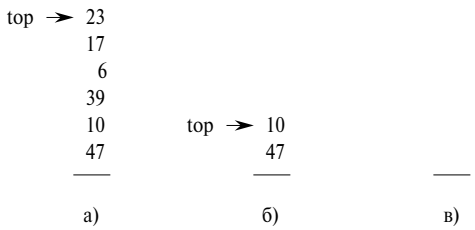


Рис. 17.1. Действие операции MULTIPOP над стеком S

PUSH(S, x) добавляет объект x в стек S;

POP(S) извлекает объект с вершины стека S и возвращает его.

Поскольку каждая из этих операций выполняется в течение времени $O(1)$, примем их длительность за единицу. Тогда полная стоимость последовательности, состоящей из n операций PUSH и POP, равна n, а фактическое время выполнения n таких операций — $\Theta(n)$.

Теперь добавим к стеку операцию MULTIPOP(S, k), извлекающую с вершины стека S k объектов (или все оставшиеся, если всего их меньше, чем k). В приведенном ниже псевдокоде операция STACK_EMPTY возвращает значение TRUE, если в данный момент в стеке нет объектов, и значение FALSE в противном случае:

```
MULTIPOP(S, k)
1  while STACK_EMPTY(S) ≠ FALSE и k ≠ 0
2        do POP(S)
3           k ← k − 1
```

Пример работы операции MULTIPOP проиллюстрирован на рис. 17.1. На рис. 17.1*а* показано начальное состояние стека. В результате выполнения операции MULTIPOP$(S, 4)$ извлекаются верхние 4 объекта (см. рис. 17.1 *б*). Следующая операция — MULTIPOP$(S, 7)$. Она опустошает стек, поскольку в нем осталось меньше семи объектов. Результирующее состояние стека показано на рис. 17.1*в*.

В течение какого времени операция MULTIPOP(S, k) выполняется над стеком, содержащем s объектов? Фактическое время работы линейно зависит от количества реально выполняемых операций POP, поэтому достаточно проанализировать процедуру MULTIPOP в терминах абстрактных единичных стоимостей каждой из операций PUSH и POP. Количество итераций цикла **while** равно количеству $\min(s, k)$ объектов, извлекаемых из стека. При каждой итерации в строке 2 выполняется однократный вызов процедуры POP. Таким образом, полная стоимость процедуры MULTIPOP равна $\min(s, k)$, а фактическое время работы является линейной функцией от этой величины.

Теперь проанализируем последовательность операций PUSH, POP и MULTIPOP, действующих на изначально пустой стек. Стоимость операции MULTIPOP в наихудшем случае равна $O(n)$, поскольку в стеке не более n объектов. Таким образом, время работы любой стековой операции в наихудшем случае равно $O(n)$, поэтому стоимость последовательности n операций равна $O\left(n^2\right)$, так как эта последовательность может содержать $O(n)$ операций MULTIPOP, стоимость каждой из которых равна $O(n)$. Но несмотря на то, что анализ проведен правильно, результат $O\left(n^2\right)$, полученный при рассмотрении наихудшей стоимости каждой операции в отдельности, является слишком грубым.

С помощью группового анализа можно получить более точную верхнюю границу при рассмотрении совокупной последовательности n операций. Фактически хотя одна операция MULTIPOP может быть весьма дорогостоящей, стоимость произвольной последовательности, состоящей из n операций PUSH, POP и MULTIPOP, которая выполняется над изначально пустым стеком, не превышает $O(n)$. Почему? Потому что каждый помещенный в стек объект можно извлечь оттуда не более одного раза. Таким образом, число вызовов операций POP (включая их вызовы в процедуре MULTIPOP) для непустого стека не может превышать количество выполненных операций PUSH, которое, в свою очередь, не больше n. При любом n для выполнения произвольной последовательности из n операций PUSH, POP и MULTIPOP требуется суммарное время $O(n)$. Таким образом, средняя стоимость операции равна $O(n)/n = O(1)$. В групповом анализе амортизированная стоимость каждой операции принимается равной ее средней стоимости, поэтому в данном примере все три стековые операции характеризуются амортизированной стоимостью, равной $O(1)$.

Еще раз стоит заметить, что хотя только что было показано, что средняя стоимость (а следовательно, и время выполнения) стековой операции равна $O(1)$, при этом не делалось никаких вероятностных рассуждений. Фактически была найдена граница времени выполнения последовательности из n операций *в наихудшем случае*, равная $O(n)$. После деления этой полной стоимости на n получается средняя стоимость одной операции, или амортизированная стоимость.

Приращение показаний бинарного счетчика

В качестве другого примера группового анализа рассмотрим задачу о реализации k-битового бинарного счетчика, который ведет счет от нуля в восходящем направлении. В качестве счетчика используется битовый массив $A[0..k-1]$, где $length[A] = k$. Младший бит хранящегося в счетчике бинарного числа x находится в элементе $A[0]$, а старший бит — в элементе $A[k-1]$, так что $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Изначально $x = 0$, т.е. $A[i] = 0$ для $i = 0, 1, \ldots, k-1$. Чтобы увеличить показания счетчика на 1 (по модулю 2^k), используется приведенная ниже процедура:

Значение счетчика	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Общая стоимость
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Рис. 17.2. Бинарный 8-битовый счетчик в процессе изменения его значения от 0 до 16 операцией INCREMENT

```
INCREMENT(A)
1  i ← 0
2  while i < length[A] и A[i] = 1
3        do A[i] ← 0
4           i ← i + 1
5  if i < length[A]
6     then A[i] ← 1
```

На рис. 17.2 показано, что происходит в бинарном счетчике при его увеличении 16 раз. Биты, значения которых изменяются при переходе к следующему этапу, заштрихованы. Справа приведена стоимость, требуемая для изменения битов. Обратите внимание, что полная стоимость никогда более чем в два раза не превышает общего количества операций INCREMENT. При этом показания счетчика возрастают от 0 до 16. В начале каждой итерации цикла **while** в строках 2–4 мы добавляем 1 к биту в позиции i. Если $A[i] = 1$, то добавление 1 обнуляет бит, который находится на позиции i, и приводит к тому, что добавление 1 будет выполнено и в позиции $i + 1$, на следующей операции цикла. В противном случае цикл оканчивается, так что если по его окончании $i < k$, то $A[i] = 0$ и нам надо изменить значение i-го бита на 1, что и делается в строке 6. Стоимость каждой операции INCREMENT линейно зависит от количества измененных битов.

Как и в примере со стеком, поверхностный анализ даст правильную, но неточную оценку. В наихудшем случае, когда массив A состоит только из единиц, для

выполнения операции INCREMENT потребуется время $\Theta(k)$. Таким образом, выполнение последовательности из n операций INCREMENT для изначально обнуленного счетчика в наихудшем случае займет время $O(nk)$.

Этот анализ можно уточнить, в результате чего для последовательности из n операций INCREMENT в наихудшем случае получается стоимость $O(n)$. Такая оценка возможна благодаря тому, что далеко не при каждом вызове процедуры INCREMENT изменяются значения всех битов. Как видно из рис. 17.2, элемент $A[0]$ изменяется при каждом вызове операции INCREMENT. Следующий по старшинству бит $A[1]$ изменяется только через раз, так что последовательность из n операций INCREMENT над изначально обнуленном счетчиком приводит к изменению элемента $A[1]$ $\lfloor n/2 \rfloor$ раз. Аналогично, бит $A[2]$ изменяется только каждый четвертый раз, т.е. $\lfloor n/4 \rfloor$ раз в последовательности из n операций INCREMENT над изначально обнуленном счетчиком. В общем случае, для $i = 0, 1, \ldots, k-1$, бит $A[i]$ изменяется $\left\lfloor n/2^i \right\rfloor$ раз в последовательности из n операций INCREMENT в изначально обнуленном счетчике. Биты же на позициях $i \geqslant k$ не изменяются. Таким образом, общее количество изменений битов при выполнении последовательности операций равно

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

(см. уравнение (A.6)). Поэтому время выполнения последовательности из n операций INCREMENT над изначально обнуленном счетчиком в наихудшем случае равно $O(n)$. Средняя стоимость каждой операции, а следовательно, и амортизированная стоимость операции, равна $O(n)/n = O(1)$.

Упражнения

17.1-1. Остается ли справедливой оценка амортизированной стоимости стековых операций, равная $O(1)$, если включить в множество стековых операций операцию MULTIPUSH, помещающую в стек k элементов?

17.1-2. Покажите, что если бы в пример с k-битовым счетчиком была включена операция DECREMENT, стоимость n операций была бы равной $\Theta(nk)$.

17.1-3. Над структурой данных выполняется n операций. Стоимость i-й по порядку операции равна i, если i — это точная степень двойки, и 1 в противном случае. Определите с помощью группового анализа амортизированную стоимость операции.

17.2 Метод бухгалтерского учета

В ***методе бухгалтерского учета*** (accounting method), применяемом при групповом анализе, разные операции оцениваются по-разному, в зависимости от их

фактической стоимости. Величина, которая начисляется на операцию, называется ***амортизированной стоимостью*** (amortized cost). Если амортизированная стоимость операции превышает ее фактическую стоимость, то соответствующая разность присваивается определенным объектам структуры данных как ***кредит*** (credit). Кредит можно использовать впоследствии для компенсирующих выплат на операции, амортизированная стоимость которых меньше их фактической стоимости. Таким образом, можно полагать, что амортизированная стоимость операции состоит из ее фактической стоимости и кредита, который либо накапливается, либо расходуется. Этот метод значительно отличается от группового анализа, в котором все операции характеризуются одинаковой амортизированной стоимостью.

При выборе амортизированной стоимости следует соблюдать осторожность. Если нужно провести анализ с использованием амортизированной стоимости, чтобы показать, что в наихудшем случае средняя стоимость операции невелика, полная амортизированная стоимость последовательности операций должна быть верхней границей полной фактической стоимости последовательности. Более того, как и в групповом анализе, это соотношение должно соблюдаться для всех последовательностей операций. Если обозначить фактическую стоимость i-й операции через c_i, а амортизированную стоимость i-й операции через $\widehat{c_i}$, то это требование для всех последовательностей, состоящих из n операций, выразится следующим образом:

$$\sum_{i=1}^{n} \widehat{c_i} \geqslant \sum_{i=1}^{n} c_i. \tag{17.1}$$

Общий кредит, хранящийся в структуре данных, представляет собой разность между полной амортизированной стоимостью и полной фактической стоимостью:

$$\sum_{i=1}^{n} \widehat{c_i} - \sum_{i=1}^{n} c_i.$$

Согласно неравенству (17.1), полный кредит, связанный со структурой данных, все время должен быть неотрицательным. Если бы полный кредит в каком-либо случае мог стать отрицательным (в результате недооценки ранних операций с надеждой восполнить счет впоследствии), то полная амортизированная стоимость в тот момент была бы ниже соответствующей фактической стоимости; значит, для последовательности операций полная амортизированная стоимость не была бы в этот момент времени верхней границей полной фактической стоимости. Таким образом, необходимо позаботиться, чтобы полный кредит для структуры данных никогда не был отрицательным.

Стековые операции

Чтобы проиллюстрировать, как производится амортизационный анализ методом бухгалтерского учета, вернемся к примеру со стеком. Напомним, что фактическая стоимость операций была такой:

PUSH 1 ,

POP 1 ,

MULTIPOP $\min(k, s)$,

где k — аргумент процедуры MULTIPOP, а s — размер стека на момент ее вызова. Присвоим приведенные ниже амортизированные стоимости:

PUSH 2 ,

POP 0 ,

MULTIPOP 0 .

Заметим, что амортизированная стоимость операции MULTIPOP равна константе (0), в то время как ее фактическая стоимость — переменная величина. В этой схеме все три амортизированные стоимости равны $O(1)$, хотя в общем случае асимптотическое поведение амортизированных стоимостей рассматриваемых операций может быть разным.

Теперь покажем, что любую последовательность стековых операций можно оплатить путем начисления амортизированных стоимостей. Предположим, что для представления каждой единицы затрат используется денежный счет. Пусть изначально стек пуст. Вспомним аналогию между стеком как структурой данных и стопкой тарелок в кафетерии, приведенную в разделе 10.1. При добавлении тарелки в стопку 1 грн. затрачивается на оплату самой операции добавления, а еще 1 грн. (из двух начисленных на операцию PUSH) остается в запасе, как бы на этой тарелке. В любой момент времени каждой тарелке стека соответствует 1 грн. кредита.

Хранящаяся на тарелке 1 грн. предназначена для уплаты стоимости извлечения ее из стека. На операцию POP ничего не начисляется, а ее фактическая стоимость выплачивается за счет кредита, который хранится в стеке. При извлечении тарелки изымается 1 грн. кредита, которая используется для уплаты фактической стоимости операции. Таким образом, благодаря небольшой переоценке операции PUSH, отпадает необходимость начислять какую-либо сумму на операцию POP.

Более того, на операцию MULTIPOP также не нужно начислять никакой суммы. При извлечении первой тарелки мы берем из нее 1 грн. кредита и тратим ее на оплату фактической стоимости операции POP. При извлечении второй тарелки на ней также есть 1 грн. кредита на оплату фактической стоимости операции POP и т.д. Таким образом, начисленной заранее суммы всегда достаточно для уплаты операций MULTIPOP. Другими словами, поскольку с каждой тарелкой стека

связана 1 грн. кредита и в стеке всегда содержится неотрицательное количество тарелок, можно быть уверенным, что сумма кредита всегда неотрицательна. Таким образом, для любой последовательности из n операций PUSH, POP и MULTIPOP полная амортизированная стоимость является верхней границей полной фактической стоимости. Поскольку полная амортизированная стоимость равна $O(n)$, полная фактическая стоимость тоже определяется этим значением.

Приращение показаний бинарного счетчика

В качестве другого примера, иллюстрирующего метод бухгалтерского учета, проанализируем операцию INCREMENT, которая выполняется над бинарным изначально обнуленным счетчиком. Ранее мы убедились, что время выполнения этой операции пропорционально количеству битов, изменяющих свое значение. В данном примере это количество используется в качестве стоимости. Как и в предыдущем примере, для представления каждой единицы затрат (в данном случае — изменения битов) будет использован денежный счет.

Чтобы провести амортизационный анализ, начислим на операцию, при которой биту присваивается значение 1 (т.е. бит устанавливается), амортизированную стоимость, равную 2 грн. Когда бит устанавливается, 1 грн. (из двух начисленных) расходуется на оплату операции по самой установке. Оставшаяся 1 грн. вкладывается в этот бит в качестве кредита для последующего использования при его обнулении. В любой момент времени с каждой единицей содержащегося в счетчике значения связана 1 грн. кредита, поэтому для обнуления бита нет необходимости начислять какую-либо сумму; за сброс бита достаточно будет уплатить 1 грн.

Теперь можно определить амортизированную стоимость операции INCREMENT. Стоимость обнуления битов в цикле **while** выплачивается за счет тех денег, которые связаны с этими битами. В процедуре INCREMENT устанавливается не более одного бита (в строке 6), поэтому амортизированная стоимость операции INCREMENT не превышает 2 грн. Количество единиц в бинарном числе, представляющем показания счетчика, не может быть отрицательным, поэтому сумма кредита тоже всегда неотрицательна. Таким образом, полная амортизированная стоимость n операций INCREMENT равна $O(n)$. Это и есть оценка полной фактической стоимости.

Упражнения

17.2-1. Над стеком выполняется последовательность операций; размер стека при этом никогда не превышает k. После каждых k операций производится резервное копирование стека. Присвоив различным стековым операциям соответствующие амортизированные стоимости, покажите, что стоимость n стековых операций, включая копирование стека, равна $O(n)$.

17.2-2. Выполните упражнение 17.1-3, применив для анализа метод бухгалтерского учета.

17.2-3. Предположим, что нам нужно не только иметь возможность увеличивать показания счетчика, но и сбрасывать его (т.е. делать так, чтобы значения всех битов были равны 0). Покажите, как осуществить реализацию счетчика в виде массива битов, чтобы для выполнения произвольной последовательности из n операций INCREMENT и RESET над изначально обнуленном счетчиком потребовалось бы время $O(n)$. (*Указание*: отслеживайте в счетчике самый старший разряд, содержащий единицу.)

17.3 Метод потенциалов

Вместо представления предоплаченной работы в виде кредита, хранящегося в структуре данных вместе с отдельными объектами, при амортизированном анализе по ***методу потенциалов*** (potential method) такая работа представляется в виде "потенциальной энергии", или просто "потенциала", который можно высвободить для оплаты последующих операций. Этот потенциал связан со структурой данных в целом, а не с ее отдельными объектами.

Метод потенциалов работает следующим образом. Мы начинаем с исходной структуры данных D_0, над которой выполняется n операций. Для всех $i = 1, 2, \ldots, n$ обозначим через c_i фактическую стоимость i-й операции, а через D_i — структуру данных, которая получается в результате применения i-й операции к структуре данных D_{i-1}. ***Потенциальная функция*** (potential function) Φ отображает каждую структуру данных D_i на действительное число $\Phi(D_i)$, которое является ***потенциалом*** (potential), связанным со структурой данных D_i. ***Амортизированная стоимость*** (amortized cost) $\widehat{c}_i$ i-й операции определяется соотношением

$$\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}). \tag{17.2}$$

Поэтому амортизированная стоимость каждой операции — это ее фактическая стоимость плюс приращение потенциала в результате выполнения операции. Согласно уравнению (17.2), полная амортизированная стоимость n операций равна

$$\sum_{i=1}^{n} \widehat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0). \tag{17.3}$$

Второе равенство следует из уравнения (A.9), поскольку слагаемые $\Phi(D_i)$ — так называемые телескопические.

Если потенциальную функцию Φ можно определить таким образом, чтобы выполнялось неравенство $\Phi(D_n) \geqslant \Phi(D_0)$, то полная амортизированная стоимость $\sum_{i=1}^{n} \widehat{c}_i$ является верхней границей полной фактической стоимости $\sum_{i=1}^{n} c_i$. На

практике не всегда известно, сколько операций может быть выполнено, поэтому, если наложить условие $\Phi(D_i) \geqslant \Phi(D_0)$ для всех i, то, как и в методе бухгалтерского учета, будет обеспечена предоплата. Часто удобно определить величину $\Phi(D_0)$ равной нулю, а затем показать, что для всех i выполняется неравенство $\Phi(D_i) \geqslant 0$. (См. упражнение 17.3-1, в котором идет речь о простом способе справиться со случаями, когда $\Phi(D_0) \neq 0$.)

Интуитивно понятно, что если разность потенциалов $\Phi(D_i) - \Phi(D_{i-1})$ для i-й операции положительна, то амортизированная стоимость $\widehat{c_i}$ представляет переоценку i-й операции и потенциал структуры данных возрастает. Если разность потенциалов отрицательна, то амортизированная стоимость представляет недооценку i-й операции и фактическая стоимость операции выплачивается за счет уменьшения потенциала.

Амортизированные стоимости, определенные уравнениями (17.2) и (17.3), зависят от выбора потенциальной функции Φ. Амортизированные стоимости, соответствующие разным потенциальным функциям, могут различаться, при этом все равно оставаясь верхними границами фактических стоимостей. При выборе потенциальных функций часто допустимы компромиссы; то, какую потенциальную функцию лучше всего использовать, зависит от оценки времени, которую нужно получить.

Стековые операции

Чтобы проиллюстрировать метод потенциала, еще раз обратимся к примеру со стековыми операциями PUSH, POP и MULTIPOP. Определим потенциальную функцию Φ для стека как количество объектов в этом стеке. Для пустого стека D_0, с которого мы начинаем, выполняется соотношение $\Phi(D_0) = 0$. Поскольку количество объектов в стеке не может быть отрицательным, стеку D_i, полученному в результате выполнения i-й операции, соответствует неотрицательный потенциал; следовательно,

$$\Phi(D_i) \geqslant 0 = \Phi(D_0).$$

Таким образом, полная амортизированная стоимость n операций, связанная с функцией Φ, представляет собой верхнюю границу фактической стоимости.

Теперь вычислим амортизированные стоимости различных стековых операций. Если i-я операция над стеком, содержащим s объектов, — операция PUSH, то разность потенциалов равна

$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1.$$

Согласно уравнению (17.2), амортизированная стоимость операции PUSH равна

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2.$$

Предположим, что i-я операция над стеком — операция MULTIPOP(S, k), и что из него извлекается $k' = \min(k, s)$ объектов. Фактическая стоимость этой операции равна k', а разность потенциалов —

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'.$$

Таким образом, амортизированная стоимость операции MULTIPOP равна

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0.$$

Аналогично можно показать, что амортизированная стоимость операции POP также равна 0.

Амортизированная стоимость каждой из трех операций равна $O(1)$, поэтому полная амортизированная стоимость последовательности из n операций равна $O(n)$. Мы уже доказали, что $\Phi(D_i) \geqslant \Phi(D_0)$, поэтому полная амортизированная стоимость n операций является верхней границей полной фактической стоимости. Поэтому в наихудшем случае стоимость n операций равна $O(n)$.

Увеличение показаний бинарного счетчика

В качестве другого примера метода потенциалов снова обратимся к задаче о приращении показаний бинарного счетчика. На этот раз определим потенциал счетчика после выполнения i-й операции INCREMENT как количество b_i содержащихся в счетчике единиц после этой операции.

Вычислим амортизированную стоимость операции INCREMENT. Предположим, что i-я операция INCREMENT обнуляет t_i битов. В таком случае фактическая стоимость этой операции не превышает значения $t_i + 1$, поскольку вдобавок к обнулению t_i битов значение 1 присваивается не более чем одному биту. Если $b_i = 0$, то в ходе выполнения i-й операции обнуляются все k битов, так что $b_{i-1} = t_i = k$. Если $b_i > 0$, то выполняется соотношение $b_i = b_{i-1} - t_i + 1$. В любом случае, справедливо неравенство $b_i \leqslant b_{i-1} - t_i + 1$ и разность потенциалов равна

$$\Phi(D_i) - \Phi(D_{i-1}) \leqslant (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i.$$

Таким образом, амортизированная стоимость определяется соотношением

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leqslant (t_i + 1) + (1 - t_i) = 2.$$

Если вначале показания счетчика равны нулю, то $\Phi(D_0) = 0$. Поскольку при всех i справедливо неравенство $\Phi(D_i) \geqslant 0$, то полная амортизированная стоимость последовательности из n операций INCREMENT является верхней границей полной фактической стоимости, поэтому в наихудшем случае стоимость n операций INCREMENT равна $O(n)$.

Метод потенциалов предоставляет простой способ анализа счетчика даже в том случае, когда отсчет начинается не с нуля. Пусть изначально показание счетчика содержит b_0 единиц, а после выполнения n операций INCREMENT — b_n единиц, где $0 \leqslant b_0, b_n \leqslant k$. (Напомним, что k — количество битов в счетчике.) Тогда уравнение (17.3) можно переписать следующим образом:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \widehat{c_i} - \Phi(D_n) + \Phi(D_0). \tag{17.4}$$

При всех $1 \leqslant i \leqslant n$ выполняется неравенство $\widehat{c_i} \leqslant 2$. В силу соотношений $\Phi(D_0) = b_0$ и $\Phi(D_n) = b_n$ полная фактическая стоимость n операций INCREMENT равна

$$\sum_{i=1}^{n} c_i \leqslant \sum_{i=1}^{n} 2 - b_n + b_0 = 2n - b_n + b_0.$$

В частности, заметим, что поскольку $b_0 \leqslant k$, то при $k = O(n)$ полная фактическая стоимость равна $O(n)$. Другими словами, если выполняется не менее $n = \Omega(k)$ операций INCREMENT, полная фактическая стоимость независимо от начального показания счетчика равна $O(n)$.

Упражнения

17.3-1. Предположим, задана такая потенциальная функция Φ, что при всех i выполняется неравенство $\Phi(D_i) \geqslant \Phi(D_0)$, но $\Phi(D_0) \neq 0$. Покажите, что существует потенциальная функция Φ' такая, что $\Phi'(D_0) = 0$ и $\Phi'(D_i) \geqslant 0$ для всех $i \geqslant 1$, и соответствующая функции Φ' амортизированная стоимость такая же, как и для функции Φ.

17.3-2. Еще раз выполните упражнение 17.1-3 с помощью метода потенциалов.

17.3-3. Рассмотрим n-элементную бинарную неубывающую пирамиду. Пусть в этой структуре данных поддерживаются операции INSERT и EXTRACT_MIN, для выполнения которых в наихудшем случае требуется время $O(\lg n)$. Определите потенциальную функцию Φ такую, что амортизированная стоимость операции INSERT равна $O(\lg n)$, а амортизированная стоимость операции EXTRACT_MIN — $O(1)$, и покажите, что она работает.

17.3-4. Чему равна полная стоимость выполнения n стековых операций PUSH, POP и MULTIPOP, если предположить, что в начале стек содержит s_0 объектов, а в конце — s_n объектов?

17.3-5. Предположим, что изначально показание счетчика не равно нулю, а определяется числом, содержащим в двоичном представлении b единиц. Покажите, что стоимость выполнения n операций INCREMENT равна $O(n)$ при условии, что $n = \Omega(b)$. (Не следует предполагать, что b — константа.)

17.3-6. Покажите, как реализовать очередь на основе двух обычных стеков (см. упражнение 10.1-6) таким образом, чтобы амортизированная стоимость каждой из операций ENQUEUE и DEQUEUE была равна $O(1)$.

17.3-7. Разработайте структуру данных для поддержки следующих двух операций над динамическим множеством целых чисел S:

INSERT(S, x) — вставка объекта x в S;

DELETE_LARGER_HALF(S) — удаление $\lceil |S|/2 \rceil$ наибольших элементов из множества S.

Поясните, как реализовать эту структуру данных, чтобы любая последовательность из m операций выполнялась за время $O(m)$.

17.4 Динамические таблицы

В некоторых приложениях заранее не известно, сколько элементов будет храниться в таблице. Может возникнуть ситуация, когда для таблицы выделяется место, а впоследствии оказывается, что его недостаточно. В этом случае приходится выделять больший объем памяти и копировать все объекты из исходной таблицы в новую, большего размера. Аналогично, если из таблицы удаляется много объектов, может понадобиться преобразовать ее в таблицу меньшего размера. В этом разделе исследуется задача о динамическом расширении и сжатии таблицы. Методом амортизационного анализа будет показано, что амортизированная стоимость вставки и удаления равна всего лишь $O(1)$, даже если фактическая стоимость операции больше из-за того, что она приводит к расширению или сжатию таблицы. Более того, мы выясним, как обеспечить соблюдение условия, при котором неиспользованное пространство динамической таблицы не превышает фиксированную долю ее полного пространства.

Предполагается, что в динамической таблице поддерживаются операции TABLE_INSERT и TABLE_DELETE. В результате выполнения операции TABLE_INSERT в таблицу добавляется элемент, занимающий одну ***ячейку*** (slot), — пространство для одного элемента. Аналогично, операцию TABLE_DELETE можно представлять как удаление элемента из таблицы, в результате чего освобождается одна ячейка. Подробности метода, с помощью которого организуется таблица, не важны; можно воспользоваться стеком (раздел 10.1), пирамидой (глава 6) или хеш-таблицей (глава 11). Для реализации хранилища объектов можно также использовать массив или набор массивов, как это было сделано в разделе 10.3.

Оказывается, достаточно удобно применить концепцию, введенную при анализе хеширования (глава 11). Определим ***коэффициент заполнения*** (load factor) $\alpha(T)$ непустой таблицы T как количество хранящихся в них элементов, деленное на ее размер (измеряемый в количестве ячеек). Размер пустой таблицы (в которой нет ни одного элемента) примем равным нулю, а ее коэффициент заполнения — единице. Если коэффициент заполнения динамической таблицы ограничен снизу

константой, ее неиспользованное пространство никогда не превышает фиксированной части ее полного размера.

Начнем анализ с динамической таблицы, в которую выполняются только вставки. Впоследствии будет рассмотрен более общий случай, когда разрешены и вставки, и удаления.

17.4.1 Расширение таблицы

Предположим, что место для хранения таблицы выделяется в виде массива ячеек. Таблица становится заполненной, когда заполняются все ее ячейки или (что эквивалентно) когда ее коэффициент заполнения становится равным 1[1]. В некоторых программных средах при попытке вставить элемент в заполненную таблицу не остается ничего другого, как прибегнуть к аварийному завершению программы, сопровождаемому выдачей сообщения об ошибке. Однако мы предполагаем, что наша программная среда, подобно многим современным средам, обладает системой управления памятью, позволяющей по запросу выделять и освобождать блоки хранилища. Таким образом, когда в заполненную таблицу вставляется элемент, ее можно ***расширить*** (expand), выделив место для новой таблицы, содержащей больше ячеек, чем было в старой. Поскольку таблица всегда должна размещаться в непрерывной области памяти, для большей таблицы необходимо выделить новый массив, а затем — скопировать элементы из старой таблицы в новую.

Общепринятый эвристический подход заключается в том, чтобы в новой таблице было в два раза больше ячеек, чем в старой. Если в таблицу элементы только вставляются, то значение ее коэффициента заполнения будет не меньше $1/2$, поэтому объем неиспользованного места никогда не превысит половины полного размера таблицы.

В приведенном ниже псевдокоде предполагается, что T — объект, представляющий таблицу. Поле $table\,[T]$ содержит указатель на блок памяти, представляющий таблицу. Поле $num\,[T]$ содержит количество элементов в таблице, а поле $size\,[T]$ — полное количество ячеек в таблице. Изначально таблица пустая: $num\,[T] = size\,[T] = 0$.

```
TABLE_INSERT(T, x)
 1  if size[T] = 0
 2     then Выделить память для table[T] с 1 ячейкой
 3          size[T] ← 1
 4  if num[T] = size[T]
 5     then Выделить память для новой таблицы с 2 · size[T] ячейками
```

[1] В некоторых случаях, например, при работе с хешированными таблицами с открытой адресацией может понадобиться считать таблицу заполненной, если ее коэффициент заполнения равен некоторой константе, строго меньшей 1. (См. упражнение 17.4-1.)

```
 6          Вставить все элементы из table[T] в new_table
 7          Освободить table[T]
 8          table[T] ← new_table
 9          size[T] ← 2 · size[T]
10    Вставить x в table[T]
11    num[T] ← num[T] + 1
```

Обратите внимание, что здесь имеется две "вставки": сама процедура TABLE_INSERT и операция ***элементарной вставки*** (elementary insertion) в таблицу, выполняемая в строках 6 и 10. Время работы процедуры TABLE_INSERT можно проанализировать в терминах количества элементарных вставок, считая стоимость каждой такой операции равной 1. Предполагается, что фактическое время работы процедуры TABLE_INSERT линейно зависит от времени вставки отдельных элементов, так что накладные расходы на выделение исходной таблицы в строке 2 — константа, а накладные расходы на выделение и освобождение памяти в строках 5 и 7 пренебрежимо малы по сравнению со стоимостью переноса элементов в строке 6. Назовем ***расширением*** (expansion) событие, при котором выполняется оператор **then** (строки 5–9).

Проанализируем последовательность n операций TABLE_INSERT, которые выполняются над изначально пустой таблицей. Чему равна стоимость c_i i-й операции? Если в данный момент в таблице имеется свободное место (или если это первая операция), то $c_i = 1$, поскольку достаточно выполнить лишь одну элементарную операцию вставки в строке 10. Если же таблица заполнена и нужно произвести ее расширение, то $c_i = i$: стоимость равна 1 для элементарной вставки в строке 10 плюс $i - 1$ для элементов, которые необходимо скопировать из старой таблицы в новую (строка 6). Если выполняется n операций, то стоимость последней из них в наихудшем случае равна $O(n)$, в результате чего верхняя граница полного времени выполнения n операций равна $O\left(n^2\right)$.

Эта оценка является неточной, поскольку при выполнении n операций TABLE_INSERT потребность в расширении таблицы возникает не так часто. Точнее говоря, i-я операция приводит к расширению, только если величина $i - 1$ равна степени двойки. С помощью группового анализа можно показать, что амортизированная стоимость операции в действительности равна $O(1)$. Стоимость i-й операции равна

$$c = \begin{cases} i & \text{если } i-1 \text{ является степенью 2,} \\ 1 & \text{в противном случае.} \end{cases}$$

Таким образом, полная стоимость выполнения n операций TABLE_INSERT равна

$$\sum_{i=1}^{n} c_i \leqslant n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n,$$

поскольку стоимость не более n операций равна 1, а стоимости остальных операций образуют геометрическую прогрессию. Поскольку полная стоимость n операций TABLE_INSERT равна $3n$, амортизированная стоимость одной операции равна 3.

С помощью метода бухгалтерского учета можно показать "на пальцах", что амортизированная стоимость операции TABLE_INSERT должна быть равна 3. Интуитивно понятно, что для каждого элемента приходится трижды платить за элементарную операцию вставки: за саму вставку в текущую таблицу, за перемещение этого элемента при расширении таблицы и за перемещение еще одного элемента, который уже однажды был перемещен в ходе расширения таблицы. Например, предположим, что сразу после расширения таблицы ее размер становится равным m. Тогда количество элементов в ней равно $m/2$ и таблице соответствует нулевой кредит. На каждую вставку начисляется по 3 грн. Элементарная вставка, которая выполняется тут же, стоит 1 грн. Еще 1 грн. зачисляется в кредит на вставляемый элемент. Третья гривня зачисляется в качестве кредита на один из уже содержащихся в таблице $m/2$ элементов. Для заполнения таблицы требуется $m/2-1$ дополнительных вставок, поэтому к тому времени, когда в таблице будет m элементов и она станет заполненной, каждому элементу будет соответствовать гривня, предназначенная для уплаты за его перемещение в новую таблицу в процессе расширения.

Анализ последовательности из n операций TABLE_INSERT можно выполнить и с помощью метода потенциалов. Этим мы сейчас и займемся; кроме того, этот метод будет использован в разделе 17.4.2 для разработки операции TABLE_DELETE, амортизированная стоимость которой равна $O(1)$. Начнем с того, что определим потенциальную функцию Φ, которая становится равной 0 сразу после расширения и достигающей значения, равного размеру матрицы, к тому времени, когда матрица станет заполненной. В этом случае предстоящее расширение можно будет оплатить за счет потенциала. Одним из возможных вариантов является функция

$$\Phi(T) = 2 \cdot num[T] - size[T]. \tag{17.5}$$

Сразу после расширения выполняется соотношение $num[T] = size[T]/2$, поэтому, как и требуется, $\Phi(T) = 0$. Непосредственно перед расширением справедливо равенство $num[T] = size[T]$, следовательно, как и требуется, $\Phi(T) = num[T]$. Начальное значение потенциала равно 0, и поскольку таблица всегда заполнена не менее чем наполовину, выполняется неравенство $num[T] \geqslant size[T]/2$, из которого следует, что функция $\Phi(T)$ всегда неотрицательна. Таким образом, суммарная амортизированная стоимость n операций TABLE_INSERT является верхней границей суммарной фактической стоимости.

Чтобы проанализировать амортизированную стоимость i-й операции TABLE_INSERT, обозначим через num_i количество элементов, хранящихся в таблице после этой операции, через $size_i$ — общий размер таблицы после этой операции, и через Φ_i — потенциал после этой операции. Изначально $num_0 = 0$, $size_0 = 0$ и $\Phi_0 = 0$.

Если i-я операция TABLE_INSERT не приводит к расширению, то $size_i = size_{i-1}$ и амортизированная стоимость операции равна

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} = \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) = \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot (num_i - 1) - size_{i-1}) = \\ &= 3.\end{aligned}$$

Если же i-я операция TABLE_INSERT приводит к расширению, то $size_i = 2 \cdot size_{i-1}$ и $size_{i-1} = num_{i-1} = num_i - 1$, откуда следует, что $size_i = 2 \cdot (num_i - 1)$. В этом случае амортизированная стоимость операции будет следующей:

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} = \\ &= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) = \\ &= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2 \cdot (num_i - 1) - (num_i - 1)) = \\ &= num_i + 2 - (num_i - 1) = \\ &= 3.\end{aligned}$$

На рис. 17.3 приведена зависимость величин num_i, $size_i$ и Φ_i от i. Величина num_i обозначена тонкой линией, величина $size_i$ — пунктирной линией и Φ_i — толстой линией. Обратите внимание, что непосредственно перед расширением потенциал возрастает до значения, равного количеству содержащихся в таблице элементов, поэтому становится достаточным для уплаты за перемещение всех элементов в новую таблицу. Впоследствии потенциал снижается до нуля, но тут же возрастает до значения 2, когда в таблицу вставляется элемент, вызвавший ее расширение.

17.4.2 Расширение и сжатие таблицы

Чтобы реализовать операцию TABLE_DELETE, достаточно просто удалить из нее указанный элемент. Однако часто желательно ***сжать*** (contract) таблицу, если ее коэффициент заполнения становится слишком мал, чтобы пространство не расходовалось напрасно. Сжатие таблицы выполняется аналогично ее расширению: если количество элементов в ней достигает некоторой критической нижней отметки, выделяется место для новой, меньшей таблицы, после чего элементы из старой таблицы копируются в новую. Затем место, используемое для старой таблицы, можно будет освободить путем возвращения его системе управления памятью. В идеале желательно, чтобы выполнялись два свойства:

- коэффициент заполнения динамической таблицы ограничен снизу некоторой константой;

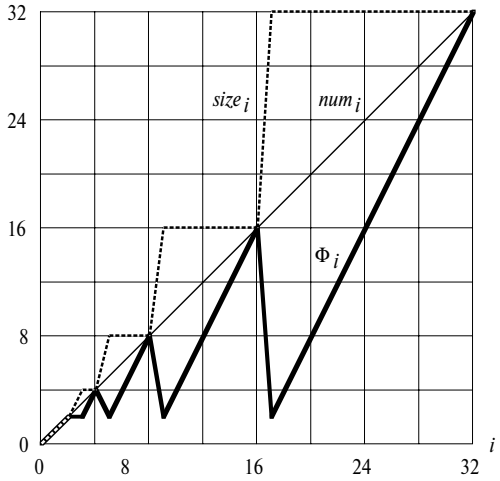


Рис. 17.3. Зависимость величин num_i (количество элементов в таблице), $size_i$ (количество ячеек в таблице) и потенциала Φ_i от количества операций TABLE_INSERT

- амортизированная стоимость операций по обработке таблицы ограничена сверху некоторой константой.

Предполагается, что эту стоимость можно измерить в терминах элементарных вставок и удалений.

Естественная стратегия при расширении и сжатии таблицы состоит в том, чтобы удваивать ее размер при вставке элемента в заполненную таблицу и в два раза уменьшать его, когда удаление элемента из таблицы приводит к тому, что она становится заполненной меньше чем на половину. При этом гарантируется, что величина коэффициента заполнения таблицы не может быть меньше $1/2$. Однако, к сожалению, это может привести к тому, что амортизированная стоимость операции станет довольно большой. Рассмотрим такой сценарий. Над таблицей T выполняется n операций, где n — степень двойки. Первые $n/2$ операций — операции вставки, и их полная стоимость, согласно нашему предыдущему анализу, равна $\Theta(n)$. В конце этой последовательности вставок $num[T] = size[T] = n/2$. В качестве следующих $n/2$ операций выполним такую последовательность:

$$\text{I, D, D, I, I, D, D, I, I, } \ldots,$$

где I обозначает вставку, а D — удаление. Первая вставка приводит к расширению таблицы, объем которой равен n. Два следующих удаления вызовут сжатие таблицы обратно до размера $n/2$. Следующие две вставки приведут к еще одному расширению и т.д. Стоимость каждого расширения и сжатия равна $\Theta(n)$,

и всего их — $\Theta(n)$. Таким образом, полная стоимость n операций равна $\Theta\left(n^2\right)$, а амортизированная стоимость одной операции — $\Theta(n)$.

Трудность, возникающая при использовании данной стратегии, очевидна: после расширения мы не успеваем выполнить достаточное количество удалений, чтобы оплатить сжатие. Аналогично, после сжатия не выполняется достаточное количество вставок, чтобы оплатить расширение.

Эту стратегию можно улучшить, если позволить, чтобы коэффициент заполнения таблицы опускался ниже $1/2$. В частности, как и раньше, размер таблицы будет удваиваться при вставке элемента в заполненную таблицу, но ее размер будет уменьшаться в два раза, если в результате удаления таблица становится заполненной не наполовину, как это было раньше, а на четверть. При этом коэффициент заполнения таблицы будет ограничен снизу константой $1/4$. Идея заключается в том, что после расширения таблицы ее коэффициент заполнения равен $1/2$, поэтому перед сжатием необходимо будет удалить половину элементов, поскольку сжатие не будет выполнено до тех пор, пока коэффициент заполнения не станет меньше $1/4$. Кроме того, после сжатия коэффициент заполнения таблицы также равен $1/2$. Таким образом, количество элементов таблицы должно быть удвоено путем выполнения вставок перед тем, как сможет произойти расширение, поскольку оно выполняется только в том случае, когда коэффициент заполнения может превысить 1.

Код процедуры TABLE_DELETE здесь не приводится, поскольку он аналогичен коду процедуры TABLE_INSERT. Однако для анализа удобно предположить, что если количество элементов таблицы падает до нуля, то выделенное для нее пространство освобождается, т.е. если $num[T] = 0$, то $size[T] = 0$.

Теперь можно проанализировать стоимость последовательности из n операций TABLE_INSERT и TABLE_DELETE, воспользовавшись методом потенциалов. Начнем с того, что определим потенциальную функцию Φ, которая равна 0 сразу после расширения или сжатия, и возрастает по мере того, как коэффициент заполнения увеличивается до 1 или уменьшается до $1/4$. Обозначим коэффициент заполнения непустой таблицы T через $\alpha(T) = num[T]/size[T]$. Поскольку для пустой таблицы выполняются соотношения $num[T] = size[T] = 0$ и $\alpha(T) = 1$, равенство $num[T]\,\alpha(T) \cdot size[T]$ всегда справедливо, независимо от того, является таблица пустой или нет. В качестве потенциальной будет использоваться следующая функция:

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \text{при } \alpha(T) \geqslant 1/2, \\ size[T]/2 - num[T] & \text{при } \alpha(T) < 1/2. \end{cases} \tag{17.6}$$

Обратите внимание, что потенциал пустой таблицы равен 0 и что потенциальная функция ни при каких аргументах не принимает отрицательного значения. Таким образом, полная амортизированная стоимость последовательности операций для

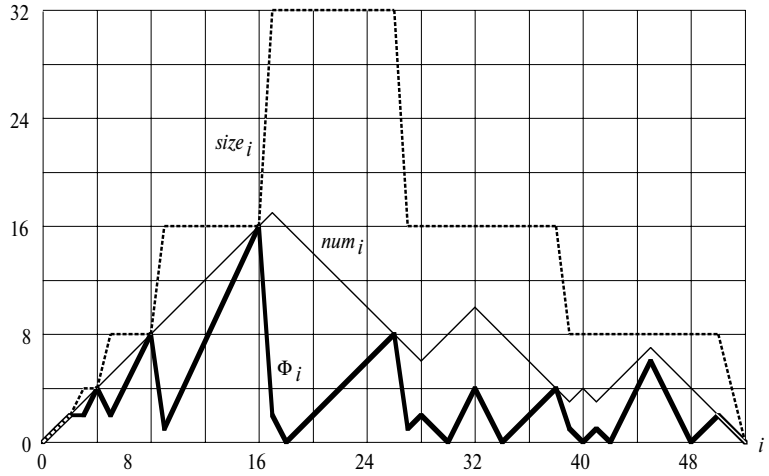


Рис. 17.4. Зависимость величин num_i (количество элементов в таблице), $size_i$ (количество ячеек в таблице) и потенциала Φ_i от количества операций TABLE_INSERT и TABLE_DELETE

функции Φ является верхней границей фактической стоимости этой последовательности.

Прежде чем продолжить анализ, сделаем паузу для обсуждения некоторых свойств потенциальной функции. Заметим, что когда коэффициент заполнения равен $1/2$, потенциал равен 0. Когда коэффициент заполнения равен 1, то $size\,[T] = num\,[T]$, откуда $\Phi\,(T) = num\,[T]$, следовательно, значение потенциала оказывается достаточным для оплаты расширения в случае добавления элемента. Если же коэффициент заполнения равен $1/4$, то выполняется соотношение $size\,[T] = 4 \cdot num\,[T]$, откуда $\Phi\,(T) = num\,[T]$, и в этом случае значение потенциала также является достаточным для оплаты сжатия в результате удаления элемента.

Чтобы проанализировать последовательность n операций TABLE_INSERT и TABLE_DELETE, обозначим через c_i фактическую стоимость i-й операции, через $\widehat{c_i}$ — ее амортизированную стоимость в соответствии с функцией Φ, через num_i — количество хранящихся в таблице элементов после выполнения i-й операции, через $size_i$ — полный объем таблицы после выполнения i-й операции, через α_i — коэффициент заполнения таблицы после выполнения i-й операции и через Φ_i — потенциал после выполнения i-й операции. Изначально $num_0 = 0$, $size_0 = 0$, $\alpha_0 = 1$ и $\Phi_0 = 0$.

На рис. 17.4 проиллюстрировано поведение потенциала при выполнении последовательности операций. Величина num_i обозначена тонкой линией, величина $size_i$ — пунктирной линией и величина Φ_i — толстой линией. Обратите внимание, что непосредственно перед расширением потенциал возрастает до значения, равного количеству содержащихся в таблице элементов, поэтому становится доста-

точным для уплаты за перемещение всех элементов в новую таблицу. Аналогично, непосредственно перед сжатием потенциал тоже достигает значения, равного количеству элементов таблицы.

Начнем со случая, когда i-й по счету выполняется операция TABLE_INSERT. Если $\alpha_{i-1} \geqslant 1/2$, то этот анализ идентичен тому, который был проведен в разделе 17.4.1 для расширения таблицы. Независимо от того, расширяется таблица или нет, амортизированная стоимость $\widehat{c_i}$ операции не превышает 3. Если $\alpha_{i-1} < 1/2$, таблицу нельзя расширить в результате выполнения операции, поскольку расширение происходит только когда $\alpha_{i-1} = 1$. Если, кроме того, $\alpha_i < 1/2$, то амортизированная стоимость i-й операции равна

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} = \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) = \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) = \\ &= 0.\end{aligned}$$

Если $\alpha_{i-1} < 1/2$, но $\alpha_i \geqslant 1/2$, то

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} = \\ &= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) = \\ &= 1 + (2 \cdot (num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) = \\ &= 3 \cdot num_{i-1} - \frac{3}{2}\, size_{i-1} + 3 = \\ &= 3\alpha_{i-1}\, size_{i-1} - \frac{3}{2}\, size_{i-1} + 3 < \\ &< \frac{3}{2}\, size_{i-1} - \frac{3}{2}\, size_{i-1} + 2 = 3.\end{aligned}$$

Таким образом, амортизированная стоимость операции TABLE_INSERT не превышает 3.

Теперь обратимся к случаю, когда i-й по счету выполняется операция TABLE_DELETE. В этом случае $num_i = num_{i-1} - 1$. Если $\alpha_{i-1} < 1/2$, то необходимо анализировать, приведет ли эта операция к сжатию матрицы. Если нет, то выполняется условие $size_i = size_{i-1}$, а амортизированная стоимость операции равна

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} = \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) = \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) = \\ &= 2.\end{aligned}$$

Если $\alpha_{i-1} < 1/2$ и i-я операция приводит к сжатию, то фактическая стоимость операции равна $c_i = num_i + 1$, поскольку один элемент удаляется и num_i элементов перемещается. В этом случае $size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$, и амортизированная стоимость операции равна

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} = \\ &= (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) = \\ &= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) = \\ &= 1.\end{aligned}$$

Если i-я операция — TABLE_DELETE и выполняется неравенство $\alpha_{i-1} \geqslant 1/2$, то амортизированная стоимость также ограничена сверху константой. Анализ этого случая предлагается выполнить в упражнении 17.4-2.

Подведем итог. Поскольку амортизированная стоимость каждой операции ограничена сверху константой, фактическая стоимость выполнения произвольной последовательности из n операций над динамической таблицей равна $O(n)$.

Упражнения

17.4-1. Предположим, что нужно реализовать динамическую хеш-таблицу с открытой адресацией. Почему может понадобиться считать такую таблицу заполненной, когда ее коэффициент заполнения достигнет некоторого значения α, строго меньшего 1? Кратко опишите, как следует выполнять вставки в динамическую хеш-таблицу с открытой адресацией, чтобы математическое ожидание амортизированной стоимости вставки было равно $O(1)$. Почему математическое ожидание фактической стоимости может быть равным $O(1)$ не для всех вставок?

17.4-2. Покажите, что если $\alpha_{i-1} \geqslant 1/2$ и i-я операция, выполняющаяся над динамической таблицей, — TABLE_DELETE, то амортизированная стоимость операции для потенциальной функции (17.6) ограничена сверху константой.

17.4-3. Предположим, что вместо того, чтобы сжимать таблицу, вдвое уменьшая ее размер при уменьшении коэффициента заполнения ниже значения $1/4$, мы будем сжимать ее до $2/3$ исходного размера, когда значение коэффициента заполнения становится меньше $1/3$. С помощью потенциальной функции

$$\Phi(T) = |2 \cdot num[T] - size[T]|$$

покажите, что амортизированная стоимость процедуры TABLE_DELETE, в которой применяется эта стратегия, ограничена сверху константой.

Задачи

17-1. **Обратный бинарный битовый счетчик**

В главе 30 описывается важный алгоритм, получивший название быстрого преобразования Фурье (Fast Fourier Transform, FFT). На первом этапе алгоритма FFT выполняется ***обратная перестановка битов*** (bit-reversal permutation) входного массива $A[0..n-1]$, длина которого равна $n = 2^k$ для некоторого неотрицательного целого k. В результате перестановки каждый элемент массива меняется местом с элементом, индекс которого представляет собой преобразованный описанным далее образом индекс исходного элемента.

Каждый индекс a можно представить в виде последовательности k битов $\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle$, где $a = \sum_{i=0}^{k-1} a_i 2^i$. Определим операцию

$$\mathrm{rev}_k\left(\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle\right) = \langle a_0, a_1, \ldots, a_{k-1} \rangle ,$$

т.е.

$$\mathrm{rev}_k\left(a\right) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i .$$

Например, если $n = 16$ (или, что эквивалентно, $k = 4$), то $\mathrm{rev}_k\left(3\right) = = 12$, поскольку 4-битовое двоичное представление числа 3 выглядит как 0011, и в результате преобразования которого получим 1100, т.е. 4-битовое двоичное представление числа 12.

а) Пусть имеется функция rev_k, выполняющаяся в течение времени $\Theta\left(n\right)$. Разработайте алгоритм обратной перестановки битов в массиве длиной $n = 2^k$, время работы которого равно $O\left(nk\right)$.

Чтобы улучшить время перестановки битов, можно воспользоваться алгоритмом, основанным на амортизационном анализе. Мы поддерживаем счетчик с обратным порядком битов и процедуру BIT_REVERSED_INCREMENT, которая для заданного показания счетчика a возвращает значение $\mathrm{rev}_k\left(\mathrm{rev}_k\left(a\right) + 1\right)$. Например, если $k = 4$ и счетчик начинает отсчет от 0, то в результате последовательных вызовов процедуры BIT_REVERSED_INCREMENT получим последовательность

$$0000, 1000, 0100, 1100, 0010, 1010, \cdots = 0, 8, 4, 12, 2, 10, \ldots .$$

б) Предположим, что слова в компьютере хранятся в виде k-битовых значений и что в течение единичного промежутка времени компьютер может произвести над этими бинарными значениями такие операции, как сдвиг влево или вправо на произвольное число позиций, побитовое И, побитовое ИЛИ и т.п. Опишите реализацию процедуры

BIT_REVERSED_INCREMENT, позволяющую выполнять обращение порядка битов n-элементного массива в течение времени $O(n)$.

в) Предположим, что в течение единичного промежутка времени сдвиг слова вправо или влево можно произвести только на один бит. Сохраняется ли при этом возможность реализовать обращение порядка битов в массиве за время $O(n)$?

17-2. **Динамический бинарный поиск**

Бинарный поиск в отсортированном массиве осуществляется в течение промежутка времени, описываемого логарифмической функцией, однако время вставки нового элемента линейно зависит от размера массива. Время вставки можно улучшить, используя несколько отсортированных массивов.

Предположим, что операции SEARCH и INSERT должны поддерживаться для n-элементного множества. Пусть $k = \lceil \lg(n+1) \rceil$ и пусть бинарное преставление числа n имеет вид $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. У нас имеется k таких отсортированных массивов $A_0, A_1, \ldots, A_{k-1}$, что для всех $i = 0, 1, \ldots, k-1$ длина массива A_i равна 2^i. Каждый массив либо заполнен, либо пуст, в зависимости от того, равно ли $n_i = 1$ или $n_i = 0$ соответственно. Таким образом, полное количество элементов, хранящихся во всех k массивах, равно $\sum_{i=0}^{k-1} n_i 2^i = n$. Хотя каждый массив отсортирован, между элементами различных массивов нет никаких отношений.

а) Опишите, как в этой структуре данных реализовать операцию SEARCH. Проанализируйте время ее работы в наихудшем случае.

б) Опишите, как в эту структуру данных вставлять новый элемент. Проанализируйте время выполнения этой операции в наихудшем случае и ее амортизированное время работы.

в) Обсудите реализацию процедуры DELETE.

17-3. **Амортизированные сбалансированные по весу деревья**

Рассмотрим обычное бинарное дерево поиска, расширенное за счет добавления к каждому узлу x поля $size[x]$, содержащего количество ключей, которые хранятся в поддереве с корнем в узле x. Пусть α — константа из диапазона $1/2 \leqslant \alpha < 1$. Назовем данный узел x ***α-сбалансированным*** (α-balanced), если

$$size[left[x]] \leqslant \alpha \cdot size[x]$$

и

$$size[right[x]] \leqslant \alpha \cdot size[x].$$

Дерево в целом является ***α-сбалансированным*** (α-balanced), если α-сбалансирован каждый его узел. Описанный ниже амортизированный подход к сбалансированным деревьям был предложен Варгисом (G. Varghese).

а) В определенном смысле $1/2$-сбалансированное дерево является наиболее сбалансированным. Пусть в произвольном бинарном дереве поиска задан узел x. Покажите, как перестроить поддерево с корнем в узле x таким образом, чтобы оно стало $1/2$-сбалансированным. Время работы алгоритма должно быть равным $\Theta(size[x])$, а объем используемой вспомогательной памяти — $O(size[x])$.

б) Покажите, что поиск в бинарном α-сбалансированном дереве поиска с n узлами в наихудшем случае выполняется в течение времени $O(\lg n)$.

При выполнении оставшейся части этой задачи предполагается, что константа α строго больше $1/2$. Предположим, что операции INSERT и DELETE реализованы так же, как и в обычном бинарном дереве поиска с n узлами, за исключением того, что после каждой такой операции, если хоть один узел дерева не является α-сбалансированным, то поддерево с самым высоким несбалансированным узлом "перестраивается" так, чтобы оно стало $1/2$-сбалансированным.

Проанализируем эту схему с помощью метода потенциалов. Для узла x, принадлежащего бинарному дереву поиска T, определим величину

$$\Delta(x) = |size[left[x]] - size[right[x]]|,$$

а потенциал дерева T определим как

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geqslant 2} \Delta x,$$

где c — достаточно большая константа, зависящая от α.

в) Покажите, что потенциал любого бинарного дерева поиска всегда неотрицателен и что потенциал $1/2$-сбалансированного дерева равен 0.

г) Предположим, что m единиц потенциала достаточно для оплаты перестройки поддерева, содержащего m узлов. Насколько большой должна быть константа c для данного α, чтобы амортизированное время перестройки поддерева, не являющегося α-сбалансированным, было равным $O(1)$?

д) Покажите, что амортизированное время вставки узла в α-сбалансированное дерево с n узлами, как и удаления узла из этого дерева, равно $O(\lg n)$.

17-4. **Стоимость перестройки красно-черных деревьев**

Имеется четыре основных операции над красно-черными деревьями, которые выполняют ***структурные модификации*** (structural modifications): вставка узла, удаление узла, поворот и изменение цвета. Мы убедимся, что для поддержки свойств красно-черных деревьев в процедурах RB_INSERT и RB_DELETE достаточно использовать $O(1)$ операций поворота, вставки узлов и удаления узлов, но в них потребуется намного больше операций изменения цвета.

а) Опишите корректное красно-черное дерево с n узлами, в котором вызов процедуры RB_INSERT для вставки $(n+1)$-го узла приводит к $\Omega(\lg n)$ изменениям цвета. Затем опишите корректное красно-черное дерево с n узлами, в котором вызов процедуры RB_DELETE для удаления определенного узла приводит к $\Omega(\lg n)$ изменениям цвета.

Несмотря на то, что в наихудшем случае количество модификаций цветов, приходящихся на одну операцию, может определяться логарифмической функцией, мы докажем, что произвольная последовательность m операций RB_INSERT и RB_DELETE, выполняемых над изначально пустым красно-черном деревом, в наихудшем случае приводит к $O(m)$ структурным модификациям.

б) Некоторые из случаев, обрабатываемых в основном цикле кода процедур RB_INSERT_FIXUP и RB_DELETE_FIXUP, являются ***завершающими*** (terminating): однажды случившись, они вызовут завершение цикла после выполнения фиксированного количества дополнительных операций. Установите для каждого случая, встречающегося в процедурах RB_INSERT_FIXUP и RB_DELETE_FIXUP, какой из них является завершающим, а какой — нет. (*Указание*: обратитесь к рис. 13.5–13.7.)

Сначала мы проанализируем структурные модификации для случая, когда выполняются только вставки. Пусть T — красно-черное дерево. Определим для него функцию $\Phi(T)$ как количество красных узлов в дереве T. Предположим, что одной единицы потенциала достаточно для оплаты структурной модификации, выполняющейся в любом из трех случаев процедуры RB_INSERT_FIXUP.

в) Пусть T' — результат применения случая 1 процедуры RB_INSERT_FIXUP к дереву T. Докажите, что $\Phi(T') = \Phi(T) - 1$.

г) Операцию вставки узла в красно-черное дерево с помощью процедуры RB_INSERT можно разбить на три части. Перечислите структурные модификации и изменения потенциала, которые происходят в результате выполнения строк 1–16 процедуры RB_INSERT для

незавершающих случаев процедуры RB_INSERT_FIXUP и для завершающих случаев этой процедуры.

д) Воспользовавшись результатами, полученными в части *г*), докажите, что амортизированное количество структурных модификаций, которые выполняются при любом вызове процедуры RB_INSERT, равно $O(1)$.

Теперь нам нужно доказать, что если выполняются и вставки, и удаления, то производится $O(m)$ структурных модификаций. Для каждого узла x определим величину

$$w(x) = \begin{cases} 0 & \text{если } x \text{ красный,} \\ 1 & \text{если } x \text{ черный и не имеет красных дочерних узлов,} \\ 0 & \text{если } x \text{ черный и имеет один красный дочерний узел,} \\ 2 & \text{если } x \text{ черный и имеет два красных дочерних узла.} \end{cases}$$

Теперь переопределим потенциал красно-черного дерева в виде

$$\Phi(T) = \sum_{x \in T} w(x),$$

и обозначим через T' дерево, получающееся в результате применения любого незавершающего случая процедуры RB_INSERT_FIXUP или RB_DELETE_FIXUP к дереву T.

е) Покажите, что для всех незавершающих случаев процедуры RB_INSERTFIXUP выполняется неравенство $\Phi(T') \leqslant \Phi(T) - 1$. Докажите, что амортизированное количество структурных модификаций, которые выполняются в произвольном вызове процедуры RB_INSERT_FIXUP, равно $O(1)$.

ж) Покажите, что для всех незавершающих случаев процедуры RB_DELETE_FIXUP выполняется неравенство $\Phi(T') \leqslant \Phi(T) - 1$. Докажите, что амортизированное количество структурных модификаций, которые выполняются в произвольном вызове процедуры RB_DELETE_FIXUP, равно $O(1)$.

з) Завершите доказательство того, что в наихудшем случае в ходе выполнения произвольной последовательности m операций RB_INSERT и RB_DELETE производится $O(m)$ структурных модификаций.

Заключительные замечания

Групповой анализ был использован Ахо (Aho), Хопкрофтом (Hopcroft) и Ульманом (Ullman) [5]. Таржан (Tarjan) [293] представил обзор разновидностей метода бухгалтерского учета и метода потенциала, использующихся в амортизационном анализе, а также некоторые их приложения. Метод бухгалтерского учета он приписывает нескольким авторам, включая Брауна (M.R. Brown), Таржана, Хаддельстона (S. Huddelston) и Мельхорна (K. Mehlhorn). Метод потенциала он приписывает Слитору (D.D. Sleator). Термин "амортизационный" вошел в обиход благодаря Слитору и Таржану.

Потенциальные функции оказываются полезными также при доказательстве нижних границ в задачах определенных типов. Для каждой конфигурации, возникающей в такой задаче, определяется потенциальная функция, отображающая эту конфигурацию на действительное число. После этого определяется потенциал Φ_{init} начальной конфигурации, потенциал Φ_{final} конечной конфигурации и максимальное изменение потенциала $\Delta\Phi_{\max}$ в результате выполнения произвольного шага. Таким образом, число шагов должно быть не меньше $\frac{|\Phi_{final}-\Phi_{init}|}{|\Delta\Phi_{\max}|}$. Примеры использования потенциальных функций для обоснования нижних границ при оценке сложности ввода-вывода представлены в работах Кормена (Cormen) [71], Флойда (Floyd) [91], а также Аггарвала (Aggarwal) и Виттера (Vitter) [4]. Крумм (Krumme), Цыбенко (Cybenko) и Венкатараман (Venkataraman) [194] воспользовались потенциальными функциями для оценки нижних границ ***скорости распространения слухов*** (gossiping): передачи единственного элемента из каждой вершины графа во все другие вершины.

ЧАСТЬ V

Сложные структуры данных

Введение

Эта часть возвращает нас к изучению структур данных, поддерживающих операции над динамическими множествами, но уже на более высоком уровне по сравнению с частью III. В двух главах этой части мы, например, будем использовать технологии амортизационного анализа, которые рассматриваются в главе 17.

В главе 18 мы рассмотрим B-деревья, которые представляют собой сбалансированные деревья поиска, разработанные специально для хранения информации на дисках. Поскольку дисковые операции существенно медленнее, чем работа с оперативной памятью, производительность B-деревьев определяется не только временем, необходимым для проведения вычислений, но и количеством необходимых дисковых операций. Для каждой операции с B-деревом количество обращений к диску растет с ростом высоты B-дерева, так что мы должны поддерживать ее невысокой.

В главах 19 и 20 рассматриваются различные реализации объединяемых пирамид, которые поддерживают операции INSERT, MINIMUM, EXTRACT_MIN и UNION[1] (операция UNION объединяет две пирамиды). Структуры данных в этих главах, кроме того, поддерживают операции DELETE и DECREASE_KEY.

Биномиальные пирамиды, о которых речь пойдет в главе 19, поддерживают все перечисленные операции со временем выполнения $O(\lg n)$ в худшем случае (n — общее количество элементов в пирамиде (или в двух пирамидах в случае операции UNION). При необходимости поддержки операции UNION биномиальные пирамиды превосходят бинарные пирамиды из главы 6, поскольку последним в наихудшем случае требуется для объединения двух пирамид время $\Theta(n)$.

Фибоначчиевы пирамиды (глава 20) представляют собой усовершенствованные биномиальные пирамиды — по крайней мере, теоретически. Для оценки производительности Фибоначчиевых пирамид используется амортизированное время выполнения операций. Операции INSERT, MINIMUM и UNION у Фибоначчиевых пирамид имеют амортизированное (а также фактическое) время работы, равное $O(1)$, а операции EXTRACT_MIN и DELETE — $O(\lg n)$. Главное преимущество Фибоначчиевых пирамид, однако, заключается в том, что операция DECREASE_KEY имеет амортизированное время $O(1)$. Именно это свойство делает Фибоначчиевы пирамиды ключевым компонентом ряда наиболее быстрых асимптотических алгоритмов для работы с графами.

И наконец, в главе 21 представлены структуры данных для хранения непересекающихся множеств. У нас имеется универсум из n элементов, сгруппированных

[1] Как и в задаче 10-2, мы определяем объединяемые пирамиды как поддерживающие операции MINIMUM и EXTRACT_MIN, так что мы можем называть их **объединяемыми неубывающими пирамидами** (mergeable min-heaps). Если в пирамиде поддерживаются операции MAXIMUM и EXTRACT_MAXIMUM, ее можно называть **объединяемой невозрастающей пирамидой** (mergeable max-heap).

в динамические множества. Изначально каждый элемент принадлежит своему собственному множеству. Операция UNION позволяет объединить два множества, а запрос FIND_SET определяет множество, в котором в данный момент находится искомый элемент. Представляя каждое множество в виде простого корневого дерева, мы получаем возможность удивительно производительной работы: последовательность из m операций выполняется за время $O(m\alpha(n))$, где $\alpha(n)$ — исключительно медленно растущая функция и не превышает 4 для всех мыслимых приложений. Амортизационный анализ, который позволяет доказать эту оценку, столь же сложен, сколь проста сама используемая структура.

Рассмотренные в этой части книги структуры данных — не более чем примеры "сложных" структур данных, и множество интересных структур сюда не вошли. Хотелось бы упомянуть следующие из них.

- ***Динамические деревья*** (dynamic trees), разработанные Слитором (Sleator) и Таржаном (Tarjan) [281, 292], поддерживают лес из непересекающихся деревьев. Каждое ребро каждого дерева имеет некоторую стоимость, выраженную действительным числом. Динамические деревья поддерживают запросы поиска родителя, корня, стоимости ребра и минимальной стоимости ребра на пути от узла к корню. У деревьев могут удаляться ребра, может изменяться стоимость на пути от узла к корню, корень может связываться с другим деревом, а также делать узел корнем дерева, в котором он находится. Имеется реализация динамических деревьев, которая обеспечивает амортизированное время работы каждой операции, равное $O(\lg n)$; другая, более сложная реализация обеспечивает время работы $O(\lg n)$ в наихудшем случае.
- ***Расширяющиеся деревья*** (splay trees), также разработанные Слитором (Sleator) и Таржаном (Tarjan) [282, 292], представляют собой версию бинарных деревьев поиска, в которых операции поиска имеют амортизированное время работы $O(\lg n)$. Одно из применений расширяющихся деревьев — упрощение реализации динамических деревьев.
- ***Перманентные структуры*** (persistent data structures), которые обеспечивают возможность выполнения запроса о предшествующих версиях структуры данных, а иногда и их обновления. Соответствующие методы с малыми затратами времени и памяти были предложены в работе [82] Дрисколлом (Driscoll), Сарнаком (Sarnak), Слитором (Sleator) и Таржаном (Tarjan). В задаче 13-1 приведен простой пример перманентного динамического множества.
- Ряд структур данных обеспечивает более быстрое выполнение словарных операций (INSERT, DELETE, SEARCH) для ограниченных универсумов ключей. Использование ограничений позволяет получить лучшее асимптотическое время работы, чем для структур данных, основанных на сравнении.

Структура данных, предложенная Ван Эмде Боасом (van Emde Boas) [301], обеспечивает в наихудшем случае время выполнения операций MINIMUM, MAXIMUM, INSERT, DELETE, SEARCH, EXTRACT_MIN, EXTRACT_MAX, PREDECESSOR и SUCCESSOR, равное $O(\lg\lg n)$, если ключи ограничены множеством $\{1, 2, \ldots, n\}$. Фредман (Fredman) и Виллард (Willard) [99] предложили использовать ***деревья слияний*** (fusion trees), которые были первыми структурами данных, обеспечивавшие повышение скорости словарных операций за счет ограничения пространства целыми числами. Они показали, как можно реализовать эти операции, чтобы их время работы составляло $O(\lg n/\lg\lg n)$. Ряд других структур данных, включая ***экспоненциальные деревья поиска*** (exponential search trees) [16], также обеспечивают улучшение всех (или некоторых) словарных операций и упоминаются в заключительных замечаниях в некоторых главах книги.

- ***Динамические графы*** (dynamic graph) поддерживают различные запросы, позволяя структуре графа изменяться в процессе операций добавления или удаления вершин или ребер. Примеры поддерживаемых запросов включают связи вершин [144], связи ребер, минимальные связующие деревья [143], двусвязность и транзитивное замыкание [142].

В заключительных замечаниях к различным главам данной книги упоминаются, помимо описанных, некоторые другие структуры данных.

ГЛАВА 18

B-деревья

B-деревья представляют собой сбалансированные деревья поиска, созданные специально для эффективной работы с дисковой памятью (и другими типами вторичной памяти с непосредственным доступом). B-деревья похожи на красно-черные деревья (см. главу 13), но отличаются более высокой оптимизацией дисковых операций ввода-вывода. Многие СУБД используют для хранения информации именно B-деревья (или их разновидности).

B-деревья отличаются от красно-черных деревьев тем, что узлы B-дерева могут иметь много дочерних узлов — до тысяч, так что степень ветвления B-дерева может быть очень большой (хотя обычно она определяется характеристиками используемых дисков). B-деревья схожи с красно-черными деревьями в том, что все B-деревья с n узлами имеют высоту $O(\lg n)$, хотя само значение высоты B-дерева существенно меньше, чем у красно-черного дерева за счет более сильного ветвления. Таким образом, B-деревья также могут использоваться для реализации многих операций над динамическими множествами за время $O(\lg n)$.

B-деревья представляют собой естественное обобщение бинарных деревьев поиска. На рис. 18.1 показан пример простого B-дерева. Если внутренний узел B-дерева содержит $n[x]$ ключей, то у него $n[x]+1$ дочерних узлов. Ключи в узле x используются как разделители диапазона ключей, с которыми имеет дело данный узел, на $n[x]+1$ поддиапазонов, каждый из которых относится к одному из дочерних узлов x. При поиске ключа в B-дереве мы выбираем один из $n[x]+$ $+1$ дочерних узлов путем сравнения искомого значения с $n[x]$ ключами узла x. Структура листьев B-дерева отличается от структуры внутренних узлов; мы рассмотрим эти отличия в разделе 18.1.

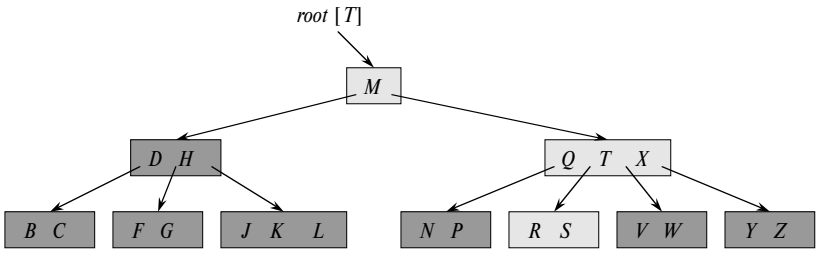


Рис. 18.1. B-дерево с согласными английского алфавита в качестве ключей

В разделе 18.1 приведено точное определение B-деревьев и доказан логарифмический рост высоты B-дерева в зависимости от количества его узлов. В разделе 18.2 описаны процессы поиска в B-дереве и вставки элемента в B-дерево, а в разделе 18.3 — процесс удаления. Однако перед тем как приступить к работе с B-деревьями, давайте выясним, почему структуры данных, созданные для работы с дисковой памятью, так отличаются от структур, предназначающихся для работы с оперативной памятью.

Структуры данных во вторичной памяти

Имеется несколько видов используемой компьютером памяти. ***Основная***, или ***оперативная***, ***память*** (primary, main memory) представляет собой специализированные микросхемы и обладает более высоким быстродействием и существенно более высокой ценой, чем магнитные носители, такие как магнитные диски или ленты. Большинство компьютеров, помимо первичной памяти, оснащены ***вторичной памятью*** (secondary storage) на базе магнитных дисков. Цена такой памяти на пару порядков ниже, чем первичной, а ее суммарный объем в типичной вычислительной системе на те же пару порядков превышает объем первичной памяти.

На рис. 18.2*а* показан типичный дисковый накопитель, состоящий из нескольких ***дисков*** (platters), вращающихся с постоянной скоростью на общем ***шпинделе*** (spindle). Поверхность каждого диска покрыта магнитным материалом. Каждый диск читается и записывается при помощи магнитной головки (***head***), расположенной на специальном рычаге. Все рычаги с головками собраны в единый пакет, который позволяет перемещать головки вдоль радиуса по направлению к шпинделю или от него к краю дисков. Когда головки находятся в зафиксированном состоянии, поверхность, проходящая под ними при вращении дисков, называется ***дорожкой*** (track). Поскольку головки оказываются выровнены по вертикали благодаря общей системе рычагов, так что обращение к набору дорожек (называющемуся ***цилиндром*** (cylinder) и показанному на рис. 18.2*б*) выполняется одновременно.

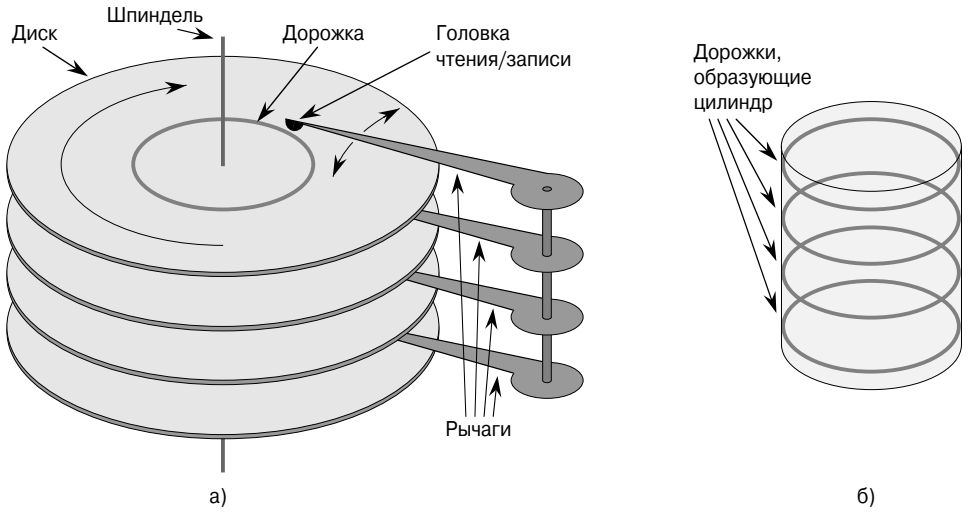


Рис. 18.2. Типичный дисковый накопитель

Хотя диски существенно дешевле оперативной памяти и имеют высокую емкость, они гораздо, гораздо медленнее оперативной памяти. Механическое движение головки относительно диска определяется двумя компонентами — перемещением головки по радиусу и вращением дисков. Когда писались эти строки, типичная скорость вращения дисков составляла 5 400–15 000 оборотов в минуту (rpm), с наиболее распространенной скоростью 7 200 rpm. Хотя такая скорость может показаться очень большой, один оборот требует примерно 8.33 мс, что почти на 5 порядков превышает время обращения к оперативной памяти (которое составляет примерно 100 нс). Другими словами, пока мы ждем оборота диска, чтобы считать необходимые нам данные, из оперативной памяти мы могли бы получить эти данные почти 100 000 раз! В среднем приходится ждать только половину оборота диска, но это практически ничего не меняет. Радиальное перемещение головок тоже требует времени. Одним словом, когда писались эти строки, наиболее распространенное время доступа к дисковой памяти составляло от 3 до 9 миллисекунд.

Для того чтобы снизить время ожидания, связанное с механическим перемещением, при обращении к диску выполняется обращение одновременно сразу к нескольким элементам, хранящимся на диске. Информация разделяется на несколько ***страниц*** (pages) одинакового размера, которые хранятся последовательно друг за другом в пределах одного цилиндра, и каждая операция чтения или записи работает сразу с несколькими страницами. Типичный размер страницы — от 2^{11} до 2^{14} байтов. После того как головка позиционирована на нужную

дорожку, а диск поворачивается так, что головка становится на начало интересующей нас страницы, операции чтения и записи выполняются очень быстро.

Зачастую обработка прочитанной информации занимает меньше времени, чем ее поиск и чтение с диска. По этой причине в данной главе мы отдельно рассматриваем два компонента времени работы алгоритма:

- количество обращений к диску;
- время вычислений (процессорное время).

Количество обращений к диску измеряется в терминах количества страниц информации, которое должно быть считано с диска или записано на него. Заметим, что время обращения к диску не является постоянной величиной, поскольку зависит от расстояния между текущей дорожкой и дорожкой с интересующей нас информацией, а также текущего угла поворота диска. Мы будем игнорировать это обстоятельство и в качестве первого приближения времени, необходимого для обращения к диску, будем использовать просто количество считываемых или записываемых страниц.

В типичном приложении, использующем B-деревья, количество обрабатываемых данных достаточно велико, и все они не могут одновременно разместиться в оперативной памяти. Алгоритмы работы с B-деревьями копируют в оперативную память с диска только некоторые выбранные страницы, необходимые для работы, и вновь записывают на диск те из них, которые были изменены в процессе работы. Алгоритмы работы с B-деревьями сконструированы таким образом, чтобы в любой момент времени обходиться только некоторым постоянным количеством страниц в основной памяти, так что ее объем не ограничивает размер B-деревьев, с которыми могут работать алгоритмы.

В нашем псевдокоде мы моделируем дисковые операции следующим образом. Пусть x — указатель на объект. Если объект находится в оперативной памяти компьютера, то мы обращаемся к его полям обычным способом — например, $key\,[x]$. Если же объект, к которому мы обращаемся посредством x, находится на диске, то мы должны выполнить операцию DISK_READ(x) для чтения объекта x в оперативную память перед тем, как будем обращаться к его полям. (Считаем, что если объект x уже находится в оперативной памяти, то операция DISK_READ(x) не требует обращения к диску.) Аналогично, для сохранения любых изменений в полях объекта x выполняется операция DISK_WRITE(x). Таким образом, типичный шаблон работы с объектом x имеет следующий вид:

$x \leftarrow$ указатель на некоторый объект
DISK_READ(x)
Операции, обращающиеся и/или изменяющие поля x
DISK_WRITE(x) ▷ Не требуется, если поля x не изменялись
Прочие операции, не изменяющие поля x

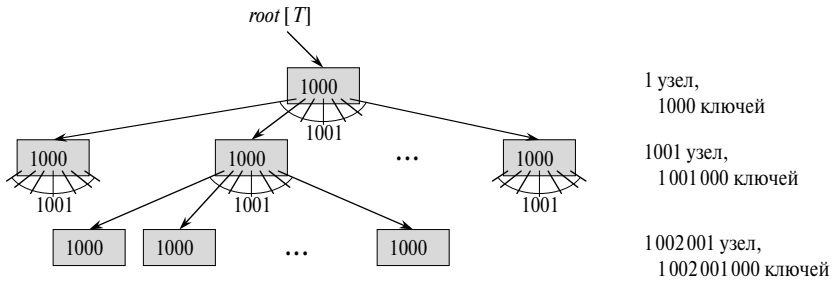


Рис. 18.3. B-дерево, хранящее более миллиарда ключей

Система в состоянии поддерживать в процессе работы в оперативной памяти только ограниченное количество страниц. Мы будем считать, что страницы, которые более не используются, удаляются из оперативной памяти системой; наши алгоритмы работы с B-деревьями не будут заниматься этим самостоятельно.

Поскольку в большинстве систем время выполнения алгоритма, работающего с B-деревьями, зависит в первую очередь от количества выполняемых операций с диском DISK_READ и DISK_WRITE, желательно минимизировать их количество и за один раз считывать и записывать как можно больше информации. Таким образом, размер узла B-дерева обычно соответствует дисковой странице. Количество потомков узла B-дерева, таким образом, ограничивается размером дисковой страницы.

Для больших B-деревьев, хранящихся на диске, степень ветвления обычно находится между 50 и 2000, в зависимости от размера ключа относительно размера страницы. Большая степень ветвления резко снижает как высоту дерева, так и количество обращений к диску для поиска ключа. На рис. 18.3 показано B-дерево высота которого равна 2, а степень ветвления — 1001; такое дерево может хранить более миллиарда ключей. При этом, поскольку корневой узел может храниться в оперативной памяти постоянно, для поиска ключа в этом дереве требуется максимум *два* обращения к диску!

18.1 Определение B-деревьев

Для простоты предположим, как и в случае бинарных деревьев поиска и красно-черных деревьев, что сопутствующая информация, связанная с ключом, хранится в узле вместе с ключом. На практике вместе с ключом может храниться только указатель на другую дисковую страницу, содержащую сопутствующую информацию для данного ключа. Псевдокод в данной главе неявно подразумевает, что при перемещении ключа от узла к узлу вместе с ним перемещается и сопутствующая информация или указатель на нее. В распространенном варианте

B-дерева, который называется B^+***-деревом***, вся сопутствующая информация хранится в листьях, а во внутренних узлах хранятся только ключи и указатели на дочерние узлы. Таким образом удается получить максимально возможную степень ветвления во внутренних узлах.

B-дерево T представляет собой корневое дерево (корень которого $root\,[T]$), обладающее следующими свойствами.

1. Каждый узел x содержит следующие поля:
   а) $n\,[x]$, количество ключей, хранящихся в настоящий момент в узле x.
   б) Собственно ключи, количество которых равно $n\,[x]$ и которые хранятся в невозрастающем порядке, так что $key_1\,[x] \leqslant key_2\,[x] \leqslant \cdots \leqslant \leqslant key_{n[x]}\,[x]$.
   в) Логическое значение $leaf\,[x]$, равное TRUE, если x является листом, и FALSE, если x — внутренний узел.
2. Кроме того, каждый внутренний узел x содержит $n\,[x]+1$ указателей $c_1\,[x]$, $c_2\,[x], \ldots, c_{n[x]+1}\,[x]$ на дочерние узлы. Листья не имеют дочерних узлов, так что их поля c_i не определены.
3. Ключи $key_i\,[x]$ разделяют поддиапазоны ключей, хранящихся в поддеревьях: если k_i — произвольный ключ, хранящийся в поддереве с корнем $c_i\,[x]$, то $k_1 \leqslant key_1\,[x] \leqslant k_2 \leqslant key_2\,[x] \leqslant \cdots \leqslant key_{n[x]}\,[x] \leqslant k_{n[x]+1}$.
4. Все листья расположены на одной и той же глубине, которая равна высоте дерева h.
5. Имеются нижняя и верхняя границы количества ключей, которые могут содержаться в узле. Эти границы могут быть выражены с помощью одного фиксированного целого числа $t \geqslant 2$, называемого ***минимальной степенью*** (minimum degree) B-дерева:
   а) Каждый узел, кроме корневого, должен содержать как минимум $t-1$ ключей. Каждый внутренний узел, не являющийся корневым, имеет, таким образом, как минимум t дочерних узлов. Если дерево не является пустым, корень должен содержать как минимум один ключ.
   б) Каждый узел содержит не более $2t-1$ ключей. Таким образом, внутренний узел имеет не более $2t$ дочерних узлов. Мы говорим, что узел ***заполнен*** (full), если он содержит ровно $2t-1$ ключей[1].

Простейшее B-дерево получается при $t = 2$. При этом каждый внутренний узел может иметь 2, 3 или 4 дочерних узла, и мы получаем так называемое ***2-3-4-дерево*** (2-3-4 tree).Однако обычно на практике используются гораздо бóльшие значения t.

[1] Другой распространенный вариант B-дерева под названием B^***-дерева**, требует, чтобы каждый внутренний узел был заполнен как минимум на две трети, а не наполовину, как в случае B-дерева.

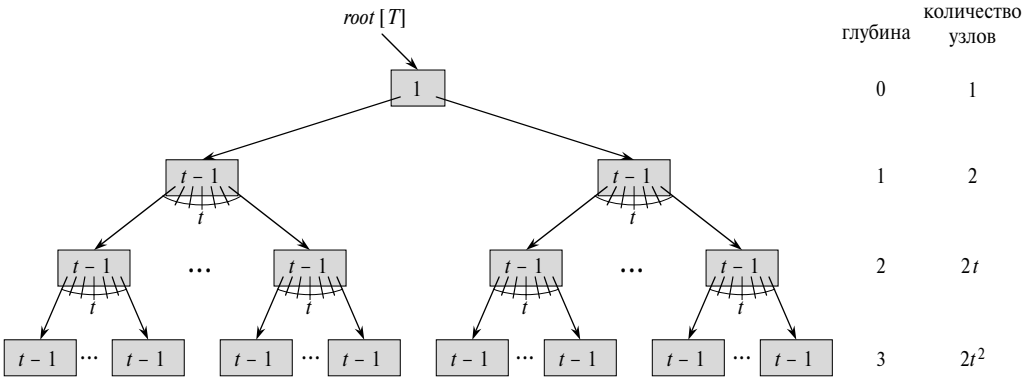


Рис. 18.4. B-дерево, высота которого равна 3, с минимально возможным количеством ключей

Высота B-дерева

Количество обращений к диску, необходимое для выполнения большинства операций с B-деревом, пропорционально его высоте. Проанализируем высоту B-дерева в наихудшем случае.

Теорема 18.1. Высота B-дерева T с $n \geqslant 1$ узлами и минимальной степенью $t \geqslant 2$ не превышает $\log_t (n+1)/2$.

Доказательство. Пусть B-дерево имеет высоту h. Корень дерева содержит как минимум один ключ, а все остальные узлы — как минимум по $t-1$ ключей. Таким образом, имеется как минимум 2 узла на глубине 1, как минимум $2t$ узлов на глубине 2, как минимум $2t^2$ узлов на глубине 3 и т.д., до глубины h, на которой имеется как минимум $2t^{h-1}$ узлов (пример такого дерева, высота которого равна 3, показан на рис. 18.4). Следовательно, число ключей n удовлетворяет следующему неравенству:

$$n \geqslant 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1.$$

Простейшее преобразование дает нам неравенство $t^h \leqslant (n+1)/2$. Теорема доказывается путем логарифмирования по основанию t обеих частей этого неравенства. ■

Здесь мы видим преимущества B-деревьев над красно-черными деревьями. Хотя высота деревьев растет как $O(\lg n)$ в обоих случаях (вспомним, что t — константа), в случае B-деревьев основание логарифмов имеет гораздо большее значение. Таким образом, B-деревья требуют исследования примерно в $\lg t$ раз

меньшего количества узлов по сравнению с красно-черными деревьями. Поскольку исследование узла дерева обычно требует обращения к диску, количество дисковых операций при работе с B-деревьями оказывается существенно сниженным.

Упражнения

18.1-1. Почему минимальная степень B-дерева не может быть равна 1?

18.1-2. Для каких значений t дерево на рис. 18.1 является корректным B-деревом?

18.1-3. Изобразите все корректные B-деревья с минимальной степенью 2, представляющие множество $\{1, 2, 3, 4, 5\}$.

18.1-4. Чему равно максимальное количество ключей, которое может храниться в B-дереве высотой h, выраженное в виде функции от минимальной степени дерева t?

18.1-5. Опишите структуру данных, которая получится, если в красно-черном дереве каждый черный узел поглотит красные дочерние узлы, а их потомки станут дочерними узлами этого черного узла.

18.2 Основные операции с B-деревьями

В этом разделе мы более подробно рассмотрим операции B_TREE_SEARCH, B_TREE_CREATE и B_TREE_INSERT. При рассмотрении этих операций мы примем следующие соглашения.

- Корень B-дерева всегда находится в оперативной памяти, так что операция DISK_READ для чтения корневого узла не нужна; однако при изменении корневого узла требуется выполнение операции DISK_WRITE, сохраняющей внесенные изменения на диске.
- Все узлы, передаваемые в качестве параметров, уже считаны с диска.

Все представленные здесь процедуры выполняются за один нисходящий проход по дереву.

Поиск в B-дереве

Поиск в B-дереве очень похож на поиск в бинарном дереве поиска, но с тем отличием, что если в бинарном дереве поиска мы выбирали один из двух путей, то здесь предстоит сделать выбор из большего количества альтернатив, зависящего от того, сколько дочерних узлов имеется у текущего узла. Точнее, в каждом внутреннем узле x нам предстоит выбрать один из $n[x] + 1$ дочерних узлов.

Операция B_TREE_SEARCH представляет собой естественное обобщение процедуры TREE_SEARCH, определенной для бинарных деревьев поиска. В качестве параметров процедура B_TREE_SEARCH получает указатель на корневой узел

x поддерева и ключ k, который следует найти в этом поддереве. Таким образом, вызов верхнего уровня для поиска во всем дереве имеет вид B_TREE_SEARCH($root\,[T]\,, k$). Если ключ k имеется в B-дереве, процедура B_TREE_SEARCH вернет упорядоченную пару (y, i), состоящую из узла y и индекса i, такого что $key_i\,[y] = k$. В противном случае процедура вернет значение NIL.

```
B_TREE_SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] и k > key_i[x]
3        do i ← i + 1
4  if i ≤ n[x] и k = key_i[x]
5     then return (x, i)
6  if leaf[x]
7     then return NIL
8     else  DISK_READ(c_i[x])
9           return B_TREE_SEARCH(c_i[x], k)
```

В строках 1–3 выполняется линейный поиск наименьшего индекса i, такого что $k \leqslant key_i\,[x]$ (иначе i присваивается значение $n\,[x]+1$). В строках 4–5 проверяется, не найден ли ключ в текущем узле, и если он найден, то выполняется его возврат. В строках 6–9 процедура либо завершает свою работу неудачей (если x является листом), либо рекурсивно вызывает себя для поиска в соответствующем поддереве x (после выполнения чтения с диска необходимого дочернего узла, являющегося корнем исследуемого поддерева).

На рис. 18.1 показана работа процедуры B_TREE_SEARCH: светлым цветом выделены узлы, просмотренные в процессе поиска ключа R.

Процедура B_TREE_SEARCH, как и процедура TREE_SEARCH при поиске в бинарном дереве, проходит в процессе рекурсии узлы от корня в нисходящем порядке. Количество дисковых страниц, к которым выполняется обращение процедурой B_TREE_SEARCH, равно $O\,(h) = O\,(\log_t n)$, где h — высота B-дерева, а n — количество содержащихся в нем узлов. Поскольку $n\,[x] < 2t$, количество итераций цикла **while** в строках 2–3 в каждом узле равно $O\,(t)$, а общее время вычислений — $O\,(th) = O\,(t \log_t n)$.

Создание пустого B-дерева

Для построения B-дерева T мы сначала должны воспользоваться процедурой B_TREE_CREATE для создания пустого корневого узла, а затем вносить в него новые ключи при помощи процедуры B_TREE_INSERT. В обеих этих процедурах используется вспомогательная процедура ALLOCATE_NODE, которая выделяет дисковую страницу для нового узла за время $O\,(1)$. Мы можем считать, что эта

процедура не требует вызова DISK_READ, поскольку никакой полезной информации о новом узле на диске нет.

B_TREE_CREATE(T)
1 $x \leftarrow$ ALLOCATE_NODE()
2 $leaf[x] \leftarrow$ TRUE
3 $n[x] \leftarrow 0$
4 DISK_WRITE(x)
5 $root[T] \leftarrow x$

Процедура B_TREE_CREATE требует $O(1)$ дисковых операций и выполняется за время $O(1)$.

Вставка ключа в B-дерево

Вставка ключа в B-дерево существенно сложнее вставки в бинарное дерево поиска. Как и в случае бинарных деревьев поиска, мы ищем позицию листа, в который будет вставлен новый ключ. Однако при работе с B-деревом мы не можем просто создать новый лист и вставить в него ключ, поскольку такое дерево не будет являться корректным B-деревом. Вместо этого мы вставляем новый ключ в существующий лист. Поскольку вставить новый ключ в заполненный лист невозможно, мы вводим новую операцию — ***разбиение*** (splitting) заполненного (т.е. содержащего $2t - 1$ ключей) узла на два, каждый из которых содержит по $t - 1$ ключей. ***Медиана***, или средний ключ, — $key_t[y]$ (median key) — при этом перемещается в родительский узел, где становится разделительной точкой для двух вновь образовавшихся поддеревьев. Однако если родительский узел тоже заполнен, перед вставкой нового ключа его также следует разбить, и такой процесс разбиения может идти по восходящей до самого корня.

Как и в случае бинарного дерева поиска, в B-дереве мы вполне можем осуществить вставку за один нисходящий проход от корня к листу. Для этого нам не надо выяснять, требуется ли разбить узел, в который должен вставляться новый ключ. Вместо этого при проходе от корня к листьям в поисках позиции для нового ключа мы разбиваем все заполненные узлы, через которые проходим (включая лист). Тем самым гарантируется, что если нам надо разбить какой-то узел, то его родительский узел не будет заполнен.

Разбиение узла B-дерева

Процедура B_TREE_SPLIT_CHILD получает в качестве входного параметра *незаполненный* внутренний узел x (находящийся в оперативной памяти), индекс i и узел y (также находящийся в оперативной памяти), такой что $y = c_i[x]$ является заполненным дочерним узлом x. Процедура разбивает дочерний узел на два и соответствующим образом обновляет поля x, внося в него информацию о новом

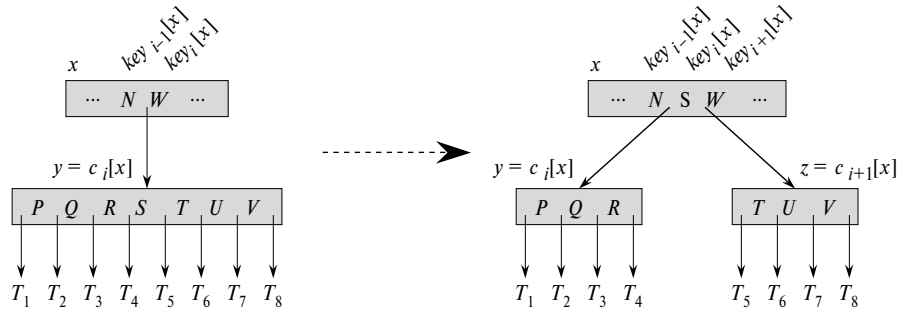


Рис. 18.5. Разбиение узла с $t = 4$

дочернем узле. Для разбиения заполненного корневого узла мы сначала делаем корень дочерним узлом нового пустого корневого узла, после чего можем использовать вызов B_TREE_SPLIT_CHILD. При этом высота дерева увеличивается на 1. Разбиение — единственное средство увеличения высоты B-дерева.

На рис. 18.5 показан этот процесс. Заполненный узел y разбивается медианой S, которая перемещается в родительский узел. Те ключи из y, которые больше медианы, помещаются в новый узел z, который становится новым дочерним узлом x.

```
B_TREE_SPLIT_CHILD(x, i, y)
 1  z ← ALLOCATE_NODE()
 2  leaf[z] ← leaf[y]
 3  n[z] ← t − 1
 4  for j ← 1 to t − 1
 5        do key_j[z] ← key_{j+t}[y]
 6  if not leaf[y]
 7     then for j ← 1 to t
 8               do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t − 1
10  for j ← n[x] + 1 downto i + 1
11        do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14        do key_{j+1}[x] ← key_j[x]
15  key_i[x] ← key_t[y]
16  n[x] ← n[x] + 1
17  DISK_WRITE(y)
18  DISK_WRITE(z)
19  DISK_WRITE(x)
```

Процедура B_TREE_SPLIT_CHILD использует простой способ "вырезать и вставить". Здесь y является i-м дочерним узлом x и представляет собой именно тот узел, который будет разбит на два. Изначально узел y имеет $2t$ дочерних узла (содержит $2t-1$ ключей); после разбиения количество его дочерних узлов снизится до t ($t-1$ ключей). Узел z получает t бóльших дочерних узлов ($t-1$ ключей) y и становится новым дочерним узлом x, располагаясь непосредственно после y в таблице дочерних узлов x. Медиана y перемещается в узел x и разделяет в нем y и z.

В строках 1–8 создается узел z и в него переносятся бóльшие $t-1$ ключей и соответствующие t дочерних узлов y. В строке 9 обновляется поле количества ключей в y. И наконец, строки 10–16 делают z дочерним узлом x, перенося медиану из y в x для разделения y и z, и обновляют поле количества ключей в x. В строках 17–19 выполняется запись на диск всех модифицированных данных. Время работы процедуры равно $\Theta(t)$ из-за циклов в строках 4–5 и 7–9 (прочие циклы выполняют $O(t)$ итераций). Процедура выполняет $O(1)$ дисковых операций.

Вставка ключа в B-дерево за один проход

Вставка ключа k в B-дерево T высоты h выполняется за один нисходящий проход по дереву, требующий $O(h)$ обращений к диску. Необходимое процессорное время составляет $O(th) = O(t \log_t n)$. Процедура B_TREE_INSERT использует процедуру B_TREE_SPLIT_CHILD для гарантии того, что рекурсия никогда не столкнется с заполненным узлом.

```
B_TREE_INSERT(T, k)
 1  r ← root[T]
 2  if n[r] = 2t − 1
 3     then s ← ALLOCATE_NODE()
 4          root[T] ← s
 5          leaf[s] ← FALSE
 6          n[s] ← 0
 7          c1[s] ← r
 8          B_TREE_SPLIT_CHILD(s, 1, r)
 9          B_TREE_INSERT_NONFULL(s, k)
10     else B_TREE_INSERT_NONFULL(r, k)
```

Строки 3–9 предназначены для случая, когда заполнен корень дерева: при этом корень разбивается и новый узел s (у которого два дочерних узла) становится новым корнем B-дерева. Разбиение корня — единственный путь увеличения высоты B-дерева. На рис. 18.6 проиллюстрирован такой процесс разбиения корневого узла. В отличие от бинарных деревьев поиска, у B-деревьев высота увеличивается "сверху", а не "снизу". Завершается процедура вызовом другой про-

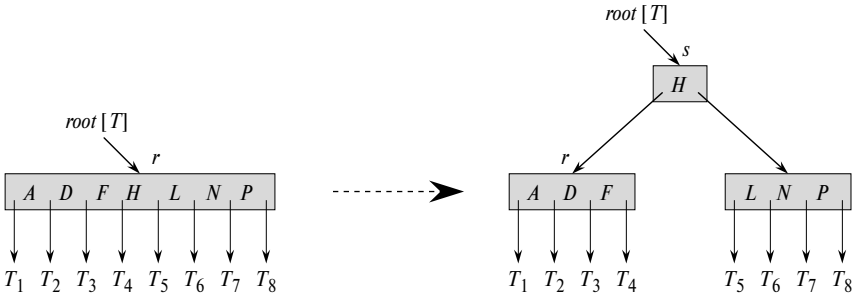


Рис. 18.6. Разбиение корня с $t = 4$

цедуры — B_TREE_INSERT_NONFULL, которая выполняет вставку ключа k в дерево с незаполненным корнем. Данная процедура при необходимости рекурсивно спускается вниз по дереву, причем каждый узел, в который она входит, является незаполненным, что обеспечивается (при необходимости) вызовом процедуры B_TREE_SPLIT_CHILD.

Вспомогательная процедура B_TREE_INSERT_NONFULL вставляет ключ k в узел x, который должен быть незаполненным при вызове процедуры. Операции B_TREE_INSERT и B_TREE_INSERT_NONFULL гарантируют, что это условие незаполненности будет выполнено.

```
B_TREE_INSERT_NONFULL(x, k)
 1  i ← n[x]
 2  if leaf[x]
 3     then while i ≥ 1 и k < key_i[x]
 4               do key_{i+1}[x] ← key_i[x]
 5                  i ← i − 1
 6          key_{i+1}[x] ← k
 7          n[x] ← n[x] + 1
 8          DISK_WRITE(x)
 9     else while i ≥ 1 и k < key_i[x]
10               do i ← i − 1
11          i ← i + 1
12          DISK_READ(c_i[x])
13          if n[c_i[x]] = 2t − 1
14            then B_TREE_SPLIT_CHILD(x, i, c_i[x])
15                 if k > key_i[x]
16                   then i ← i + 1
17          B_TREE_INSERT_NONFULL(c_i[x], k)
```

Процедура B_TREE_INSERT_NONFULL работает следующим образом. Строки 3–8 обрабатывают случай, когда x является листом; при этом ключ k просто вставляется в данный лист. Если же x не является листом, то мы должны вставить k в подходящий лист в поддереве, корнем которого является внутренний узел x. В этом случае строки 9–11 определяют дочерний узел x, в который спустится рекурсия. В строке 13 проверяется, не заполнен ли этот дочерний узел, и если он заполнен, то вызывается процедура B_TREE_SPLIT_CHILD, которая разбивает его на два незаполненных узла, а строки 15–16 определяют, в какой из двух получившихся в результате разбиения узлов должна спуститься рекурсия. (Обратите внимание, что в строке 16 после увеличения i операция чтения с диска не нужна, поскольку процедура рекурсивно спускается в узел, только что созданный процедурой B_TREE_SPLIT_CHILD.) Строки 13–16 гарантируют, что процедура никогда не столкнется с заполненным узлом. Строка 17 рекурсивно вызывает процедуру B_TREE_INSERT_NONFULL для вставки k в соответствующее поддерево. На рис. 18.7 проиллюстрированы различные ситуации, возникающие при вставке в B-дерево.

Количество обращений к диску, выполняемых процедурой B_TREE_INSERT для B-дерева высотой h, составляет $O(h)$, поскольку между вызовами B_TREE_INSERT_NONFULL выполняется только $O(1)$ операций DISK_READ и DISK_WRITE. Необходимое процессорное время равно $O(th) = O(t \log_t n)$. Поскольку в B_TREE_INSERT_NONFULL использована оконечная рекурсия, ее можно реализовать итеративно с помощью цикла **while**, наглядно показывающего, что количество страниц, которые должны находиться в оперативной памяти, в любой момент времени равно $O(1)$.

Упражнения

18.2-1. Покажите результат вставки ключей F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E в указанном порядке в изначально пустое B-дерево с минимальной степенью 2. Нарисуйте только конфигурации дерева непосредственно перед выполнением разбиения и окончательный вид дерева.

18.2-2. Поясните, при каких условиях могут выполняться излишние операции DISK_READ и DISK_WRITE в процессе работы процедуры B_TREE_INSERT (если таковые могут иметь место). Под излишней операцией чтения подразумевается чтение страницы, уже находящейся в оперативной памяти, а под излишней записью — запись на диск информации, идентичной уже имеющейся там.

18.2-3. Как найти минимальный ключ в B-дереве? Как найти элемент B-дерева, предшествующий данному?

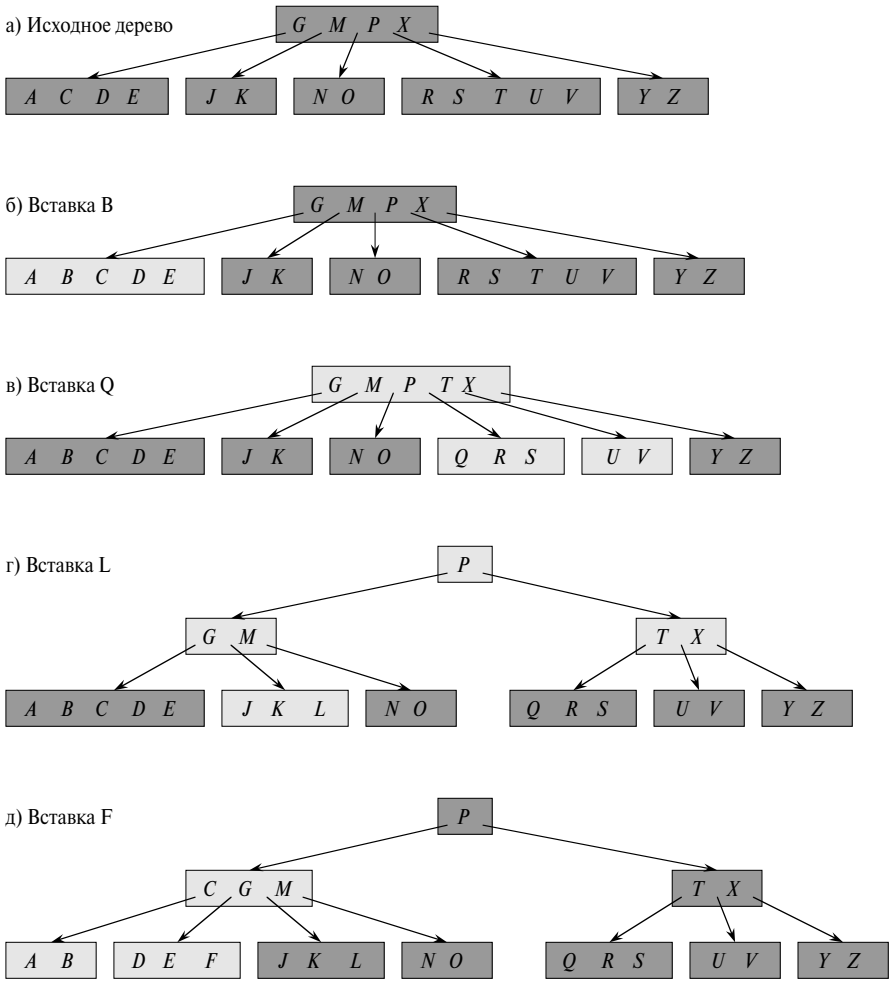


Рис. 18.7. Вставка ключей в B-дерево с минимальной степенью 3. Узлы, изменяемые в процессе вставки, выделены светлым цветом

⋆ 18.2-4. Предположим, что ключи $\{1, 2, \ldots, n\}$ вставлены в пустое B-дерево с минимальной степенью 2. Сколько узлов будет иметь получившееся в результате B-дерево?

18.2-5. Поскольку листья не содержат указателей на дочерние узлы, в них можно разместить большее количество ключей, чем во внутренних узлах при том же размере дисковой страницы. Покажите, как надо изменить процедуры создания B-дерева и вставки в него для работы с таким видоизмененным B-деревом.

18.2-6. Предположим, что линейный поиск в узле в процедуре B_TREE_SEARCH заменен бинарным поиском. Покажите, что при этом процессорное время, необходимое для выполнения процедуры B_TREE_SEARCH, равно $O(\lg n)$ (и не зависит от t).

18.2-7. Предположим, что дисковое аппаратное обеспечение позволяет произвольно выбирать размер дисковой страницы, но время, необходимое для чтения дисковой страницы, равно $a + bt$, где a и b — определенные константы, а t — минимальная степень B-дерева, использующего страницу данного размера. Каким образом выбрать t, чтобы (приближенно) минимизировать время поиска в B-дереве? Оцените оптимальное значение t для $a = 5$ мс и $b = 10$ мкс.

18.3 Удаление ключа из B-дерева

Удаление ключа из B-дерева, хотя и аналогично вставке, представляет собой более сложную задачу. Это связано с тем, что ключ может быть удален из любого узла, а не только из листа, а удаление из внутреннего узла требует определенной перестройки дочерних узлов. Как и в случае вставки, мы должны обеспечить, чтобы при выполнении операции удаления не были нарушены свойства B-дерева. Аналогично тому, как мы имели возможность убедиться, что узлы не слишком сильно заполнены для вставки нового ключа, нам предстоит убедиться, что узел не становится слишком мало заполнен в процессе удаления ключа (за исключением корневого узла, который может иметь менее $t - 1$ ключей, хотя и не может иметь более $2t - 1$ ключей).

Итак, пусть процедура B_TREE_DELETE должна удалить ключ k из поддерева, корнем которого является узел x. Эта процедура разработана таким образом, что при ее рекурсивном вызове для узла x гарантировано наличие в этом узле по крайней мере t ключей. Это условие требует наличия в узле большего количества ключей, чем минимальное в обычном B-дереве, так что иногда ключ может быть перемещен в дочерний узел перед тем, как рекурсия обратится к этому дочернему узлу. Такое ужесточение свойства B-дерева (наличие “запасного” ключа) дает нам возможность выполнить удаление ключа за один нисходящий проход по дереву (с единственным исключением, которое будет пояснено позже). Следует также учесть, что если корень дерева x становится внутренним узлом, не содержащим ни одного ключа (такая ситуация может возникнуть в рассматриваемых ниже случаях 2*в* и 3*б*), то узел x удаляется, а его единственный дочерний узел $c_1[x]$ становится новым корнем дерева (при этом уменьшается высота B-дерева и сохраняется его свойство, требующее, чтобы корневой узел непустого дерева содержал как минимум один ключ).

Вместо того чтобы представить вам полный псевдокод процедуры удаления узла из B-дерева, мы просто набросаем последовательность выполняемых действий. На рис. 18.8 показаны различные возникающие при этом ситуации.

1. Если узел k находится в узле x и x является листом — удаляем k из x.
2. Если узел k находится в узле x и x является внутренним узлом, выполняем следующие действия.

   а) Если дочерний по отношению к x узел y, предшествующий ключу k в узле x, содержит не менее t ключей, то находим k' — предшественника k в поддереве, корнем которого является y. Рекурсивно удаляем k' и заменяем k в x ключом k'. (Поиск ключа k' и удаление его можно выполнить в процессе одного нисходящего прохода.)

   б) Ситуация, симметричная ситуации *а*: если дочерний по отношению к x узел z, следующий за ключом k в узле x, содержит не менее t ключей, то находим k' — следующий за k ключ в поддереве, корнем которого является z. Рекурсивно удаляем k' и заменяем k в x ключом k'. (Поиск ключа k' и удаление его можно выполнить в процессе одного нисходящего прохода.)

   в) В противном случае, если и y, и z содержат по $t-1$ ключей, вносим k и все ключи z в y (при этом из x удаляется k и указатель на z, а узел y после этого содержит $2t-1$ ключей), а затем освобождаем z и рекурсивно удаляем k из y.

3. Если ключ k отсутствует во внутреннем узле x, находим корень $c_i[x]$ поддерева, которое должно содержать k (если таковой ключ имеется в данном B-дереве). Если $c_i[x]$ содержит только $t-1$ ключей, выполняем шаг 3*а* или 3*б* для того, чтобы гарантировать, что далее мы переходим в узел, содержащий как минимум t ключей. Затем мы рекурсивно удаляем k из поддерева с корнем $c_i[x]$.

   а) Если $c_i[x]$ содержит только $t-1$ ключей, но при этом один из ее непосредственных соседей (под которым мы понимаем дочерний по отношению к x узел, отделенный от рассматриваемого ровно одним ключом-разделителем) содержит как минимум t ключей, передадим в $c_i[x]$ ключ-разделитель между данным узлом и его непосредственным соседом из x, на его место поместим крайний ключ из соседнего узла и перенесем соответствующий указатель из соседнего узла в $c_i[x]$.

   б) Если и $c_i[x]$ и оба его непосредственных соседа содержат по $t-1$ ключей, объединим $c_i[x]$ с одним из его соседей (при этом бывший ключ-разделитель из x станет медианой нового узла).

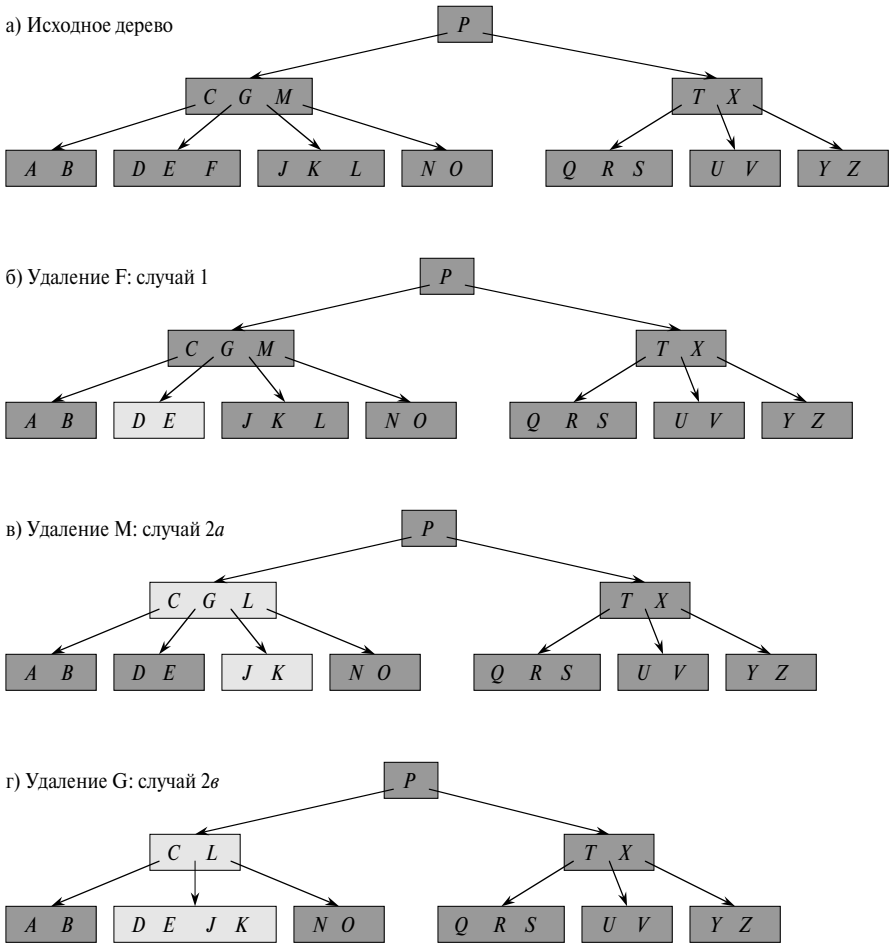
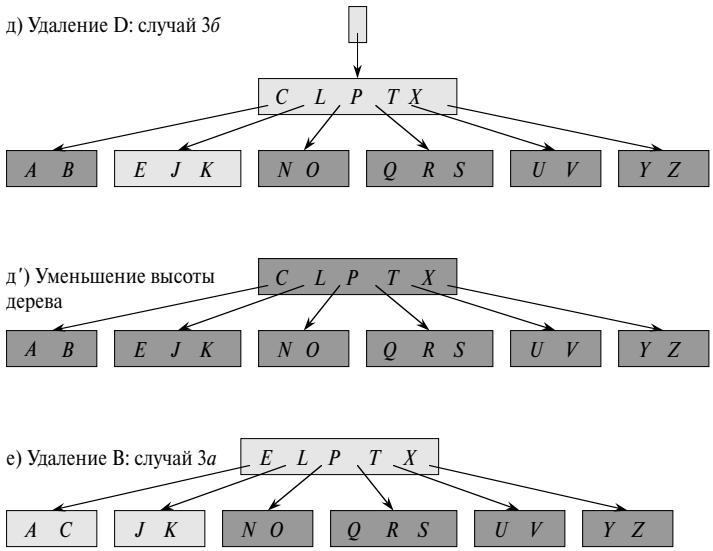


Рис. 18.8. Удаление узла из B-дерева с минимальной степенью 3. Модифицируемые узлы показаны светлым цветом

Поскольку большинство ключей в B-дереве находится в листьях, можно ожидать, что на практике чаще всего удаления будут выполняться из листьев. Процедура B_TREE_DELETE в этом случае выполняется за один нисходящий проход по дереву, без возвратов. Однако при удалении ключа из внутреннего узла процедуре может потребоваться возврат к узлу, ключ из которого был удален и замещен его предшественником или последующим за ним ключом (случаи 2*а* и 2*б*).

Хотя описание процедуры выглядит достаточно запутанным, она требует всего лишь $O(h)$ дисковых операций для дерева высотой h, поскольку между рекурсивными вызовами процедуры выполняется только $O(1)$ вызовов процедур



DISK_READ или DISK_WRITE. Необходимое процессорное время составляет $O(th) = O(t \log_t n)$.

Упражнения

18.3-1. Изобразите результат удаления ключей C, P и V (в указанном порядке) из B-дерева, приведенного на рис. 18.8*е*.

18.3-2. Напишите псевдокод процедуры B_TREE_DELETE.

Задачи

18-1. **Стеки во вторичной памяти**

Рассмотрим реализацию стека в компьютере с небольшой оперативной памятью, но относительно большим дисковым пространством. Поддерживаемые стеком операции PUSH и POP должны работать со значениями, представляющими отдельные машинные слова. Стек должен иметь возможность роста до размеров, существенно превышающих размер оперативной памяти, так что его бóльшая часть должна храниться на диске.

Простая, но неэффективная реализация стека хранит его весь на диске. В памяти поддерживается только указатель стека, который представляет собой дисковый адрес элемента на вершине стека. Если указатель имеет значение p, элемент на вершине стека является $(p \bmod m)$-м словом на странице $\lfloor p/m \rfloor$, где m — количество слов в одной странице.

Для реализации операции PUSH мы увеличиваем указатель стека, считываем в оперативную память с диска соответствующую страницу, копируем вносимый в стек элемент в соответствующее место на странице и записываем страницу обратно на диск. Реализация операции POP аналогична реализации операции PUSH. Мы уменьшаем указатель стека, считываем в оперативную память с диска соответствующую страницу и возвращаем элемент на вершине стека. Нет необходимости записывать страницу назад на диск, поскольку она не была модифицирована.

Поскольку дисковые операции достаточно дорогие в смысле времени их выполнения, мы учитываем при любой реализации стека две стоимости: общее количество обращений к диску и необходимое процессорное время. Будем считать, что дисковые операции со страницей размером m слов требуют одного обращения к диску и процессорное время $\Theta(m)$.

а) Чему в наихудшем случае равно асимптотическое количество обращений к диску при такой простейшей реализации стека? Чему равно требуемое процессорное время для n стековых операций? (Ваш ответ на этот и последующие вопросы данной задачи должен выражаться через m и n.)

Теперь рассмотрим реализацию стека, при которой одна страница стека находится в оперативной памяти (кроме того, небольшое количество оперативной памяти используется для отслеживания того, какая именно страница находится в оперативной памяти в настоящее время). Само собой разумеется, что мы можем выполнять стековые операции только в том случае, если соответствующая дисковая страница находится в оперативной памяти. При необходимости страница, находящаяся в данный момент в оперативной памяти, может быть записана на диск, а в память считана новая дисковая страница. Если нужная нам страница уже находится в оперативной памяти, то выполнение дисковых операций не требуется.

б) Чему в наихудшем случае равно количество необходимых обращений к диску для n операций PUSH? Чему равно соответствующее процессорное время?

в) Чему в наихудшем случае равно количество необходимых обращений к диску для n стековых операций? Чему равно соответствующее процессорное время?

Предположим, что стек реализован таким образом, что в оперативной памяти постоянно находятся две страницы (в дополнение к небольшому количеству оперативной памяти, которое используется для отслеживания

того, какие именно страницы находятся в оперативной памяти в настоящее время).

г) Как следует управлять страницами стека, чтобы амортизированное количество обращений к диску для любой стековой операции составляло $O(1/m)$, а амортизированное процессорное время — $O(1)$?

18-2. **Объединение и разделение 2-3-4-деревьев**

Операция ***объединения*** (join) получает два динамических массива S' и S'' и элемент x, такой что $key[x'] < key[x] < key[x'']$ для любых $x' \in S'$ и $x'' \in S''$, и возвращает множество $S = S' \cup \{x\} \cup S''$. Операция ***разделения*** (split) представляет собой обращение операции объединения: для данного динамического множества S и элемента $x \in S$ она создает множество S', содержащее все элементы из $S - \{x\}$, ключи которых меньше $key[x]$, и множество S'', содержащее все элементы из $S - \{x\}$, ключи которых больше $key[x]$. В данной задаче мы рассматриваем реализацию этих операций над 2-3-4-деревьями. Для удобства полагаем, что элементы представляют собой ключи и что все ключи различны.

а) Покажите, каким образом поддерживать хранение в каждом узле дерева информации о высоте поддерева, корнем которого является данный узел, чтобы это не повлияло на асимптотическое время выполнения операций поиска, вставки и удаления.

б) Покажите, как реализовать операцию объединения. Для данных 2-3-4-деревьев T' и T'' и ключа k объединение должно выполняться за время $O(1 + |h' - h''|)$, где h' и h'' — значения высоты деревьев T' и T'' соответственно.

в) Рассмотрим путь p от корня 2-3-4-дерева T к данному ключу k, множество S' ключей T, которые меньше k, и множество S'' ключей T, которые больше k. Покажите, что p разбивает S' на множество деревьев $\{T'_0, T'_1, \ldots, T'_m\}$ и множество ключей $\{k'_0, k'_1, \ldots, k'_m\}$, причем для всех $y \in T'_{i-1}$ и $z \in T'_i$ выполняется $y < k'_i < z$ $(i = 1, 2, \ldots, m)$. Как связаны высоты T'_{i-1} и T'_i? На какие множества деревьев и ключей p разбивает S''?

г) Покажите, как реализовать операцию разделения 2-3-4-дерева T. Воспользуйтесь операцией объединения для того, чтобы собрать ключи из S' в единое 2-3-4-дерево T', а ключи из S'' в единое 2-3-4-дерево T''. Время работы операции разделения должно составлять $O(\lg n)$, где n — количество ключей в T. (*Указание*: при сложении стоимости операций объединения происходит сокращение членов (телескопической) суммы.)

Заключительные замечания

Сбалансированные деревья и B-деревья достаточно подробно рассмотрены в книгах Кнута (Knuth) [185], Ахо (Aho), Хопкрофта (Hopcroft), Ульмана (Ullman) [5] и Седжвика (Sedgewick) [269]. Подробный обзор B-деревьев дан Комером (Comer) [66]. Гибас (Guibas) и Седжвик [135] рассмотрели взаимосвязи между различными видами сбалансированных деревьев, включая красно-черные деревья и 2-3-4-деревья.

В 1970 году Хопкрофт предложил ввести понятие 2-3-деревьев, которые были предшественниками B-деревьев и 2-3-4-деревьев и в которых каждый внутренний узел имел 2 или 3 дочерних узла. B-деревья предложены Баером (Bayer) и Мак-Крейтом (McCreight) в 1972 году [32] (выбор названия для своей разработки они не пояснили).

Бендер (Bender), Демейн (Demaine) и Фарах-Колтон (Farach-Colton) [37] изучили производительность B-деревьев при наличии иерархической памяти. Алгоритмы с игнорированием кэширования эффективно работают без явного знания о размерах обмена данными между различными видами памяти.

ГЛАВА 19

Биномиальные пирамиды

В данной главе и главе 20 представлены структуры данных, известные под названием ***сливаемых пирамид*** (mergeable heaps), которые поддерживают следующие пять операций.

MAKE_HEAP() создает и возвращает новую пустую пирамиду.

INSERT(H, x) вставляет узел x (с заполненным полем key) в пирамиду H.

MINIMUM(H) возвращает указатель на узел в пирамиде H, обладающий минимальным ключом.

EXTRACT_MIN(H) удаляет из пирамиды H узел, ключ которого минимален, возвращая при этом указатель на этот узел.

UNION(H_1, H_2) создает (и возвращает) новую пирамиду, которая содержит все узлы пирамид H_1 и H_2. Исходные пирамиды при выполнении этой операции уничтожаются.

Кроме того, структуры данных, описанные в этих главах, поддерживают следующие две операции.

DECREASE_KEY(H, x, k) присваивает узлу x в пирамиде H новое значение ключа k (предполагается, что новое значение ключа не превосходит текущего)[1].

DELETE(H, x) удаляет узел x из пирамиды H.

[1]Как упоминалось во введении к пятой части книги, по умолчанию сливаемые пирамиды являются неубывающими сливаемыми пирамидами, так что к ним применимы операции MINIMUM, EXTRACT_MIN и DECREASE_KEY. Мы можем также определить невозрастающие сливаемые пирамиды с операциями MAXIMUM, EXTRACT_MAX и INCREASE_KEY.

Как видно из табл. 19.1, если нам не нужна операция UNION, вполне хорошо работают обычные бинарные пирамиды, использовавшиеся в пирамидальной сортировке (глава 6). Все остальные операции выполняются в бинарной пирамиде в худшем случае за время $O(\lg n)$. Однако при необходимости поддержки операции UNION привлекательность бинарных пирамид резко уменьшается, поскольку реализация операции UNION путем объединения двух массивов, в которых хранятся бинарные пирамиды, с последующим выполнением процедуры MIN_HEAPIFY, как показано в упражнении 6.2-2, требует в худшем случае времени $\Theta(n)$.

Таблица 19.1. Время выполнения операций у разных реализаций сливаемых пирамид

Процедура	Бинарная пирамида (наихудший случай)	Биномиальная пирамида (наихудший случай)	Пирамида Фибоначчи (амортизированное время)
MAKE_HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT_MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Omega(\lg n)$	$\Theta(1)$
DECREASE_KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

В данной главе мы познакомимся с биномиальными пирамидами, время выполнения операций в наихудшем случае для которых также показаны в табл. 19.1. В частности, операция UNION требует только $O(\lg n)$ времени для объединения двух биномиальных пирамид с общим количеством n элементов.

В главе 20 мы познакомимся с пирамидами Фибоначчи, у которых для выполнения ряда операций нужно еще меньше времени (однако в табл. 19.1 показано амортизированное время выполнения операций, а не время выполнения операции в наихудшем случае).

В этой главе мы не рассматриваем вопросы выделения памяти для узлов перед вставкой и освобождения памяти после их удаления, полагая, что за это отвечает код, вызывающий процедуры для работы с пирамидами.

Все три указанных вида пирамид не могут обеспечить эффективную реализацию поиска элемента с заданным ключом (операция SEARCH), поэтому такие операции, как DECREASE_KEY и DELETE в качестве параметра получают указатель на узел, а не значение его ключа. Как говорилось при рассмотрении очередей с приоритетами в разделе 6.5, когда мы используем сливаемые пирамиды в приложении, мы часто храним в каждом элементе пирамиды дескриптор соответствующего объекта приложения, так же как и каждый объект приложения

хранит дескриптор соответствующего элемента пирамиды. Точная природа этих дескрипторов зависит от приложения и его реализации.

В разделе 19.1 определяются биномиальные деревья и пирамиды, а также частично рассмотрен вопрос их реализации. В разделе 19.2 показано, каким образом можно реализовать операции над биномиальными пирамидами, время работы которых будет соответствовать приведенному в табл. 19.1.

19.1 Биномиальные деревья и биномиальные пирамиды

Биномиальная пирамида представляет собой коллекцию биномиальных деревьев, так что данный раздел начинается с определения биномиальных деревьев и доказательства некоторых ключевых свойств. Затем мы определим биномиальные пирамиды и познакомимся с их возможным представлением.

19.1.1 Биномиальные деревья

Биномиальное дерево (binomial tree) B_k представляет собой рекурсивно определенное упорядоченное дерево (см. раздел Б.5.2). Как показано на рис. 19.1*а*, биномиальное дерево B_0 состоит из одного узла. Биномиальное дерево B_k состоит из двух биномиальных деревьев B_{k-1}, ***связанных*** вместе: корень одного из них является крайним левым дочерним узлом корня второго дерева. На рис. 19.1*б* показаны биномиальные деревья от B_0 до B_4.

Некоторые свойства биномиальных деревьев приводятся в следующей лемме.

Лемма 19.1 (Свойства биномиальных деревьев). Биномиальное дерево B_k

1. имеет 2^k узлов;
2. имеет высоту k;
3. имеет ровно $\binom{k}{i}$ узлов на глубине $i = 0, 1, \ldots, k$;
4. имеет корень степени k; степень всех остальных вершин меньше степени корня биномиального дерева. Кроме того, если дочерние узлы корня пронумеровать слева направо числами $k-1, k-2, \ldots, 0$, то i-й дочерний узел корня является корнем биномиального дерева B_i.

Доказательство. Доказательство выполняется по индукции по k. Базой индукции является биномиальное дерево B_0, для которого тривиальна проверка выполнения свойств, перечисленных в лемме.

Пусть перечисленные свойства выполняются для биномиального дерева B_{k-1}.

1. Биномиальное дерево B_k состоит из двух копий B_{k-1}, поэтому B_k содержит $2^{k-1} + 2^{k-1} = 2^k$ узлов.

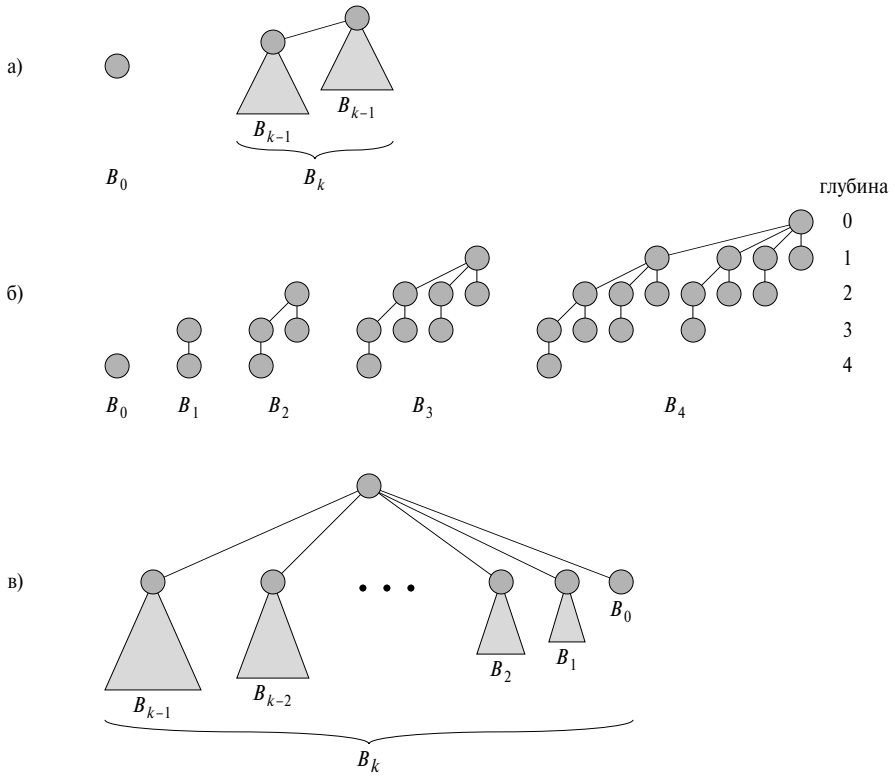


Рис. 19.1. Рекурсивное определение биномиального дерева, примеры биномиальных деревьев и альтернативное представление биномиальных деревьев

2. Из способа, которым выполняется связывание двух копий B_{k-1} в одно биномиальное дерево B_k, следует, что глубина биномиального дерева B_k на единицу больше максимальной глубины биномиального дерева B_{k-1}. Таким образом, в соответствии с гипотезой индукции максимальная глубина дерева B_k равна $(k-1)+1=k$.
3. Пусть $D(k,i)$ — количество узлов на глубине i у биномиального дерева B_k. Поскольку B_k состоит из двух связанных копий B_{k-1}, узел биномиального дерева B_{k-1} на глубине i появляется один раз на глубине i в биномиальном дереве B_k, и второй — на глубине $i+1$. Другими словами, количество узлов на глубине i в биномиальном дереве B_k равно количеству узлов на глубине i в биномиальном дереве B_{k-1} плюс количество узлов на глубине $i-1$ в B_{k-1}. Таким образом,

$$D(k,i) = D(k-1,i) + D(k-1,i-1) = \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}.$$

(При выводе использован результат упражнения B.1-7.)

4. Единственным узлом в биномиальном дереве B_k со степенью, превышающей степень в B_{k-1}, является корень, который имеет на один дочерний узел больше, чем в биномиальном дереве B_{k-1}. Поскольку корень биномиального дерева B_{k-1} имеет степень $k-1$, корень B_k имеет степень k. Теперь, в соответствии с гипотезой индукции, как показано на рис. 19.1*в*, дочерними узлами корня B_{k-1} являются корни биномиальных деревьев $B_{k-2}, B_{k-3}, \ldots, B_0$. Таким образом, когда выполняется привязка еще одного биномиального дерева B_{k-1}, дочерними узлами корня создаваемого дерева становятся кони биномиальных деревьев $B_{k-1}, B_{k-2}, \ldots, B_0$. ■

Следствие 19.2. Максимальная степень произвольного узла в биномиальном дереве с n узлами равна $\lg n$.

Доказательство. Доказательство следует из свойств 1 и 4 леммы 19.1. ■

Термин "биномиальное дерево" происходит из свойства 3 леммы 19.1, поскольку члены $\binom{k}{i}$ представляют собой биномиальные коэффициенты. В упражнении 19.1-3 приводится еще одно обоснование этого термина.

19.1.2 Биномиальные пирамиды

Биномиальная пирамида (binomial heap) H представляет собой множество биномиальных деревьев, которые удовлетворяют следующим ***свойствам биномиальных пирамид***.

1. Каждое биномиальное дерево в H подчиняется ***свойству неубывающей пирамиды*** (min-heap property): ключ узла не меньше ключа его родительского узла. Мы говорим, что такие деревья являются ***упорядоченными в соответствии со свойством неубывающей пирамиды*** (min-heap-ordered).
2. Для любого неотрицательного целого k имеется не более одного биномиального дерева H, чей корень имеет степень k.

Первое свойство говорит нам о том, что корень дерева, упорядоченного в соответствии со свойством неубывающей пирамиды, содержит наименьший ключ в дереве.

Из второго свойства следует, что биномиальная пирамида H, содержащая n узлов, состоит не более чем из $\lfloor \lg n \rfloor + 1$ биномиальных деревьев. Чтобы понять, почему это так, заметим, что бинарное представление числа n имеет $\lfloor \lg n \rfloor + 1$ битов, скажем, $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \ldots, b_0 \rangle$, так что $n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$. В соответствии со свойством 1 леммы 19.1, биномиальное дерево B_i входит в состав H тогда

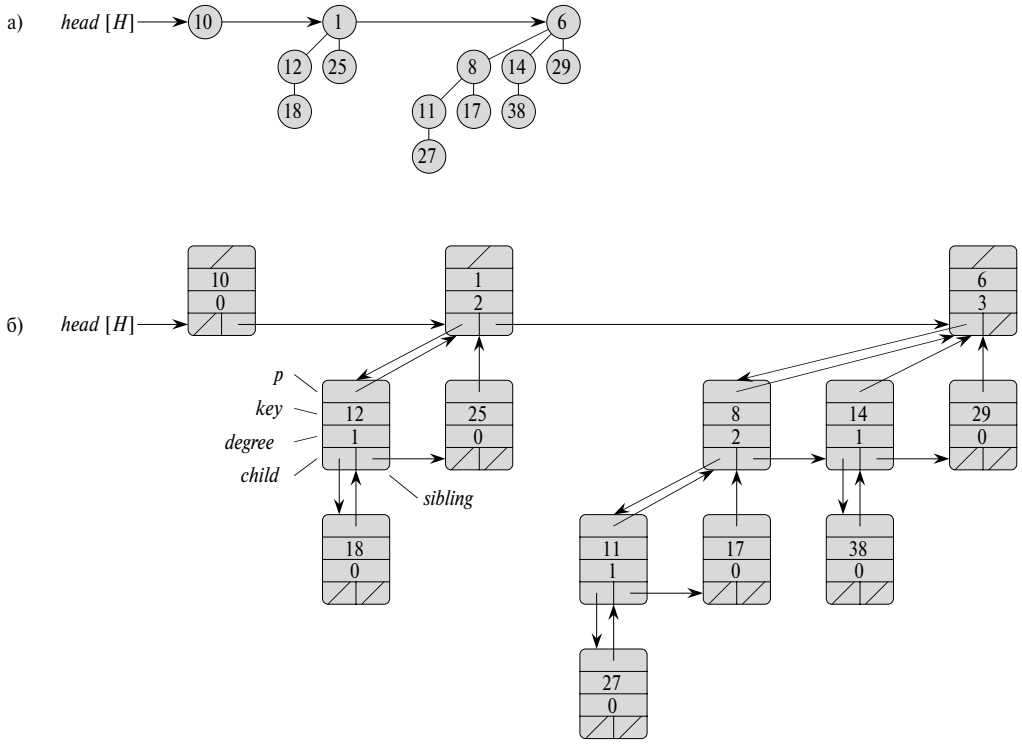


Рис. 19.2. Биномиальная пирамида с 13 узлами

и только тогда, когда бит $b_i = 1$. Следовательно, биномиальная пирамида H содержит не более $\lfloor \lg n \rfloor + 1$ биномиальных деревьев.

На рис. 19.2*а* показана биномиальная пирамида с 13 узлами. Бинарное представление числа 13 имеет вид $\langle 1101 \rangle$, и биномиальная пирамида H состоит из биномиальных деревьев B_0, B_2 и B_3, которые содержат, соответственно, 1, 4 и 8 узлов, причем ключ каждого из узлов не превышает ключ родительского узла.

Представление биномиальных пирамид

Как показано на рис. 19.2*б*, каждое биномиальное дерево в биномиальной пирамиде хранится в представлении с левым дочерним и правым сестринским узлами (которое рассматривалось в разделе 10.4). Каждый узел имеет поле key и содержит сопутствующую информацию, необходимую для работы приложения. Кроме того, каждый узел содержит указатель $p[x]$ на родительский узел, указатель $child[x]$ на крайний левый дочерний узел и указатель $sibling[x]$ на правый сестринский по отношению к x узел. Если x — корневой узел, то $p[x] = \text{NIL}$. Если узел x не имеет дочерних узлов, то $child[x] = \text{NIL}$, а если x — самый правый

дочерний узел своего родителя, то $sibling\,[x] = \text{NIL}$. Каждый узел x, кроме того, содержит поле $degree\,[x]$, в котором хранится количество дочерних узлов x.

Как видно из рис. 19.2, корни биномиальных деревьев внутри биномиальной пирамиды организованы в виде связанного списка, который мы будем называть ***списком корней*** (root list). При проходе по списку степени корней находятся в строго возрастающем порядке. В соответствии со вторым свойством биномиальных пирамид, в биномиальной пирамиде с n узлами степени корней представляют собой подмножество множества $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$. Поле $sibling$ имеет различный смысл для корней и для прочих узлов. Если x — корень, то $sibling\,[x]$ указывает на следующий корень в списке (как обычно, если x — последний корень в списке, то $sibling\,[x] = \text{NIL}$).

Обратиться к данной биномиальной пирамиде H можно при помощи поля $head\,[H]$, которое представляет собой указатель на первый корень в списке корней H. Если биномиальная пирамида не содержит элементов, то $head\,[H] = \text{NIL}$.

Упражнения

19.1-1. Предположим, что x — узел биномиального дерева в биномиальной пирамиде и что $sibling\,[x] \neq \text{NIL}$. Если x не является корнем, как соотносятся $degree\,[sibling[x]]$ и $degree\,[x]$? А если x — корень биномиального дерева?

19.1-2. Если x — некорневой узел биномиального дерева в биномиальной пирамиде, как соотносятся $degree\,[x]$ и $degree\,[p[x]]$?

19.1-3. Предположим, что мы помечаем узлы биномиального дерева B_k последовательными двоичными числами при обходе в обратном порядке, как показано на рис. 19.3. Рассмотрим узел x на глубине i, помеченный числом l, и пусть $j = k - i$. Покажите, что в двоичном представлении x содержится j единиц. Сколько всего имеется k-строк, содержащих в точности j единиц? Покажите, что степень x равна количеству единиц справа от крайнего правого нуля в двоичном представлении l.

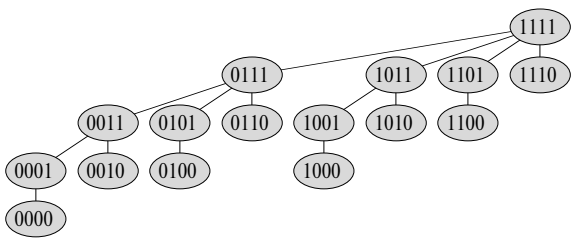


Рис. 19.3. Биномиальное дерево B_4 с узлами, помеченными двоичными числами при обходе в обратном порядке

19.2 Операции над биномиальными пирамидами

В этом разделе мы покажем, как выполнить операции над биномиальными пирамидами за время, приведенное в табл. 19.1. Мы покажем только верхние границы; поиск нижних границ остается в качестве упражнения 19.2-10.

Создание новой биномиальной пирамиды

Для создания пустой биномиальной пирамиды процедура MAKE_BINOMIAL_HEAP просто выделяет память и возвращает объект H, где $head\,[H] = \text{NIL}$. Время работы этой процедуры составляет $\Theta\,(1)$.

Поиск минимального ключа

Процедура BINOMIAL_HEAP_MINIMUM возвращает указатель на узел с минимальным ключом в биномиальной пирамиде H с n узлами. В данной реализации предполагается, что в биномиальной пирамиде отсутствуют ключи, равные ∞ (см. упражнение 19.2-5).

```
BINOMIAL_HEAP_MINIMUM(H)
1  y ← NIL
2  x ← head[H]
3  min ← ∞
4  while x ≠ NIL
5        do if key[x] < min
6              then min ← key[x]
7                   y ← x
8           x ← sibling[x]
9  return y
```

Поскольку биномиальная пирамида является невозрастающей, минимальный ключ должен находиться в корне. Процедура BINOMIAL_HEAP_MINIMUM проверяет все корни, количество которых не превышает $\lfloor \lg n \rfloor + 1$, сохраняя текущий минимум в переменной min, а указатель на текущий минимум — в переменной y. При вызове для биномиальной пирамиды, представленной на рис. 19.2, процедура BINOMIAL_HEAP_MINIMUM вернет указатель на узел с ключом 1.

Поскольку проверить надо не более $\lfloor \lg n \rfloor + 1$ корней, время работы процедуры BINOMIAL_HEAP_MINIMUM составляет $O\,(\lg n)$.

Слияние двух биномиальных пирамид

Операция по слиянию двух биномиальных пирамид используется в качестве подпрограммы в большинстве остальных операций. Процедура BINOMIAL_HEAP_UNION многократно связывает биномиальные деревья с корнями одинаковой степени. Приведенная далее процедура связывает дерево B_{k-1} с корнем y с деревом B_{k-1} с корнем z, делая z родительским узлом для y, после чего узел z становится корнем дерева B_k:

```
BINOMIAL_LINK(y, z)
1  p[y] ← z
2  sibling[y] ← child[z]
3  child[z] ← y
4  degree[z] ← degree[z] + 1
```

Процедура BINOMIAL_LINK делает узел y новым заголовком связанного списка дочерних узлов узла z за время $O(1)$. Работа процедуры основана на том, что представление с левым дочерним и правым сестринским узлами каждого биномиального дерева отвечают свойству упорядоченности дерева: в биномиальном дереве B_k крайний левый дочерний узел корня является корнем биномиального дерева B_{k-1}.

Приведенная далее процедура сливает биномиальные пирамиды H_1 и H_2, возвращая получаемую в результате биномиальную пирамиду, причем в процессе слияния представления биномиальных пирамид H_1 и H_2 уничтожаются. Помимо процедуры BINOMIAL_LINK, данная процедура использует вспомогательную процедуру BINOMIAL_HEAP_MERGE, которая объединяет списки корней H_1 и H_2 в единый связанный список, отсортированный по степеням в монотонно возрастающем порядке. Процедура BINOMIAL_HEAP_MERGE, написание которой мы оставляем в качестве упражнения 19.2-1, подобна процедуре MERGE из раздела 2.3.1.

```
BINOMIAL_HEAP_UNION(H1, H2)
 1  H ← MAKE_BINOMIAL_HEAP()
 2  head[H] ← BINOMIAL_HEAP_MERGE(H1, H2)
 3  Освобождение объектов H1 и H2, но не
    списков, на которые они указывают
 4  if head[H] = NIL
 5    then return H
 6  prev-x ← NIL
 7  x ← head[H]
 8  next-x ← sibling[x]
 9  while next-x ≠ NIL
10        do if (degree[x] ≠ degree[next-x]) или
                    (sibling[next-x] ≠ NIL и degree[sibling[next-x]] = degree[x])
```

```
11            then prev-x ← x                                  ▷ Случаи 1 и 2
12                 x ← next-x                                  ▷ Случаи 1 и 2
13            else if key[x] ⩽ key[next-x]
14                   then sibling[x] ← sibling[next-x]         ▷ Случай 3
15                        BINOMIAL_LINK(next-x, x)             ▷ Случай 3
16                   else if prev-x = NIL                      ▷ Случай 4
17                        then head[H] ← next-x                ▷ Случай 4
18                        else sibling[prev-x] ← next-x        ▷ Случай 4
19                        BINOMIAL_LINK(x, next-x)             ▷ Случай 4
20                        x ← next-x                           ▷ Случай 4
21       next-x ← sibling[x]
22 return H
```

На рис. 19.4 показан пример работы процедуры BINOMIAL_HEAP_UNION для всех четырех случаев, приведенных в псевдокоде. На рис. 19.4*а* показаны исходные биномиальные пирамиды H_1 и H_2. Рис. 19.4*б* иллюстрирует биномиальную пирамиду H, которая является результатом работы процедуры BINOMIAL_HEAP_MERGE(H_1, H_2). Изначально x является первым корнем в списке корней H. Поскольку и x, и $next\text{-}x$ имеют степень 0, и $key\,[x] < key\,[next\text{-}x]$, реализуется случай 3. После связывания, как видно из рис. 19.4*в*, x является первым из трех корней с одной и той же степенью, так что реализуется случай 2. После того как все указатели сдвигаются на одну позицию в списке корней (рис. 19.4*г*), реализуется случай 4 (x является первым из двух корней равной степени). После связывания реализуется случай 3 (рис. 19.4*д*). После очередного связывания степень x становится равной 3, а степень $next\text{-}x$ — 4, так что реализуется случай 1 (рис. 19.4*е*). Эта итерация цикла — последняя, поскольку после очередного перемещения указателей $next\text{-}x$ становится равным NIL.

Процедура BINOMIAL_HEAP_UNION имеет две фазы. Первая фаза осуществляется вызовом процедуры BINOMIAL_HEAP_MERGE, объединяет списки корней биномиальных пирамид H_1 и H_2 в единый связанный список H, который отсортирован по степеням корней в монотонно возрастающем порядке. Однако в списке может оказаться по два (но не более) корня каждой степени, так что вторая фаза связывает корни одинаковой степени до тех пор, пока все корни не будут иметь разную степень. Поскольку связанный список H отсортирован по степеням, все операции по связыванию выполняются достаточно быстро.

Более подробно процедура работает следующим образом. Процедура начинается с объединения в строках 1–3 списков корней биномиальных пирамид H_1 и H_2 в единый список корней H. Списки корней H_1 и H_2 отсортированы в строго возрастающем порядке; процедура BINOMIAL_HEAP_MERGE возвращает список корней H, который также отсортирован в строго возрастающем порядке степеней корней. Если списки корней H_1 и H_2 содержат всего m корней, то время работы

процедуры BINOMIAL_HEAP_MERGE составляет $O(m)$. Процедура на каждом шаге сравнивает корни в начале двух списков корней и добавляет в результирующий список корень с меньшей степенью, удаляя его из исходного списка.

Затем процедура BINOMIAL_HEAP_UNION инициализирует ряд указателей в списке корней H. В случае, если происходит слияние двух пустых биномиальных пирамид, в строках 4–5 выполняется возврат пустой биномиальной пирамиды. Начиная со строки 6, нам известно, что в биномиальной пирамиде H имеется по крайней мере один корень. В процедуре поддерживаются три указателя внутрь списка корней.

- x указывает на текущий корень.
- $prev\text{-}x$ указывает на корень, предшествующий x в списке корней: $sibling\,[prev\text{-}x] = x$. Поскольку изначально x не имеет предшественника, мы начинаем работу со значения $prev\text{-}x$, равного NIL.
- $next\text{-}x$ указывает на корень, следующий в списке корней за x: $sibling\,[x] = next\text{-}x$.

Изначально в списке корней H имеется не более двух узлов с данной степенью: поскольку H_1 и H_2 были биномиальными пирамидами, каждая из них имела не более одного корня данной степени. Кроме того, процедура BINOMIAL_HEAP_MERGE гарантирует, что если два корня в H имеют одну и ту же степень, то эти корни соседствуют в списке корней.

В действительности, в процессе работы BINOMIAL_HEAP_UNION в некоторый момент в списке корней H могут оказаться три корня данной степени — вскоре вы увидите, как может реализоваться такая ситуация. При каждой итерации цикла **while** в строках 9–21, мы должны решить, следует ли связывать x и $next\text{-}x$, учитывая их степени и, возможно, степень $sibling\,[next\text{-}x]$. Инвариантом цикла является то, что в начале каждого выполнения тела цикла и x, и $next\text{-}x$ не равны NIL (в упражнении 19.2-4 рассматривается точный инвариант цикла).

Случай 1, показанный на рис. 19.5*а*, реализуется, когда $degree\,[x] \neq degree\,[next\text{-}x]$, т.е. когда x является корнем B_k-дерева, а $next\text{-}x$ — корнем B_l-дерева, причем $l > k$. Эта ситуация обрабатывается в строках 11–12. Мы не связываем x и $next\text{-}x$, так что мы просто смещаем указатели на одну позицию по списку. Обновление указателя $next\text{-}x$, который указывает на корень, следующий за x, выполняется в строке 21, поскольку это действие — общее для всех случаев.

Случай 2 (рис. 19.5*б*) реализуется, когда x является первым из трех корней с одинаковой степенью, т.е. когда

$$degree\,[x] = degree\,[next\text{-}x] = degree\,[sibling\,[next\text{-}x]]\,.$$

Эта ситуация обрабатывается так же, как и случай 1 — мы просто перемещаем указатели по списку. В следующей итерации цикла будет обработан случай 3

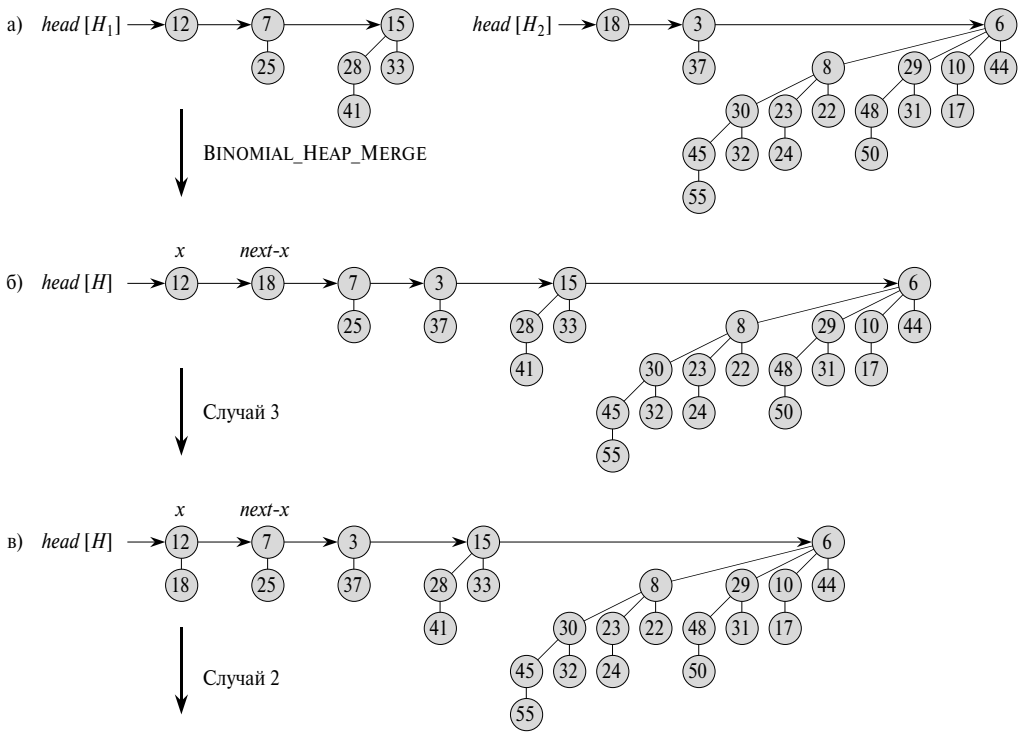


Рис. 19.4. Выполнение процедуры BINOMIAL_HEAP_UNION

или 4, когда объединяются второй и третий из трех корней с одинаковой степенью. В строке 10 выполняется проверка реализации случаев 1 и 2, а в строках 11–12 — их обработка.

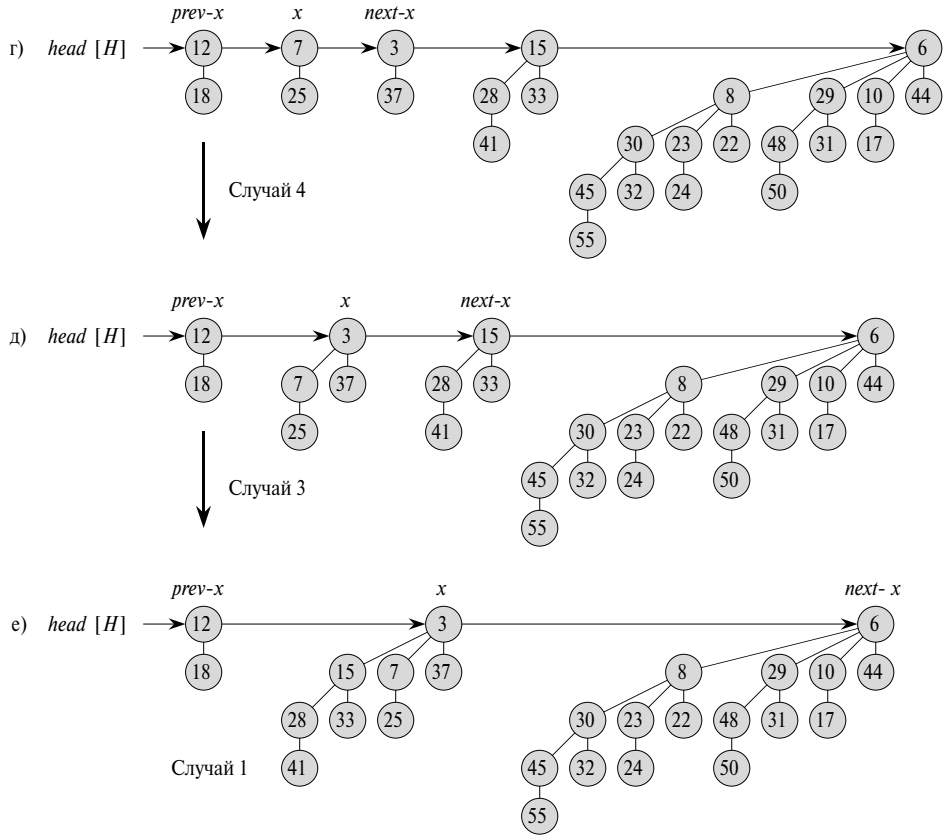
Случаи 3 и 4 реализуются, когда x представляет собой первый из двух корней одинаковой степени, т.е. когда

$$degree\,[x] = degree\,[next\text{-}x] \neq degree\,[sibling\,[next\text{-}x]]\,.$$

Эти случаи могут возникнуть на любой итерации, в частности, всегда — непосредственно после обнаружения случая 2. В случаях 3 и 4 мы связываем x и $next\text{-}x$. Различие между случаями 3 и 4 определяется тем, какой из этих узлов имеет меньший ключ и, соответственно, будет выступать в роли нового корня после связывания.

В случае 3 (рис. 19.5*в*) $key\,[x] \leqslant key\,[next\text{-}x]$, поэтому $next\text{-}x$ привязывается к x. В строке 14 происходит удаление $next\text{-}x$ из списка корней, а в строке 15 $next\text{-}x$ становится крайним левым дочерним узлом x.

В случае 4, показанном на рис. 19.5*г*, $next\text{-}x$ имеет меньший ключ, так что x привязывается к $next\text{-}x$. В строках 16–18 происходит удаление x из списка корней;



в зависимости от того, является x первым элементом списка (строка 17) или нет (строка 18). В строке 19 x делается крайним слева дочерним узлом $next\text{-}x$, а в строке 20 происходит обновление x для следующей итерации.

Настройки для следующей итерации цикла **while** после случаев 3 и 4 одинаковы. Мы должны просто связать два B_k-дерева в B_{k+1}-дерево, на которое указывает x. После него в списке может находиться от нуля до двух B_{k+1}-деревьев, так что x теперь указывает на первое в последовательности из одного, двух или трех B_{k+1}-деревьев в списке корней. Если дерево только одно, на следующей итерации мы получаем случай 1: $degree\,[x] \neq degree\,[next\text{-}x]$. Если x — первое из двух деревьев, то на следующей итерации получаются случаи 3 или 4. И наконец, если x — первое из трех деревьев, то в следующей итерации получаем случай 2.

Время работы процедуры BINOMIAL_HEAP_UNION равно $O(\lg n)$, где n — общее количество узлов в биномиальных пирамидах H_1 и H_2. Убедиться в этом можно следующим образом. Пусть H_1 содержит n_1 узлов, а $H_2 - n_2$ узлов, так что $n = n_1 + n_2$. Тогда H_1 содержит не более $\lfloor \lg n_1 \rfloor + 1$ корней, а H_2 — не более $\lfloor \lg n_2 \rfloor + 1$ корней. Таким образом, H содержит непосредственно после вызова

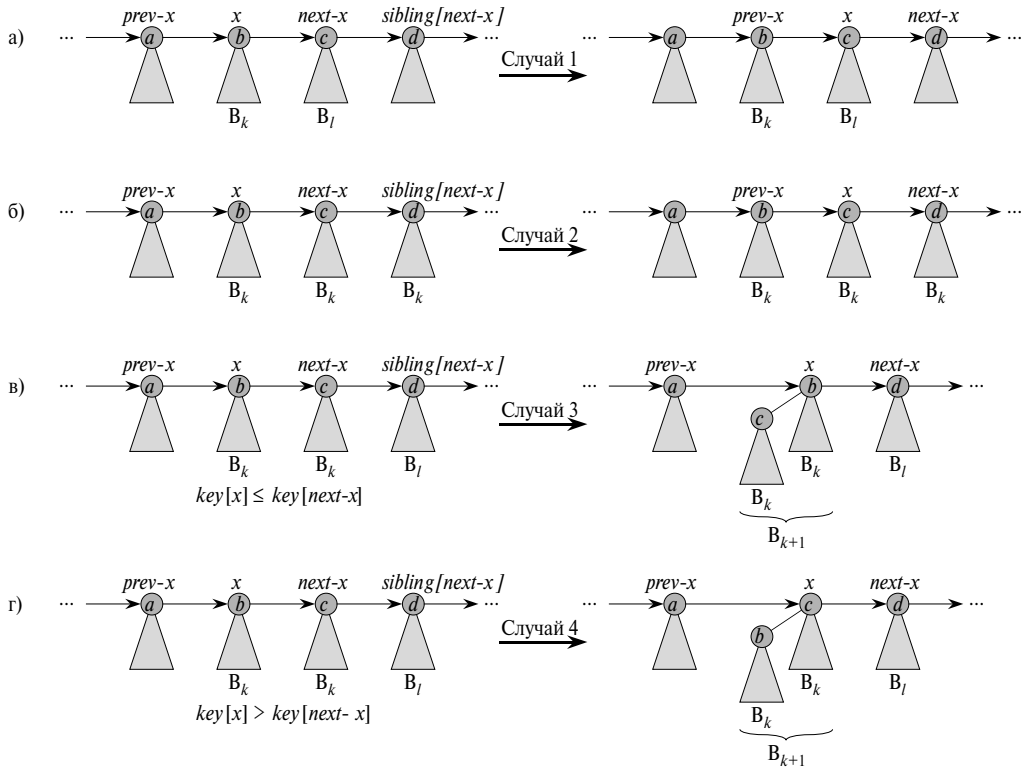


Рис. 19.5. Четыре случая, возможных при выполнении процедуры BINOMIAL_HEAP_UNION

BINOMIAL_HEAP_MERGE не более $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 \leqslant 2 \lfloor \lg n \rfloor + 2 = O(\lg n)$ корней. Следовательно, время работы процедуры BINOMIAL_HEAP_MERGE составляет $O(\lg n)$. Каждая итерация цикла **while** требует $O(1)$ времени, и всего выполняется не более $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$ итераций, поскольку каждая из них в конечном счете приводит к перемещению указателя на одну позицию в списке корней H или к удалению корня из списка. Отсюда следует, что общее время работы процедуры BINOMIAL_HEAP_UNION равно $O(\lg n)$.

Вставка узла

Приведенная далее процедура вставляет узел x в биномиальную пирамиду H (предполагается, что для x уже выделена память и поле $key[x]$ уже заполнено):

```
BINOMIAL_HEAP_INSERT(H, x)
1  H' ← MAKE_BINOMIAL_HEAP()
2  p[x] ← NIL
3  child[x] ← NIL
```

4 $sibling[x] \leftarrow$ NIL
5 $degree[x] \leftarrow 0$
6 $head[H'] \leftarrow x$
7 $H \leftarrow$ BINOMIAL_HEAP_UNION(H, H')

Процедура просто создает биномиальную пирамиду H' с одним узлом за время $O(1)$ и объединяет ее с биномиальной пирамидой H, содержащей n узлов, за время $O(\lg n)$. Вызов BINOMIAL_HEAP_UNION должен освободить память, выделенную для временной биномиальной пирамиды H'. (Реализация вставки узла без привлечения процедуры BINOMIAL_HEAP_UNION оставлена читателю в качестве упражнения 19.2-8.)

Извлечение вершины с минимальным ключом

Приведенная ниже процедура извлекает узел с минимальным ключом из биномиальной пирамиды H и возвращает указатель на извлеченный узел:

BINOMIAL_HEAP_EXTRACT_MIN(H)
1 Поиск корня x с минимальным значением ключа в списке корней H,
и удаление x из списка корней H
2 $H' \leftarrow$ MAKE_BINOMIAL_HEAP()
3 Обращение порядка связанного списка дочерних узлов x,
установка поля p каждого дочернего узла равным NIL
и присвоение указателю $head[H']$ адреса заголовка
получающегося списка
4 $H \leftarrow$ BINOMIAL_HEAP_UNION(H, H')
5 **return** x

Процедура работает так, как показано на рис. 19.6. Исходная биномиальная пирамида H показана на рис. 19.6*а*. На рис. 19.6*б* показана ситуация после выполнения строки 1: корень x с минимальным значением ключа удален из списка корней H. Если x является корнем B_k-дерева, то, в соответствии со свойством 4 из леммы 19.1, потомками x слева направо являются корни биномиальных деревьев $B_{k-1}, B_{k-2}, \ldots, B_0$. На рис. 19.6*в* показано, что, обращая порядок потомков узла x в строке 3, мы получаем биномиальную пирамиду H', которая содержит все узлы дерева x, за исключением самого x. Поскольку корень x удален из H в строке 1, биномиальная пирамида, полученная в результате слияния H и H' в строке 4 (показанная на рис. 19.6*г*), содержит все узлы, которые изначально содержались в H, кроме узла x. И наконец, в строке 5 процедура возвращает значение x.

Поскольку строки 1–4 требуют для выполнения в случае биномиальной пирамиды H с n узлами время $O(\lg n)$, время работы процедуры BINOMIAL_HEAP_EXTRACT_MIN равно $O(\lg n)$.

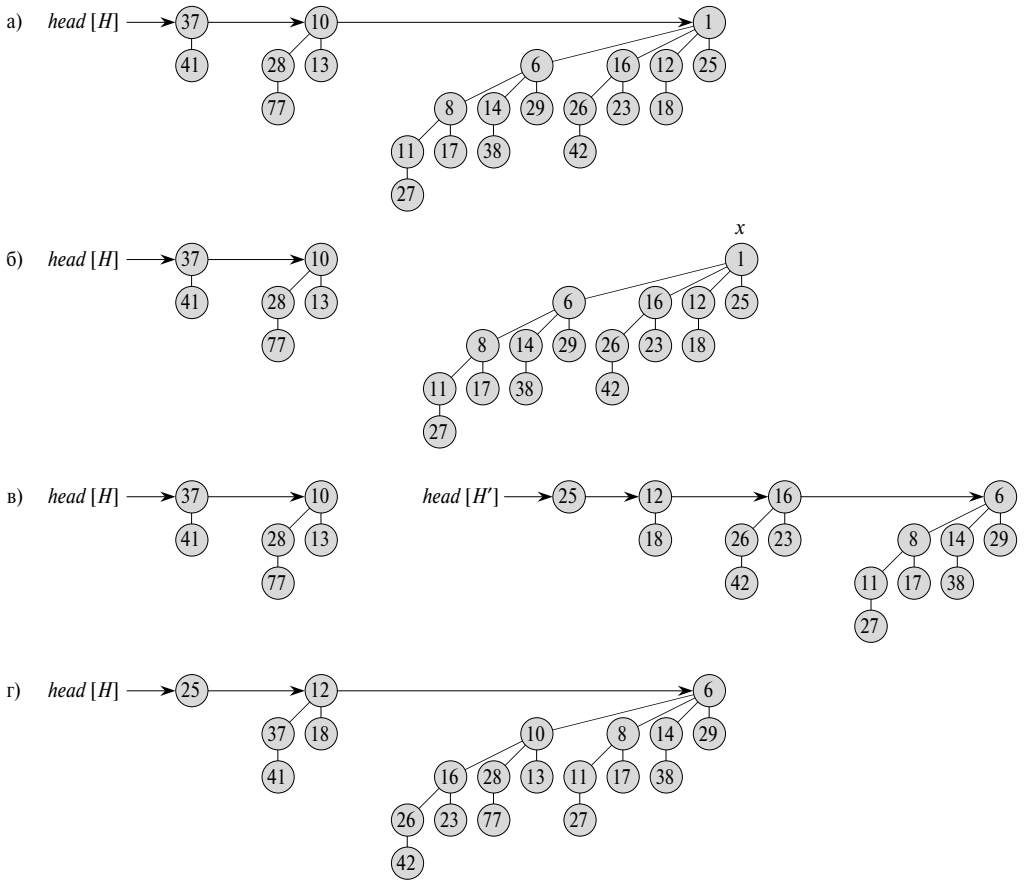


Рис. 19.6. Работа процедуры BINOMIAL_HEAP_EXTRACT_MIN

Уменьшение ключа

Приведенная далее процедура уменьшает значение ключа узла x в биномиальной пирамиде H, присваивая ему новое значение k. В случае, если значение k превышает текущий ключ x, процедура сообщает об ошибке.

BINOMIAL_HEAP_DECREASE_KEY(H, x, k)

```
1  if k > key[x]
2     then error "Новый ключ больше текущего"
3  key[x] ← k
4  y ← x
5  z ← p[y]
6  while z ≠ NIL и  key[y] < key[z]
7        do Обменять  key[y] ↔ key[z]
```

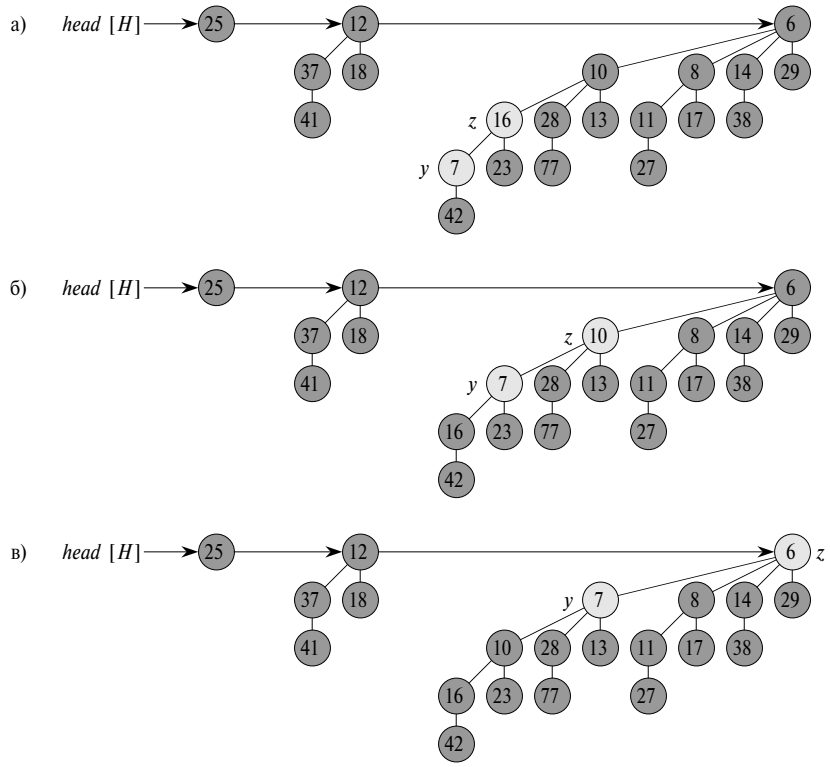


Рис. 19.7. Работа процедуры BINOMIAL_HEAP_DECREASE_KEY

```
 8        ▷ Если y и z содержат сопутствующую
 9        ▷ информацию, обменять также и ее
10        y ← z
11        z ← p[y]
```

Как показано на рис. 19.7, данная процедура уменьшает значение ключа так же, как и в бинарной неубывающей пирамиде — путем "всплывания" ключа в пирамиде. После того, как мы убедимся, что новый ключ не превышает текущий, и присвоим его x, процедура идет вверх по дереву. Изначально значение y равно x. В каждой итерации цикла **while** в строках 6–10 значение $key\,[y]$ сравнивается со значением ключа родительского по отношению к y узла z. Если y является корнем или $key\,[y] \geqslant key\,[z]$, биномиальное дерево является упорядоченным в соответствии со свойством неубывающей пирамиды. В противном случае узел y нарушает свойство неубывающей пирамиды и происходит обмен ключами и сопутствующими данными между узлами y и z, после чего процедура присваивает переменной y значение z и в очередной итерации переходит на один уровень вверх.

Процедура BINOMIAL_HEAP_DECREASE_KEY выполняется за время $O(\lg n)$. В соответствии со свойством 2 леммы 19.1, максимальная глубина x равна $\lfloor \lg n \rfloor$, так что цикл **while** в строках 6–10 выполняется не более $\lfloor \lg n \rfloor$ раз.

Удаление ключа

Удаление ключа x и сопутствующей информации из биномиальной пирамиды H легко выполняется за время $O(\lg n)$. В приведенной ниже реализации этой операции предполагается, что ни один узел в биномиальной пирамиде не имеет ключ со значением $-\infty$.

BINOMIAL_HEAP_DELETE(H, x)
1 BINOMIAL_HEAP_DECREASE_KEY$(H, x, -\infty)$
2 BINOMIAL_HEAP_EXTRACT_MIN(H)

Процедура BINOMIAL_HEAP_DECREASE_KEY делает узел x единственным узлом с минимальным ключом в биномиальной пирамиде, равным $-\infty$ (в упражнении 19.2-6 рассматривается ситуация, когда $-\infty$ не может использоваться в качестве ключа даже временно). Затем этот ключ и связанные с ним сопутствующие данные "всплывают" в биномиальной пирамиде до корня в процедуре BINOMIAL_HEAP_DECREASE_KEY, после чего этот корень удаляется из дерева вызовом процедуры BINOMIAL_HEAP_EXTRACT_MIN.

Время работы процедуры BINOMIAL_HEAP_DELETE равно $O(\lg n)$.

Упражнения

19.2-1. Напишите псевдокод процедуры BINOMIAL_HEAP_MERGE.

19.2-2. Покажите, какая биномиальная пирамида получится в результате вставки узла с ключом 24 в биномиальную пирамиду, показанную на рис. 19.6*г*.

19.2-3. Покажите, какая биномиальная пирамида получится в результате удаления узла с ключом 28 из биномиальной пирамиды, показанной на рис. 19.7*в*.

19.2-4. Докажите корректность процедуры BINOMIAL_HEAP_UNION при помощи следующего инварианта цикла.

В начале каждой итерации цикла **while** в строках 9–21 x указывает на корень, который представляет собой:

- либо единственный корень с данной степенью;
- либо первый из всего двух корней данной степени;
- либо первый или второй из всего трех корней данной степени.

Кроме того, все корни, являющиеся предшественниками предшественника x в списке корней, единственные со своими степенями в списке корней, и если предшественник x имеет степень, отличную от степени x, то эта степень также единственна в списке корней. И, наконец, степени узлов монотонно увеличиваются в процессе прохода по списку корней.

19.2-5. Поясните, почему процедура BINOMIAL_HEAP_MINIMUM может работать некорректно, если ключ может иметь значение ∞. Перепишите псевдокод так, чтобы он корректно работал и в указанной ситуации.

19.2-6. Предположим, что нет никакой возможности представления ключа со значением $-\infty$. Перепишите процедуру BINOMIAL_HEAP_DELETE так, чтобы она корректно работала в указанной ситуации. Время работы процедуры должно составлять $O(\lg n)$.

19.2-7. Рассмотрите взаимосвязь между вставкой в биномиальную пирамиду и увеличением представляющего ее двоичного числа, а также между слиянием двух биномиальных пирамид и суммированием двух двоичных чисел.

19.2-8. С учетом результатов упражнения 19.2-7, перепишите процедуру BINOMIAL_HEAP_INSERT таким образом, чтобы для вставки узла в биномиальную пирамиду не требовался вызов BINOMIAL_HEAP_UNION.

19.2-9. Покажите, что если список корней поддерживается в строго убывающем порядке по степеням (вместо строго возрастающего порядка), то все операции с биномиальными пирамидами можно реализовать без изменения их асимптотического времени работы.

19.2-10. Приведите пример входных данных, для которых процедуры BINOMIAL_HEAP_EXTRACT_MIN, BINOMIAL_HEAP_DECREASE_KEY, BINOMIAL_HEAP_UNION и BINOMIAL_HEAP_DELETE выполняются за время $\Omega(\lg n)$. Поясните, почему время работы процедур BINOMIAL_HEAP_INSERT и BINOMIAL_HEAP_MINIMUM в худшем случае составляет $\overset{\infty}{\Omega}(\lg n)$, а не $\Omega(\lg n)$ (см. задачу 3-5).

Задачи

19-1. **2-3-4-пирамиды**

В главе 18 рассматривались 2-3-4-деревья, в которых каждый внутренний узел (кроме, возможно, корня) имеет два, три или четыре дочерних узла, и все листья таких деревьев располагаются на одной и той же глубине. В данной задаче мы реализуем ***2-3-4-пирамиды***, поддерживающие операции сливаемых пирамид.

2-3-4-пирамиды отличаются от 2-3-4-деревьев следующим. В 2-3-4-пирамидах ключи хранятся только в листьях, и в каждом листе x хранится ровно один ключ в поле $key[x]$. Никакой порядок ключей в листьях не соблюдается, т.е. при перечислении слева направо ключи могут располагаться в произвольном порядке. Каждый внутренний узел x содержит значение $small[x]$, которое равно минимальному значению среди ключей листьев поддерева, корнем которого является x. Корень r содержит поле $height[r]$, в котором хранится высота дерева. И наконец, 2-3-4-пирамиды предназначены для хранения в оперативной памяти, так что при работе с ними не требуются никакие операции чтения или записи на диск.

Реализуйте перечисленные далее операции над 2-3-4-пирамидами. Каждая из операций *а–д* должна выполняться за время $O(\lg n)$ при работе с 2-3-4-пирамидой, в которой содержится n элементов. Операция UNION в части *е* должна выполняться за время $O(\lg n)$, где n — количество элементов в двух входных пирамидах.

а) Операция MINIMUM возвращает указатель на лист с наименьшим значением ключа.

б) Операция DECREASE_KEY уменьшает значение ключа в данном листе x до указанного значения $k \leqslant key[x]$.

в) Операция INSERT вставляет в пирамиду лист x с ключом k.

г) Операция DELETE удаляет из пирамиды лист x.

д) Операция EXTRACT_MIN извлекает из пирамиды лист с наименьшим значением ключа.

е) Операция UNION объединяет две 2-3-4-пирамиды, возвращая образующуюся в результате слияния 2-3-4-пирамиду, уничтожая при этом входные пирамиды.

19-2. **Алгоритм минимального остовного дерева с использованием биномиальных пирамид**

В главе 23 представлены два алгоритма для решения задачи поиска минимального остовного дерева неориентированного графа. Сейчас мы рассмотрим, каким образом для решения этой задачи можно использовать биномиальные пирамиды.

Пусть задан связный неориентированный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbf{R}$, которая ставит в соответствие каждому ребру (u, v) его вес $w(u, v)$. Мы хотим найти минимальное остовное дерево графа G, т.е. ациклическое подмножество $T \subseteq E$, которое соединяет все вершины V и чей общий вес

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

является минимальным.

Далее приведен псевдокод (корректность которого можно доказать с использованием методов из раздела 23.1), строящий минимальное остовное дерево T. Программа хранит разбиение $\{V_i\}$ множества вершин V, а для каждого множества V_i — множество ребер, инцидентных вершинам в V_i:

$$E_i \subseteq \{(u, v) : u \in V_i \text{ или } v \in V_i\} .$$

```
MST(G)
 1  T ← ∅
 2  for (Для) каждой вершины v_i ∈ V[G]
 3       do V_i ← {v_i}
 4          E_i ← {(v_i, v) ∈ E[G]}
 5  while (Пока) имеется более одного множества V_i
 6       do Выбор произвольного множества V_i
 7          Извлечение ребра (u, v) с минимальным весом из E_i
 8          Без потери общности полагаем u ∈ V_i и v ∈ V_j
 9          if i ≠ j
10            then T ← T ∪ {(u, v)}
11                 V_i ← V_i ∪ V_j, уничтожение V_j
12                 E_i ← E_i ∪ E_j
```

Опишите, как реализовать этот алгоритм с использованием биномиальных пирамид для управления множествами вершин и ребер. Требуется ли внесение изменений в представление биномиальных пирамид? Требуется ли добавление к операциям, перечисленным в табл. 19.1, других операций над биномиальными пирамидами? Оцените время работы вашей реализации алгоритма.

Заключительные замечания

Биномиальные пирамиды были разработаны в 1978 году Виллемином (Vuillemin) [307]. Детальное описание их свойств представлено в работах Брауна (Brown) [49, 50].

ГЛАВА 20

Фибоначчиевы пирамиды

В главе 19 мы познакомились с биномиальными пирамидами, для которых время выполнения операций INSERT, MINIMUM, EXTRACT_MIN, UNION, а также DECREASE_KEY и DELETE в худшем случае равно $O(\lg n)$. В этой главе мы познакомимся с фибоначчиевыми пирамидами, которые поддерживают тот же набор операций, но имеют то преимущество, что операции, в которых не требуется удаление, имеют амортизированное время работы, равное $O(1)$.

С теоретической точки зрения фибоначчиевы пирамиды особенно полезны в случае, когда количество операций EXTRACT_MIN и DELETE относительно мало по сравнению с количеством других операций. Такая ситуация возникает во многих приложениях. Например, некоторые алгоритмы в задачах о графах вызывают процедуру DECREASE_KEY для каждого ребра. В плотных графах с большим количеством ребер амортизированное время выполнения DECREASE_KEY, равное $O(1)$, представляет собой существенный выигрыш по сравнению со временем $\Theta(\lg n)$ в наихудшем случае в биномиальных пирамидах. Быстрые алгоритмы для таких задач, как поиск минимального остовного дерева (глава 23) или поиск кратчайшего пути из одной вершины (глава 24), преимущественно опираются на фибоначчиевы пирамиды.

Однако с практической точки зрения программная сложность реализации и высокие значения постоянных множителей в формулах времени работы существенно снижают эффективность применения фибоначчиевых пирамид, делая их для большинства приложений менее привлекательными, чем обычные бинарные (или k-арные) пирамиды. Таким образом, интерес к фибоначчиевым пирамидам в первую очередь сугубо теоретический. Широкое практическое использование могла бы

получить более простая структура данных, но обладающая тем же амортизированным временем работы, что и фибоначчиевы пирамиды.

Подобно биномиальным пирамидам, фибоначчиевы пирамиды представляют собой набор деревьев. В действительности фибоначчиевы пирамиды в определенной степени связаны с биномиальными пирамидами. Если над фибоначчиевой пирамидой не выполняются процедуры DECREASE_KEY и DELETE, то каждое дерево в пирамиде выглядит как биномиальное. Фибоначчиевы пирамиды имеют более слабую структуру, чем биномиальные пирамиды, что обеспечивает улучшенные асимптотические границы времени работы. Поддержка строгой структуры фибоначчиевы пирамиды в результате может быть отложена до того момента, когда выполнение соответствующих действий окажется более удобным. Подобно динамическим таблицам из раздела 17.4, фибоначчиевы пирамиды представляют собой хороший пример структуры данных, разработанной с учетом применения амортизационного анализа.

Материал данной главы предполагает, что вы ознакомились с главой 19, в которой рассматриваются биномиальные пирамиды. Дело в том, что наше представление структуры фибоначчиевых пирамид основано на структуре биномиальных пирамид, а некоторые операции, выполняемые над фибоначчиевыми пирамидами, аналогичны операциям над биномиальными пирамидами.

Так же, как и биномиальные пирамиды, фибоначчиевы пирамиды не способны обеспечить эффективную поддержку процедуры поиска SEARCH, поэтому в процедуры в качестве параметра передается не ключ, а указатель на узел. При использовании фибоначчиевых пирамид в приложении, мы часто храним в каждом элементе пирамиды дескриптор соответствующего объекта приложения, так же как и каждый объект приложения хранит дескриптор соответствующего элемента пирамиды.

В разделе 20.1 дается определение фибоначчиевых пирамид, рассматривается их представление и потенциальная функция, используемая в амортизационном анализе. В разделе 20.2 показано, как реализовать операции сливаемых пирамид, получив при этом амортизированное время работы, показанное в табл. 19.1. Остальные операции, а именно DECREASE_KEY и DELETE, рассматриваются в разделе 20.3. И наконец, раздел 20.4 завершает анализ фибоначчиевых пирамид, а также поясняет, почему они так называются.

20.1 Структура фибоначчиевых пирамид

Так же, как и биномиальная пирамида, ***фибоначчиева пирамида*** (Fibonacci heap) представляет собой набор деревьев, упорядоченных в соответствии со свойством неубывающей пирамиды. Однако в случае фибоначчиевых пирамид деревья

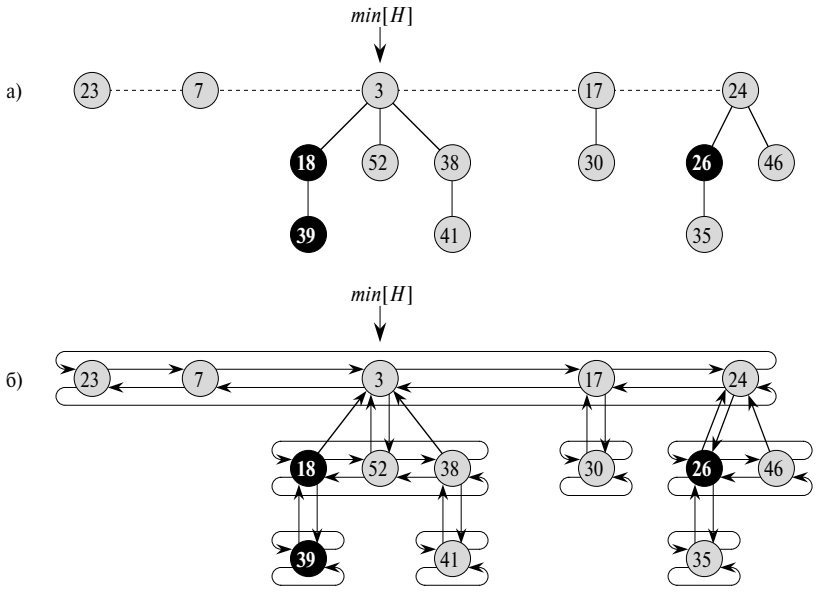


Рис. 20.1. Пример фибоначчиевой пирамиды с пятью деревьями и 14 узлами

не обязательно должны быть биномиальными. На рис. 20.1*а* приведен пример фибоначчиевой пирамиды.

В отличие от деревьев в биномиальных пирамидах, где они упорядочены, деревья в фибоначчиевых пирамидах являются неупорядоченными деревьями с корнем. Как показано на рис. 20.1*б*, каждый узел x содержит указатель $p[x]$ на родительский узел и указатель $child[x]$ на один из дочерних узлов. Дочерние узлы x объединены в один циклический дважды связанный список, который мы назовем ***списком дочерних узлов*** (child list) x. Каждый дочерний узел y в списке дочерних узлов имеет указатели $left[y]$ и $right[y]$, которые указывают на его левый и правый сестринские узлы соответственно. Если узел y является единственным дочерним узлом, то $left[y] = right[y] = y$. Порядок размещения узлов в списке дочерних узлов произволен.

Циклический дважды связанный список (см. раздел 10.2) обладает двумя преимуществами для использования в фибоначчиевых пирамидах. Во-первых, удаление элемента из такого списка выполняется за время $O(1)$. Во-вторых, если имеется два таких списка, их легко объединить в один за то же время $O(1)$. В описании операций над фибоначчиевыми пирамидами мы будем неформально ссылаться на эти операции, оставляя читателю самостоятельно добавить все детали их реализации.

Кроме того, будут использоваться два других поля каждого узла. Количество дочерних узлов x хранится в поле $degree[x]$, а логическое значение $mark[x]$ указывает, были ли потери узлом x дочерних узлов начиная с момента, когда x стал дочерним узлом какого-то другого узла. Вновь создаваемые узлы не помечены, а имеющаяся пометка снимается, если узел становится дочерним узлом какого-то другого узла. До тех пор, пока мы не встретимся с операцией DECREASE_KEY в разделе 20.3, мы считаем поле $mark$ всегда равным FALSE.

Обращение к данной фибоначчиевой пирамиде H выполняется посредством указателя $min[H]$ на корень дерева с минимальным ключом. Этот узел называется ***минимальным узлом*** (minimum node) фибоначчиевой пирамиды. Если фибоначчиева пирамида H пуста, то $min[H] = \text{NIL}$.

Корни всех деревьев в фибоначчиевой пирамиде связаны при помощи указателей $left$ и $right$ в циклический дважды связанный ***список корней*** (root list) фибоначчиевой пирамиды. Указатель $min[H]$, таким образом, указывает на узел списка корней, ключ которого минимален. Порядок деревьев в списке корней произволен.

Фибоначчиева пирамида, кроме того, имеет еще один атрибут — текущее количество узлов в фибоначчиевой пирамиде H хранится в $n[H]$.

Потенциальная функция

Как упоминалось, мы будем использовать метод потенциалов из раздела 17.3 для анализа производительности операций над фибоначчиевыми пирамидами. Для данной фибоначчиевой пирамиды H обозначим через $t(H)$ количество деревьев в списке корней H, а через $m(H)$ — количество помеченных узлов в H. Тогда потенциал фибоначчиевой пирамиды H определяется как

$$\Phi(H) = t(H) + 2m(H). \tag{20.1}$$

(Подробнее о потенциале мы поговорим в разделе 20.3.) Например, потенциал фибоначчиевой пирамиды, показанной на рис. 20.1, равен $5 + 2 \cdot 3 = 11$. Потенциал множества фибоначчиевых пирамид представляет собой сумму потенциалов составляющих его пирамид. Будем считать, что единицы потенциала достаточно для оплаты константного количества работы, где константа достаточно велика для покрытия стоимости любой операции со временем работы $O(1)$.

Кроме того, предполагается, что приложение начинает свою работу, не имея ни одной фибоначчиевой пирамиды, так что начальный потенциал равен 0 и, в соответствии с формулой (20.1), во все последующие моменты времени потенциал неотрицателен. Из (17.3) верхняя граница общей амортизированной стоимости является, таким образом, верхней границей общей фактической стоимости последовательности операций.

Максимальная степень

Амортизационный анализ, который будет выполнен в оставшихся разделах главы, предполагает, что известна верхняя граница $D(n)$ максимальной степени узла в фибоначчиевой пирамиде из n узлов. В упражнении 20.2-3 надо показать, что при поддержке только лишь операций сливаемых пирамид $D(n) \leqslant \lfloor \lg n \rfloor$. В разделе 20.3 мы покажем, что при дополнительной поддержке операций DECREASE_KEY и DELETE $D(n) = O(\lg n)$.

20.2 Операции над сливаемыми пирамидами

В этом разделе мы опишем и проанализируем операции над сливаемыми пирамидами в применении к фибоначчиевым пирамидам. Если поддерживаются только операции MAKE_HEAP, INSERT, MINIMUM, EXTRACT_MIN и UNION, то каждая фибоначчиева пирамида представляет собой набор неупорядоченных биномиальных деревьев. ***Неупорядоченное биномиальное дерево*** (unordered binomial tree) похоже на обычное биномиальное дерево и определяется рекурсивно подобно ему. Неупорядоченное биномиальное дерево U_0 состоит из единственного узла, а неупорядоченное биномиальное дерево U_k состоит из двух неупорядоченных биномиальных деревьев U_{k-1}, причем корень одного из них является *произвольным* дочерним узлом корня другого. Лемма 19.1, в которой описаны свойства биномиальных деревьев, применима и к неупорядоченным биномиальным деревьям, но со следующей вариацией свойства 4 (см. упражнение 20.2-2):

4′. В неупорядоченном биномиальном дереве U_k корень имеет степень k, которая превышает степень любого другого узла. Дочерними узлами корня являются корни поддеревьев $U_0, U_1, \ldots, U_{k-1}$ в некотором порядке.

Таким образом, если фибоначчиева пирамида с n узлами представляет собой набор неупорядоченных биномиальных деревьев, то $D(n) = \lfloor \lg n \rfloor$.

Ключевая идея применения операций над сливаемыми пирамидами к фибоначчиевым пирамидам состоит в том, чтобы по возможности отложить работу на как можно более позднее время. По сути это компромисс среди реализаций различных операций. Если количество деревьев в фибоначчиевой пирамиде невелико, то в процессе выполнения операции EXTRACT_MIN мы можем быстро определить, какой из оставшихся узлов становится минимальным. Однако, как мы видели в упражнении 19.2-10, в случае биномиальных пирамид мы должны заплатить определенную цену за то, что деревья остаются небольшими: требуется время $\Omega(\lg n)$ для того, чтобы вставить узел в биномиальную пирамиду или объединить две биномиальные пирамиды в одну. Как мы увидим, мы не будем пытаться объединять деревья в фибоначчиевых пирамидах при вставке нового узла или слиянии двух пирамид. Такое объединение сохраняется только в операции

EXTRACT_MIN, когда нам действительно нужно будет искать новый минимальный узел.

Создание новой фибоначчиевой пирамиды

Для создания пустой фибоначчиевой пирамиды процедура MAKE_FIB_HEAP выделяет память и возвращает объект фибоначчиевой пирамиды H, причем $n[H] = 0$ и $min[H] =$ NIL. Деревьев в H нет. Поскольку $t(H) = 0$ и $m(H) = 0$, потенциал пустой фибоначчиевой пирамиды $\Phi(H) = 0$. Таким образом, амортизированная стоимость процедуры MAKE_FIB_HEAP равна ее фактической стоимости $O(1)$.

Вставка узла

Приведенная далее процедура вставляет узел x в фибоначчиеву пирамиду H в предположении, что узлу уже выделена память и поле узла $key[x]$ уже заполнено.

```
FIB_HEAP_INSERT(H, x)
 1  degree[x] ← 0
 2  p[x] ← NIL
 3  child[x] ← NIL
 4  left[x] ← x
 5  right[x] ← x
 6  mark[x] ← FALSE
 7  Присоединение списка корней, содержащего x, к списку корней H
 8  if min[H] = NIL или key[x] < key[min[H]]
 9     then min[H] ← x
10  n[H] ← n[H] + 1
```

После того как в строках 1–6 выполняется инициализация полей узла x, создающая собственный циклический дважды связанный список, в строке 7 выполняется добавление x к списку корней H за фактическое время $O(1)$. Таким образом, узел x становится деревом из одного узла в составе фибоначчиевой пирамиды. Он не имеет дочерних узлов и не помечен. В строках 8–9 происходит (если оно необходимо) обновление указателя на минимальный узел, а в строке 10 увеличивается значение общего количества узлов в фибоначчиевой пирамиде $n[H]$ (что отражает добавление нового узла). На рис. 20.2 показана вставка узла с ключом 21 в фибоначчиеву пирамиду, представленную на рис. 20.1.

В отличие от процедуры BINOMIAL_HEAP_INSERT, процедура FIB_HEAP_INSERT не пытается объединять деревья в фибоначчиевой пирамиде. Если последовательно будут выполнены k операций FIB_HEAP_INSERT, то к списку корней будут добавлены k деревьев, состоящих из одного узла.

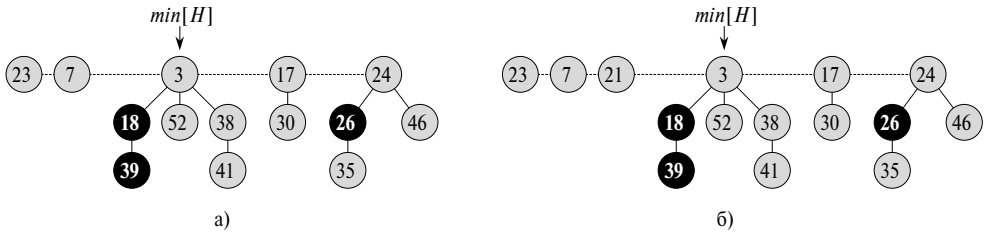


Рис. 20.2. Вставка узла в фибоначчиеву пирамиду

Для определения амортизированной стоимости процедуры FIB_HEAP_INSERT рассмотрим исходную фибоначчиеву пирамиду H и фибоначчиеву пирамиду H', которая получается в результате вставки узла. $t(H') = t(H) + 1$ и $m(H') = m(H)$, и увеличение потенциала составляет

$$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1.$$

Поскольку фактическая стоимость равна $O(1)$, амортизированная стоимость равна $O(1) + 1 = O(1)$.

Поиск минимального узла

На минимальный узел фибоначчиевой пирамиды H указывает указатель $min[H]$, так что поиск минимального узла занимает время $O(1)$. Поскольку потенциал H при этом не изменяется, амортизированная стоимость этой операции равна ее фактической стоимости $O(1)$.

Объединение двух фибоначчиевых пирамид

Приведенная далее процедура объединяет фибоначчиевы пирамиды H_1 и H_2, попросту соединяя списки корней H_1 и H_2 и находя затем новый минимальный узел:

FIB_HEAP_UNION(H_1, H_2)
1 $H \leftarrow$ MAKE_FIB_HEAP()
2 $min[H] \leftarrow min[H_1]$
3 Добавление списка корней H_2 к списку корней H
4 **if** ($min[H_1] =$ NIL) или ($min[H_2] \neq$ NIL и $key[min[H_2]] < key[min[H_1]]$)
5 **then** $min[H] \leftarrow min[H_2]$
6 $n[H] \leftarrow n[H_1] + n[H_2]$
7 Освобождение объектов H_1 и H_2
8 **return** H

В строках 1–3 алгоритм объединяет списки корней H_1 и H_2 в новый список корней H. Строки 2, 4 и 5 определяют минимальный узел H, а в строке 6 выполняется присвоение $n[H]$ общего количества узлов фибоначчиевой пирамиды. В строке 7 выполняется освобождение исходных фибоначчиевых пирамид H_1 и H_2, а в строке 8 — возврат получившейся в результате слияния фибоначчиевой пирамиды H. Как и в случае процедуры FIB_HEAP_INSERT, никакое объединение деревьев не выполняется.

Изменение потенциала равно

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) &= \\ &= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) = \\ &= 0,\end{aligned}$$

поскольку $t(H) = t(H_1) + t(H_2)$ и $m(H) = m(H_1) + m(H_2)$. Амортизированная стоимость процедуры FIB_HEAP_UNION, таким образом, равна фактической стоимости $O(1)$.

Извлечение минимального узла

Процесс извлечения минимального узла наиболее сложный из всех операций, рассматриваемых в данном разделе. Это также то место, где выполняются отложенные действия по объединению деревьев. Псевдокод процедуры извлечения минимального узла приведен ниже. Для удобства предполагается, что когда узел удаляется из связанного списка, указатели, остающиеся в списке, обновляются, но указатели в извлекаемом узле остаются неизменными. В этой процедуре используется вспомогательная процедура CONSOLIDATE, которая будет представлена чуть позже.

```
FIB_HEAP_EXTRACT_MIN(H)
 1  z ← min[H]
 2  if z ≠ NIL
 3     then for (для) каждого дочернего по отношению к z узла x
 4               do Добавить x в список корней H
 5                  p[x] ← NIL
 6          Удалить z из списка корней H
 7          if z = right[z]
 8             then min[H] ← NIL
 9             else  min[H] ← right[z]
10                   CONSOLIDATE(H)
11          n[H] ← n[H] − 1
12  return z
```

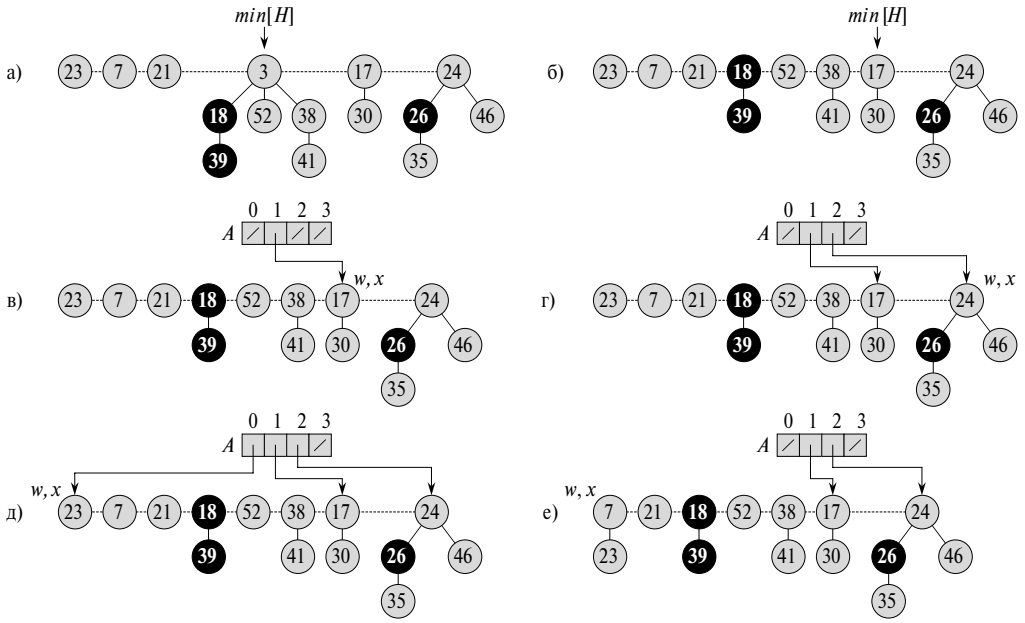
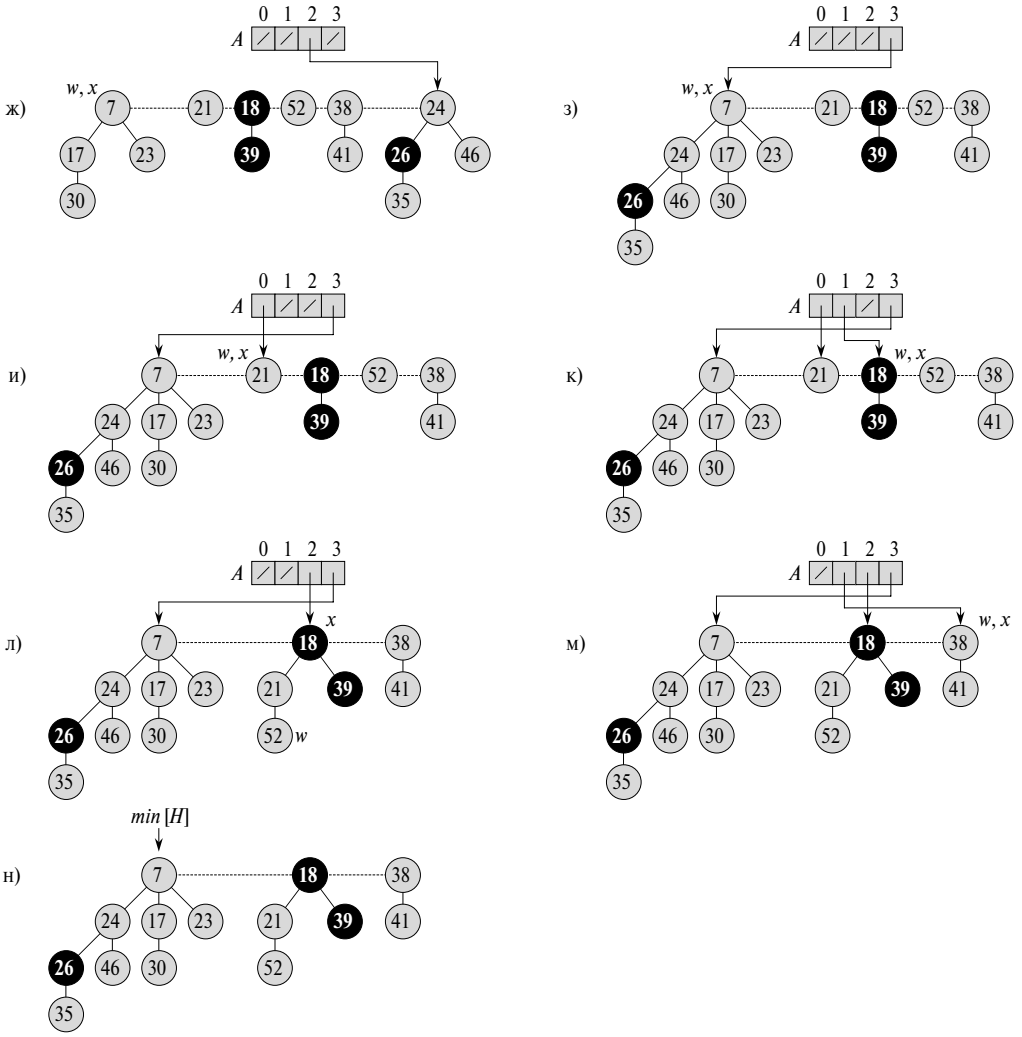


Рис. 20.3. Работа процедуры FIB_HEAP_EXTRACT_MIN

Как показано на рис. 20.3, процедура FIB_HEAP_EXTRACT_MIN сначала перемещает в список корней все дочерние узлы минимального узла, а затем удаляет последний из списка корней. Затем выполняется уплотнение списка корней путем связывания корней одинаковой степени, пока в списке останется не больше одного корня каждой степени.

Работа начинается в строке 1, где в переменной z сохраняется указатель на минимальный узел фибоначчиевой пирамиды — именно этот указатель процедура вернет в конце работы. Если $z = \text{NIL}$, это означает, что фибоначчиева пирамида пуста, и работа на этом заканчивается. В противном случае , как и в процедуре BINOMIAL_HEAP_EXTRACT_MIN, мы удаляем узел z из H, делая все дочерние узлы узла z корнями в H в строках 3–5, помещая их в список корней, и вынося из него z в строке 6. Если после выполнения строки 6 $z = right\,[z]$, значит, z был единственным узлом в списке корней, и у него нет дочерних узлов. В этом случае нам остается только сделать фибоначчиеву пирамиду пустой, перед тем как вернуть z в строке 8. В противном случае мы устанавливаем указатель $min\,[H]$ на узел, отличный от z (в нашем случае это $right\,[z]$), причем это не обязательно минимальный узел. На рис. 20.3*б* показано состояние исходной фибоначчиевой пирамиды, приведенной на рис. 20.3*а*, после выполнения строки 9.

Следующий этап, на котором будет уменьшено количество деревьев в фибоначчиевой пирамиде, — ***уплотнение*** (consolidating) списка корней H, которое



выполняется вспомогательной процедурой CONSOLIDATE(H). Уплотнение списка корней состоит в многократном выполнении следующих шагов, до тех пор, пока все корни в списке корней не будут иметь различные значения поля $degree$.

1. Найти в списке корней два корня x и y с одинаковой степенью, где $key\,[x] \leqslant \leqslant key\,[y]$.
2. ***Привязать*** (link) y к x: удалить y из списка корней и сделать его дочерним узлом x. Эта операция выполняется процедурой FIB_HEAP_LINK. Поле $degree\,[x]$ при этом увеличивается, а пометка y, если таковая была, снимается.

Процедура CONSOLIDATE использует вспомогательный массив $A[0..D(n[H])]$. Если $A[i] = y$, то y в настоящий момент является корнем со степенью $degree[y] = i$.

```
CONSOLIDATE(H)
 1  for i ← 0 to D(n[H])
 2       do A[i] ← NIL
 3  for (для) каждого узла w в списке корней H
 4       do x ← w
 5          d ← degree[x]
 6          while A[d] ≠ NIL
 7               do y ← A[d]          ▷ Узел с той же степенью, что и у x.
 8                  if key[x] > key[y]
 9                    then обменять x ↔ y
10                  FIB_HEAP_LINK(H, y, x)
11                  A[d] ← NIL
12                  d ← d + 1
13          A[d] ← x
14  min[H] ← NIL
15  for i ← 0 to D(n[H])
16       do if A[i] ≠ NIL
17             then Добавить A[i] в список корней H
18                  if min[H] = NIL или key[A[i]] < key[min[H]]
19                    then min[H] ← A[i]
```

```
FIB_HEAP_LINK(H, y, x)
1  Удалить y из списка корней H
2  Сделать y дочерним узлом x, увеличить degree[x]
3  mark[y] ← FALSE
```

Рассмотрим процедуру CONSOLIDATE более подробно. В строках 1–2 выполняется инициализация A путем присвоения каждому элементу массива значения NIL. Цикл **for** в строках 3–13 обрабатывает каждый корень w в списке корней. Обработка каждого корня приводит к созданию дерева, корнем которого является некоторый узел x, который может как совпадать, так и не совпадать с w. После этого ни один корень с списке корней не имеет ту же степень, что и x, так что мы можем присвоить элементу массива $A[degree[x]]$ значение x. По завершении работы цикла **for** в списке корней останется не более чем по одному корню каждой степени, и элементы массива A будут указывать на каждый из оставшихся в списке корней.

Цикл **while** в строках 6–12 связывает корень x дерева, в котором содержится узел w, с другим деревом, корень которого имеет ту же степень, что и x. Это

действие повторяется до тех пор, пока ни один другой корень не будет иметь ту же степень, что и x. Инвариант цикла **while** следующий:

В начале каждой итерации цикла **while** $d = degree[x]$.

Воспользуемся этим инвариантом цикла для доказательства корректности алгоритма.

Инициализация: строка 5 гарантирует, что инвариант цикла выполняется при входе в цикл.

Сохранение: в каждой итерации цикла **while** $A[d]$ указывает на некоторый корень y. Поскольку $d = degree[x] = degree[y]$, мы хотим связать x и y. В результате связывания тот из корней, значение ключа которого меньше, становится родительским узлом для другого, так что в строках 8–9 при необходимости выполняется обмен x и y. Затем мы привязываем y к x вызовом процедуры FIB_HEAP_LINK(H, y, x) в строке 10. Этот вызов увеличивает значение $degree[x]$, но оставляет равным d значение $degree[y]$. Поскольку узел y больше не является корнем, указатель на него удаляется из массива A в строке 11. Поскольку вызов процедуры FIB_HEAP_LINK увеличивает значение $degree[x]$, в строке 12 происходит восстановление инварианта $d = degree[x]$.

Завершение: мы повторяем цикл **while** до тех пор, пока не получим $A[d] = \text{NIL}$, так что ни один другой корень не имеет ту же степень, что и x.

После завершения цикла **while** мы присваиваем $A[d]$ значение x в строке 13 и выполняем очередную итерацию цикла **for**.

На рис. 20.3*в–д* показан массив A и деревья, которые получаются в результате первых трех итераций цикла **for** в строках 3–13. В следующей итерации цикла **for** выполняется три связывания; их результаты можно увидеть на рис. 20.3*е–з*. На рис. 20.3*и–м* представлены результаты следующих четырех итераций цикла **for**.

Теперь все, что остается, — выполнить завершающие действия. После того как закончена работа цикла в строках 3–13, строка 14 создает пустой список, который заполняется в строках 15–19 на основе данных из массива A. Получившаяся в результате фибоначчиева пирамида показана на рис. 20.3*н*. После уплотнения списка корней процедура FIB_HEAP_EXTRACT_MIN завершается уменьшением $n[H]$ в строке 11 и возвратом указателя на удаленный узел z в строке 12.

Заметим, что если перед выполнением процедуры FIB_HEAP_EXTRACT_MIN все деревья в фибоначчиевой пирамиде были неупорядоченными биномиальными деревьями, то и после выполнения процедуры они останутся неупорядоченными биномиальными деревьями. Имеется два пути изменения деревьев. Во-первых, в строках 3–5 процедуры FIB_HEAP_EXTRACT_MIN каждый дочерний узел x корня

z становится корнем. В соответствии с упражнением 20.2-2, каждое новое дерево является неупорядоченным биномиальным деревом. Во-вторых, деревья объединяются процедурой FIB_HEAP_LINK, только если у них одинаковые степени. Поскольку все деревья перед связыванием представляют собой неупорядоченные биномиальные деревья, два дерева с корнями, которые имеют по k дочерних узла, должны иметь структуру U_k. Следовательно, получающееся в результате дерево должно иметь структуру U_{k+1}.

Теперь мы готовы показать, что амортизированная стоимость извлечения минимального узла из фибоначчиевой пирамиды с n узлами равна $O(D(n))$. Обозначим через H фибоначчиеву пирамиду непосредственно перед выполнением операции FIB_HEAP_EXTRACT_MIN.

Фактическую стоимость извлечения минимального узла можно подсчитать следующим образом. Вклад $O(D(n))$ вносит обработка не более $D(n)$ дочерних узлов минимального узла в процедуре FIB_HEAP_EXTRACT_MIN, а также операции в строках 1–2 и 14–19 процедуры CONSOLIDATE. Остается проанализировать вклад цикла **for** в строках 3–13. Размер списка корней при вызове процедуры CONSOLIDATE не превышает $D(n)+t(H)-1$, поскольку он состоит из исходного списка корней с $t(H)$ узлами, минус извлеченный узел и плюс дочерние узлы извлеченного узла, количество которых не превышает $D(n)$. Всякий раз при выполнении цикла **while** в строках 6–12 один из корней связывается с другим, так что общее количество работы, выполняемой в цикле **for**, не более чем пропорционально $D(n)+t(H)$. Следовательно, общая фактическая работа по извлечению минимального узла равна $O(D(n)+t(H))$.

Потенциал перед извлечением минимального узла равен $t(H)+2m(H)$, а после — не превышает $(D(n)+1)+2m(H)$, поскольку остается не более $D(n)+1$ корней, и не имеется узлов, которые были бы помечены во время выполнения этой операции. Таким образом, амортизированная стоимость не превосходит величину

$$\begin{aligned} O(D(n)+t(H))+((D(n)+1)+2m(H))-(t(H)+2m(H)) = \\ = O(D(n))+O(t(H))-t(H)=O(D(n)), \end{aligned}$$

поскольку можно масштабировать единицы потенциала таким образом, чтобы можно было пренебречь константой, скрытой в $O(t(H))$. Интуитивно понятно, что стоимость выполнения каждого связывания является платой за снижение потенциала из-за уменьшения количества корней на 1 при связывании. В разделе 20.4 мы увидим, что $D(n)=O(\lg n)$, так что амортизированная стоимость извлечения минимального узла равна $O(\lg n)$.

Упражнения

20.2-1. Изобразите фибоначчиеву пирамиду, которая получается в результате применения процедуры FIB_HEAP_EXTRACT_MIN к фибоначчиевой пирамиде, показанной на рис. 20.3*н*.

20.2-2. Докажите, что лемма 19.1 выполняется для неупорядоченных биномиальных деревьев, если свойство 4 заменить свойством 4*/*.

20.2-3. Покажите, что при поддержке только операций над сливаемыми пирамидами максимальная степень $D(n)$ в фибоначчиевой пирамиде с n узлами не превышает $\lfloor \lg n \rfloor$.

20.2-4. Профессор разработал новую структуру данных, основанную на фибоначчиевых пирамидах. Пирамида имеет ту же структуру, что и фибоначчиева пирамида, и поддерживает операции над сливаемыми пирамидами. Реализация операций такая же, как и в фибоначчиевых пирамидах, но с тем отличием, что в качестве последнего шага при вставке узла и объединении пирамид выполняется уплотнение списка корней. Чему равно наихудшее время выполнения операций над пирамидами профессора? Насколько нова предложенная им структура данных?

20.2-5. Считая, что единственная доступная нам операция над ключами состоит в их сравнении (как в случае всех реализаций в данной главе), докажите, что невозможно обеспечить выполнение всех операций над сливаемыми пирамидами за амортизированное время $O(1)$.

20.3 Уменьшение ключа и удаление узла

В этом разделе будет показано, как уменьшить ключ узла фибоначчиевой пирамиды за амортизированное время $O(1)$ и как удалить произвольный узел из фибоначчиевой пирамиды с n узлами за амортизированное время $O(D(n))$. Эти операции нарушают свойство фибоначчиевых пирамид, которое заключается в том, что все деревья в фибоначчиевых пирамидах являются неупорядоченными биномиальными деревьями. Однако они остаются достаточно близки к таковым, так что мы можем ограничить максимальную степень $D(n)$ величиной $O(\lg n)$. Этот факт, который будет доказан в разделе 20.4, приводит к тому, что операции FIB_HEAP_EXTRACT_MIN и FIB_HEAP_DELETE имеют амортизированное время работы, равное $O(\lg n)$.

Уменьшение ключа

В приведенном далее псевдокоде для операции FIB_HEAP_DECREASE_KEY, как и ранее, предполагается, что при удалении узла из связанного списка никакие его поля не изменяются.

```
FIB_HEAP_DECREASE_KEY(H, x, k)
1  if k > key[x]
2     then error "Новый ключ больше текущего"
3  key[x] ← k
4  y ← p[x]
5  if y ≠ NIL и key[x] < key[y]
6     then CUT(H, x, y)
7          CASCADING-CUT(H, y)
8  if key[x] < key[min[H]]
9     then min[H] ← x
```

```
CUT(H, x, y)
1  Удаление x из списка дочерних узлов y, уменьшение degree[y]
2  Добавление x в список корней H
3  p[x] ← NIL
4  mark[x] ← FALSE
```

```
CASCADING_CUT(H, y)
1  z ← p[y]
2  if z ≠ NIL
3     then if mark[y] = FALSE
4             then mark[y] ← TRUE
5             else CUT(H, y, z)
6                  CASCADING_CUT(H, z)
```

Процедура FIB_HEAP_DECREASE_KEY работает следующим образом. В строках 1–3 проверяется, не является ли новый ключ больше старого, и если нет, то полю $key\,[x]$ присваивается значение нового ключа. Если x — корень или если $key\,[x] \geqslant key\,[y]$, где y — родитель x, то никакие структурные изменения не нужны, поскольку свойство неубывающих пирамид не нарушено. Это условие проверяется в строках 4–5.

Если же свойство неубывающих пирамид оказывается нарушенным, требуется внести множество изменений. Мы начинаем с ***вырезания*** (cutting) x в строке 6. Процедура CUT "вырезает" связь между x и его родительским узлом y, делая x корнем.

Поле $mark$ используется для получения желаемого времени работы. В нем хранится маленькая часть истории каждого узла. Предположим, что с узлом x произошли следующие события.

1. В некоторый момент времени x был корнем,
2. затем x был привязан к другому узлу,
3. после чего два дочерних узла x были вырезаны.

Как только x теряет второй дочерний узел, мы вырезаем x у его родителя, делая x новым корнем. Поле $mark\,[x]$ равно TRUE, если произошли события 1 и 2 и у x вырезан только один дочерний узел. Процедура CUT, следовательно, в строке 4 должна очистить поле $mark\,[x]$, поскольку произошло событие 1. (Теперь становится понятно, почему в строке 3 процедуры FIB_HEAP_LINK выполняется сброс поля $mark\,[y]$: узел y оказывается связан с другим узлом, т.е. выполняется событие 2. Затем, когда будет вырезаться дочерний узел у узла y, полю $mark\,[y]$ будет присвоено значение TRUE.)

Мы сделали еще не всю работу, поскольку x может быть вторым дочерним узлом, вырезанным у его родительского узла y с того момента, когда y был привязан к другому узлу. Поэтому в строке 7 процедуры FIB_HEAP_DECREASE_KEY делается попытка выполнить операцию ***каскадного вырезания*** (cascading-cut) над y. Если y — корень, то проверка в строке 2 процедуры CASCADING_CUT заставляет ее прекратить работу. Если y не помечен, процедура помечает его в строке 4, поскольку его первый дочерний узел был только что вырезан, после чего также прекращает работу. Однако если узел уже был помечен, значит, только что он потерял второй дочерний узел. Тогда y вырезается в строке 5, и процедура CASCADING_CUT в строке 6 рекурсивно вызывает себя для узла z, родительского по отношению к узлу y. Процедура CASCADING_CUT рекурсивно поднимается вверх по дереву до тех пор, пока не достигает корня или непомеченного узла.

После того как выполнены все каскадные вырезания, в строках 8–9 процедуры FIB_HEAP_DECREASE_KEY при необходимости обновляется величина $min\,[H]$. Единственным узлом, чей ключ изменяется в процессе работы процедуры, является узел x, так что минимальным узлом может быть только исходный минимальный узел или узел x.

На рис. 20.4 показано выполнение двух процедур FIB_HEAP_DECREASE_KEY над фибоначчиевой пирамидой, показанной на рис. 20.4*а*. Первый вызов, показанный на рис. 20.4*б* и уменьшающий ключ 46 до 15, выполняется без каскадного вырезания. При втором вызове (рис. 20.4*в–д*), уменьшающем ключ 35 до 5, выполняются два каскадных вырезания.

Покажем теперь, что амортизированная стоимость процедуры FIB_HEAP_DECREASE_KEY составляет только $O\,(1)$. Начнем с определения фактической стоимости. Процедура FIB_HEAP_DECREASE_KEY занимает $O\,(1)$ времени, плюс время, необходимое для каскадного вырезания. Предположим, что процедура CASCADING_CUT в данном вызове FIB_HEAP_DECREASE_KEY рекурсивно вызывается c раз. Каждый вызов CASCADING_CUT без учета рекурсии требует $O\,(1)$ времени. Следовательно, фактическая стоимость процедуры FIB_HEAP_DECREASE_KEY составляет $O\,(c)$ с учетом всех рекурсивных вызовов.

Вычислим теперь изменение потенциала. Обозначим через H фибоначчиеву пирамиду непосредственно перед вызовом процедуры FIB_HEAP_DECREASE_KEY. Каждый рекурсивный вызов CASCADING_CUT, за исключением последнего,

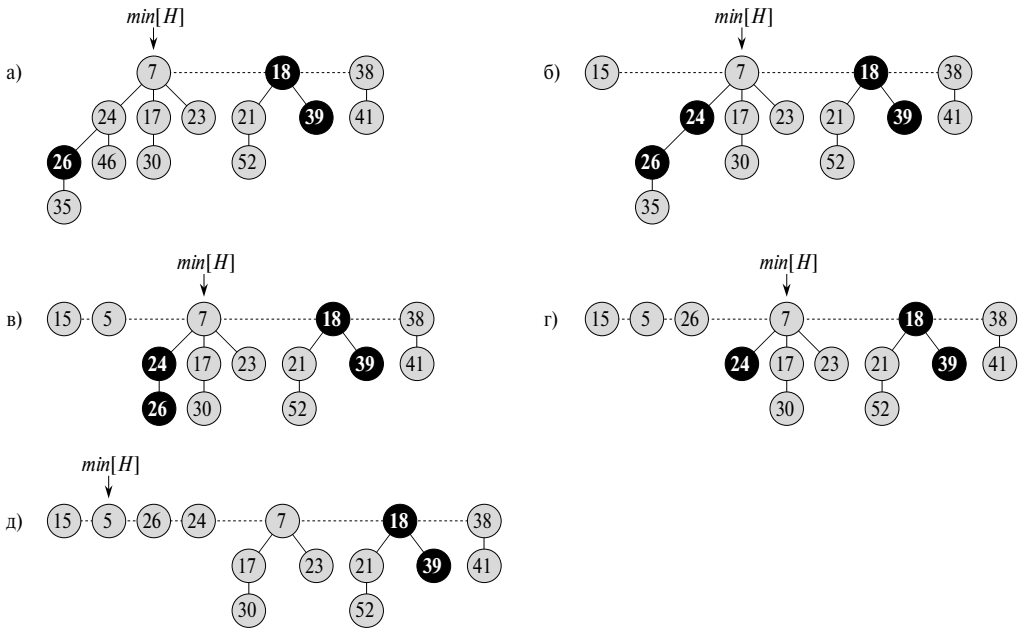


Рис. 20.4. Два вызова процедуры FIB_HEAP_DECREASE_KEY

вырезает помеченный узел и сбрасывает бит метки. После этого имеется $t(H)+c$ деревьев (исходные $t(H)$ деревьев, $c-1$ деревьев, полученных при каскадном вырезании, и дерево, корнем которого является x) и не более $m(H)-c+2$ помеченных узла (у $c-1$ узлов пометка была снята при каскадном вырезании, и последний вызов процедуры CASCADING_CUT может привести к пометке узла). Таким образом, изменение потенциала не превышает величины

$$((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H))=4-c.$$

Следовательно, амортизированная стоимость FIB_HEAP_DECREASE_KEY не превышает $O(c)+4-c=O(1)$, поскольку мы можем соответствующим образом масштабировать единицы.

Теперь вы видите, почему потенциальная функция определена таким образом, что включает член, равный удвоенному количеству помеченных узлов. Когда помеченный узел y вырезается в процессе каскадного вырезания, его пометка снимается, что приводит к снижению потенциала на 2. Одна единица потенциала служит платой за вырезание и снятие пометки, а вторая компенсирует увеличение потенциала из-за того, что y становится корнем.

Удаление узла

Удаление узла из фибоначчиевой пирамиды с n узлами за амортизированное время $O(D(n))$ легко выполняется при помощи приведенного ниже псевдокода. Предполагается, что в фибоначчиевой пирамиде нет ни одного ключа со значением $-\infty$.

FIB_HEAP_DELETE(H, x)
1 FIB_HEAP_DECREASE_KEY$(H, x, -\infty)$
2 FIB_HEAP_EXTRACT_MIN(H)

Процедура FIB_HEAP_DELETE аналогична процедуре BINOMIAL_HEAP_DELETE. Она делает x минимальным узлом фибоначчиевой пирамиды, присваивая его ключу значение $-\infty$, после чего узел x удаляется из фибоначчиевой пирамиды при помощи процедуры FIB_HEAP_EXTRACT_MIN. Амортизированное время работы FIB_HEAP_DELETE представляет собой сумму амортизированного времени работы $O(1)$ процедуры FIB_HEAP_DECREASE_KEY и амортизированного времени работы $O(D(n))$ процедуры FIB_HEAP_EXTRACT_MIN. Поскольку в разделе 20.4 мы увидим, что $D(n) = O(\lg n)$, амортизированное время работы процедуры FIB_HEAP_DELETE равно $O(\lg n)$.

Упражнения

20.3-1. Предположим, что корень x в фибоначчиевой пирамиде помечен. Поясните, как узел x мог стать помеченным корнем. Покажите, что тот факт, что x помечен, не имеет никакого значения для анализа, даже если это не корень, который сначала был привязан к другому узлу, а потом потерял один дочерний узел.

20.3-2. Докажите оценку $O(1)$ амортизированного времени работы процедуры FIB_HEAP_DECREASE_KEY как средней стоимости операции с использованием группового анализа.

20.4 Оценка максимальной степени

Для доказательства того факта, что амортизированное время работы процедур FIB_HEAP_EXTRACT_MIN и FIB_HEAP_DELETE равно $O(\lg n)$, мы должны показать, что верхняя граница $D(n)$ степени произвольного узла в фибоначчиевой пирамиде с n узлами равна $O(\lg n)$. Согласно упражнению 20.2-3, если все деревья в фибоначчиевой пирамиде являются неупорядоченными биномиальными деревьями, то $D(n) = \lfloor \lg n \rfloor$. Однако вырезания в процедуре FIB_HEAP_DECREASE_KEY могут привести к нарушению свойств неупорядоченных биномиальных деревьев. В этом разделе мы покажем, что в связи с тем, что мы вырезаем узел у родителя,

как только он теряет два дочерних узла, $D(n)$ равно $O(\lg n)$. В частности, мы покажем, что $D(n) \leqslant \lfloor \log_\phi n \rfloor$, где $\phi = (1+\sqrt{5})/2$.

Для каждого узла x в фибоначчиевой пирамиде определим $size(x)$ как количество узлов в поддереве, корнем которого является x, включая сам узел x (заметим, что узел x не обязательно должен находиться в списке корней; это может быть любой узел фибоначчиевой пирамиды). Покажем, что величина $size(x)$ экспоненциально зависит от $degree[x]$ (напомним, что поле $degree[x]$ всегда содержит точную величину степени x).

Лемма 20.1. Пусть x — произвольный узел фибоначчиевой пирамиды, и пусть $y_1, y_2, \ldots, y_k$ — дочерние узлы x в порядке их связывания с x начиная с более ранних и заканчивая более поздними. Тогда $degree[y_1] \geqslant 0$ и $degree[y_i] \geqslant i-2$ при $i = 2, 3, \ldots, k$.

Доказательство. Очевидно, что $degree[y_1] \geqslant 0$. Для $i \geqslant 2$ заметим, что когда y_i связывается с x, все узлы $y_1, y_2, \ldots, y_{i-1}$ являются дочерними узлами x, так что в этот момент $degree[x] \geqslant i-1$. Узел y_i связывается с x только в том случае, когда $degree[x] = degree[y_i]$, так что в момент связывания $degree[y_i] \geqslant i-1$. С этого момента узел y_i мог потерять не более одного дочернего узла, поскольку при потере двух дочерних узлов он должен быть вырезан у узла x. Отсюда следует, что $degree[y_i] \geqslant i-2$. ■

Сейчас мы подошли к той части анализа, которая поясняет название "фибоначчиевы пирамиды". Вспомним, что в разделе 3.2 k-ое число Фибоначчи определяется при помощи следующего рекуррентного соотношения:

$$F_k = \begin{cases} 0 & \text{при } k = 0, \\ 1 & \text{при } k = 1, \\ F_{k-1} + F_{k-2} & \text{при } k \geqslant 2. \end{cases}$$

Приведенная далее лемма дает еще один способ для выражения F_k.

Лемма 20.2. Для всех целых $k \geqslant 0$

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i.$$

Доказательство. Докажем лемму при помощи математической индукции по k. При $k = 0$

$$1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1 + 0 = 1 = F_2.$$

По индукции предполагаем, что $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$. Тогда

$$F_{k+2} = F_k + F_{k+1} = F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) = 1 + \sum_{i=0}^{k} F_i.$$

■

Приведенная далее лемма и следствие из нее завершают анализ. Они используют доказанное в упражнении 3.2-7 неравенство

$$F_{k+2} \geqslant \phi^k,$$

где ϕ — золотое сечение, определенное в уравнении (3.22) как $\phi = \left(1 + \sqrt{5}\right)/2 = 1.61803\ldots$.

Лемма 20.3. Пусть x — произвольный узел фибоначчиевой пирамиды, а $k = degree\,[x]$ — ее степень. Тогда $size\,(x) \geqslant F_{k+2} \geqslant \phi^k$, где $\phi = \left(1 + \sqrt{5}\right)/2$.

Доказательство. Обозначим через s_k минимально возможный размер узла степени k в произвольной фибоначчиевой пирамиде. Случаи $s_0 = 1$ и $s_1 = 2$ тривиальны. Число s_k не превышает величины $size\,(x)$ и, поскольку добавление дочерних узлов к узлу не может уменьшить его размер, значение s_k монотонно возрастает с возрастанием k. Рассмотрим некоторый узел z в произвольной фибоначчиевой пирамиде, такой что $degree\,[z] = k$ и $size\,(z) = s_k$. Так как $s_k \leqslant size\,(x)$, мы вычисляем нижнюю границу $size\,(x)$ путем вычисления нижней границы s_k. Как и в лемме 20.1, обозначим через $y_1, y_2, \ldots, y_k$ дочерние узлы z в порядке их связывания с z. При вычислении нижней границы s_k учтем по единице для самого z и для его первого дочернего узла y_1 (для которого $size\,(y_1) \geqslant 1$). Тогда

$$size\,(x) \geqslant s_k \geqslant 2 + \sum_{i=2}^{k} s_{degree[y_i]} \geqslant 2 + \sum_{i=2}^{k} s_{i-2},$$

где последний переход следует из леммы 20.1 (откуда $degree\,[y_i] \geqslant i - 2$) и монотонности s_k (откуда $s_{degree[y_i]} \geqslant s_{i-2}$).

Покажем теперь по индукции по k, что $s_k \geqslant F_{k+2}$ для всех неотрицательных целых k. База индукции при $k = 0$ и $k = 1$ доказывается тривиально. Далее мы предполагаем, что $k \geqslant 2$ и что $s_i \geqslant F_{i+2}$ при $i = 0, 1, \ldots, k - 1$. Тогда

$$s_k \geqslant 2 + \sum_{i=2}^{k} s_{i-2} \geqslant 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}$$

(последний переход сделан на основании леммы 20.2). Таким образом, мы показали, что $size\,(x) \geqslant s_k \geqslant F_{k+2} \geqslant \phi^k$.

■

Следствие 20.4. Максимальная степень $D(n)$ произвольного узле в фибоначчиевой пирамиде с n узлами равна $O(\lg n)$.

Доказательство. Пусть x — произвольный узел в фибоначчиевой пирамиде с n узлами, и пусть $k = degree[x]$. Согласно лемме 20.3, $n \geqslant size(x) \geqslant \phi^k$. Логарифмируя по основанию ϕ, получаем $k \leqslant \log_\phi n$ (в действительности, так как k — целое число, $k \leqslant \lfloor \log_\phi n \rfloor$). Таким образом, максимальная степень $D(n)$ произвольного узла равна $O(\lg n)$. ■

Упражнения

20.4-1. Профессор утверждает, что высота фибоначчиевой пирамиды из n узлов равна $O(\lg n)$. Покажите, что профессор ошибается и что для любого положительного n имеется последовательность операций, которая создает фибоначчиеву пирамиду, состоящую из одного дерева, которое представляет собой линейную цепочку узлов.

20.4-2. Предположим, что мы обобщили правило каскадного вырезания и что узел x вырезается у родительского узла, как только теряет k-й дочерний узел, где k — некоторая константа (в правиле из раздела 20.3 использовано значение $k = 2$). Для каких значений k справедливо соотношение $D(n) = O(\lg n)$?

Задачи

20-1. **Альтернативная реализация удаления**

Профессор предложил следующий вариант процедуры FIB_HEAP_DELETE, утверждая, что он работает быстрее для случая удаления узла, не являющегося минимальным.

```
PROF_DELETE(H, x)
1  if x = min[H]
2     then FIB_HEAP_EXTRACT_MIN(H)
3     else y ← p[x]
4          if y ≠ NIL
5             then CUT(H, x, y)
6                  CASCADING_CUT(H, y)
7          Добавить список дочерних узлов x в список корней H
8          Удалить x из списка корней H
```

а) Утверждение профессора о быстрой работе процедуры основано, в частности, на предположении, что строка 7 может быть выполнена за время $O(1)$. Какое обстоятельство упущено из виду?

б) Оцените верхнюю границу фактического времени работы процедуры PROF_DELETE, когда x не является $min\,[H]$. Ваша оценка должна быть выражена через $degree\,[x]$ и количество c вызовов процедуры CASCADING_CUT.

в) Предположим, что мы вызываем PROF_DELETE(H, x), и пусть H' — полученная в результате фибоначчиева пирамида. Считая, что x не является корнем, оцените потенциал H', выразив его через $degree\,[x]$, c, $t\,(H)$ и $m\,(H)$.

г) Выведите из полученных результатов оценку амортизированного времени работы процедуры PROF_DELETE и покажите, что оно асимптотически не лучше амортизированного времени работы процедуры FIB_HEAP_DELETE, даже если $x \neq min\,[H]$.

20-2. **Дополнительные операции над фибоначчиевыми пирамидами**

Мы хотим реализовать поддержку двух операций над фибоначчиевыми пирамидами, при этом не изменяя амортизированное время работы прочих операций над фибоначчиевыми пирамидами.

а) Операция FIB_HEAP_CHANGE_KEY(H, x, k) изменяет ключ узла x, присваивая ему значение k. Приведите эффективную реализацию процедуры FIB_HEAP_CHANGE_KEY и проанализируйте амортизированное время работы вашей реализации для случаев, когда k больше, меньше или равно $key\,[x]$.

б) Разработайте эффективную реализацию процедуры FIB_HEAP_PRUNE(H, r), которая удаляет $\min\,(r, n\,[H])$ узлов из H. То, какие именно узлы удаляются, не должно приниматься во внимание. Проанализируйте амортизированное время работы вашей реализации. (*Указание*: вам может потребоваться изменение структуры данных и потенциальной функции.)

Заключительные замечания

Фибоначчиевы пирамиды были введены Фредманом (Fredman) и Таржаном (Tarjan) [98]. В их статье описано также приложение фибоначчиевых пирамид к задачам о кратчайших путях из одной вершины, паросочетаниях с весами и о минимальном остовном дереве.

Впоследствии Дрисколл (Driscoll), Габов (Gabow), Шрейрман (Sghrairman) и Таржан [81] разработали так называемые “ослабленные пирамиды” (relaxed heaps) в качестве альтернативы фибоначчиевым пирамидам. Имеется два варианта ослабленных пирамид. Один дает то же амортизированное время работы,

что и фибоначчиевы пирамиды, второй же позволяет выполнять операцию DECREASE_KEY в наихудшем случае за (не амортизированное) время $O(1)$, а процедуры EXTRACT_MIN и DELETE в наихудшем случае за время $O(\lg n)$. Ослабленные пирамиды имеют также преимущество над фибоначчиевыми пирамидами в параллельных алгоритмах.

Обратитесь к материалу главы 6, где рассмотрены другие структуры данных, которые поддерживают быстрое выполнение операции DECREASE_KEY, когда последовательность значений, возвращаемых вызовами процедуры EXTRACT_MIN, монотонно растет со временем, а данные представляют собой целые числа в определенном диапазоне.

ГЛАВА 21

Структуры данных для непересекающихся множеств

В некоторых приложениях выполняется группировка n различных элементов в набор непересекающихся множеств. Две важные операции, которые должны выполняться с таким набором, — поиск множества, содержащего заданный элемент, и объединение двух множеств. В данной главе мы познакомимся со структурой данных, которая поддерживает эти операции.

В разделе 21.1 описаны операции, поддерживаемые указанной структурой данных, и представлено простое приложение. В разделе 21.2 мы рассмотрим использование простого связанного списка для представления связанных множеств. Более эффективное представление при помощи деревьев приведено в разделе 21.3. Время работы с использованием деревьев линейно для всех практических применений, хотя теоретически оно сверхлинейно. В разделе 21.4 определяется и обсуждается очень быстро растущая функция и ее очень медленно растущая инверсия, которая и проявляется во времени работы операций над представлением непересекающихся множеств с использованием деревьев. Там же при помощи амортизационного анализа доказывается верхняя сверхлинейная граница времени работы.

21.1 Операции над непересекающимися множествами

Структура данных для непересекающихся множеств (disjoint-set data structure) поддерживает набор $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ непересекающихся динамических множеств. Каждое множество идентифицируется ***представителем*** (representative), который представляет собой некоторый член множества. В ряде приложений не имеет значения, какой именно элемент множества используется в качестве представителя; главное, чтобы при запросе представителя множества дважды, без внесения изменений в множество между запросами, возвращался один и тот же элемент. В других приложениях может действовать предопределенное правило выбора представителя, например, наименьшего члена множества (само собой разумеется, в предположении о возможности упорядочения элементов множества).

Как и в других изученных нами реализациях динамических множеств, каждый элемент множества представляет некоторый объект. Обозначим объект через x. Мы хотим обеспечить поддержку следующих операций.

MAKE_SET(x) создает новое множество, состоящее из одного члена (который, соответственно, является его представителем) x. Поскольку множества непересекающиеся, требуется, чтобы x не входил ни в какое иное множество.

UNION(x, y) объединяет динамические множества, которые содержат x и y (обозначим их через S_x и S_y), в новое множество. Предполагается, что до выполнения операции указанные множества не пересекались. Представителем образованного в результате множества является произвольный элемент $S_x \cup \cup S_y$, хотя многие реализации операции UNION выбирают новым представителем представителя множества S_x или S_y. Поскольку нам необходимо, чтобы все множества были непересекающимися, операция UNION должна уничтожать множества S_x и S_y, удаляя их из коллекции $\mathcal{S}$.

FIND_SET(x) возвращает указатель на представителя (единственного) множества, в котором содержится элемент x.

В этой главе мы проанализируем зависимость времени работы структуры данных для непересекающихся множеств от двух параметров: количества операций MAKE_SET n и общего количества операций MAKE_SET, UNION и FIND_SET m. Поскольку множества непересекающиеся, каждая операция UNION уменьшает количество множеств на 1. Следовательно, после $n - 1$ операций UNION останется только одно множество, так что количество операций UNION не может превышать $n - 1$. Заметим также, что поскольку в общее количество операций m включены операции MAKE_SET, то $m \geqslant n$. Предполагается также, что n операций MAKE_SET являются первыми n выполненными операциями.

Приложение структур данных для непересекающихся множеств

Одно из многих применений структур данных для непересекающихся множеств — в задаче об определении связных компонентов неориентированного графа (см. раздел Б.4). Так, на рис. 21.1 показан граф, состоящий из четырех связных компонентов — $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$ и $\{j\}$.

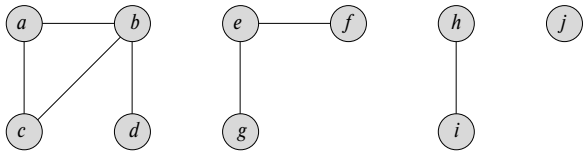


Рис. 21.1. Граф с четырьмя связными компонентами

Процедура CONNECTED_COMPONENTS, приведенная ниже, использует операции над непересекающимися множествами для вычисления связных компонентов графа. После того как процедура CONNECTED_COMPONENTS разобьет множество вершин графа на непересекающиеся множества, процедура SAME_COMPONENT может определить, принадлежат ли две данные вершины одному и тому же связному компоненту[1]. (Множество вершин графа G обозначим как $V[G]$, а множество ребер — как $E[G]$.)

```
CONNECTED_COMPONENTS(G)
1  for Каждая вершина v ∈ V[G]
2       do MAKE_SET(v)
3  for Каждое ребро (u, v) ∈ E[G]
4       do if FIND_SET(u) ≠ FIND_SET(v)
5             then UNION(u, v)

SAME_COMPONENT(u, v)
1  if FIND_SET(u) = FIND_SET(v)
2     then return TRUE
3     else  return FALSE
```

Процедура CONNECTED_COMPONENTS сначала помещает каждую вершину v в ее собственное множество. Затем для каждого ребра (u, v) выполняется объединение множеств, содержащих u и v. В соответствии с упражнением 21.1-2, после

[1] Если ребра графа являются "статическими", т.е. не изменяются с течением времени, то вычислить связные компоненты можно быстрее при помощи поиска в глубину (см. упражнение 22.3-11). Однако иногда ребра добавляются "динамически" и нам надо поддерживать связные компоненты при добавлении нового ребра. В этой ситуации приведенная здесь реализация оказывается эффективнее, чем повторное выполнение поиска вглубь для каждого нового ребра.

Таблица 21.1. Набор непересекающихся множеств в процессе вычислений

Обрабатываемое ребро	Набор непересекающихся множеств									
Начальные множества	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

обработки всех ребер две вершины будут находиться в одном связном компоненте тогда и только тогда, когда соответствующие объекты находятся в одном множестве. Итак, процедура CONNECTED_COMPONENTS вычисляет множества так, что процедура SAME_COMPONENT может определить, находятся ли две вершины в одном и том же связном компоненте. В табл. 21.1 показан процесс вычисления непересекающихся множеств процедурой CONNECTED_COMPONENTS.

В реальной реализации описанного алгоритма представления графа и структуры непересекающихся множеств требуют наличия взаимных ссылок, т.е. объект, представляющий вершину, должен содержать указатель на соответствующий объект в непересекающемся множестве и наоборот. Эти детали программной реализации зависят от используемого для реализации языка программирования и здесь не рассматриваются.

Упражнения

21.1-1. Предположим, что процедура CONNECTED_COMPONENTS вызывается для неориентированного графа $G = (V, E)$, где $V = \{a, b, c, d, e, f, g, h, i, j, k\}$, а ребра из E обрабатываются в следующем порядке: (d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e), (i, d). Перечислите вершины в каждом связном компоненте после выполнения каждой итерации в строках 3–5.

21.1-2. Покажите, что после того, как все ребра будут обработаны процедурой CONNECTED_COMPONENTS, две вершины находятся в одном связном компоненте тогда и только тогда, когда они находятся в одном и том же множестве.

21.1-3. Сколько раз вызывается процедура FIND_SET в процессе выполнения процедуры CONNECTED_COMPONENTS над неориентированным графом $G =$

$= (V, E)$ с k связными компонентами? Сколько раз вызывается процедура UNION? Выразите ваш ответ через $|V|$, $|E|$ и k.

21.2 Представление непересекающихся множеств с помощью связанных списков

Простейший способ реализации структуры данных для непересекающихся множеств — с помощью связанных списков. Первый объект в каждом связанном списке является его представителем. Каждый объект в связанном списке содержит член множества, указатель на объект, содержащий следующий член множества, и указатель на представителя. Каждый список поддерживает указатель $head$ на представителя и указатель $tail$ — на последний объект в списке. На рис. 21.2*а* показаны два множества. Внутри каждого связанного списка объекты могут располагаться в произвольном порядке (однако мы считаем, что первый объект в каждом списке является его представителем).

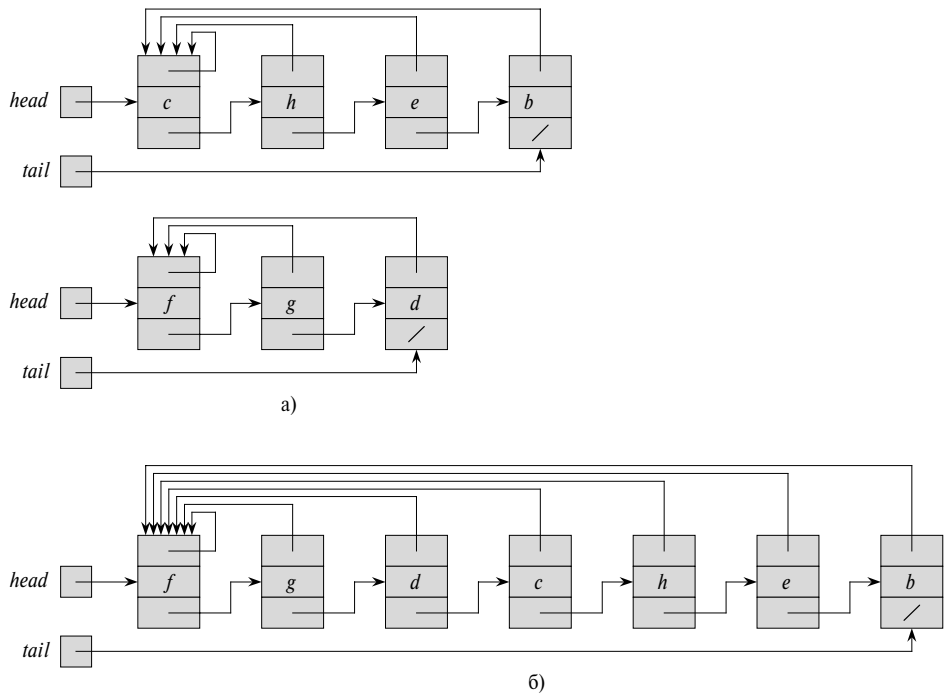


Рис. 21.2. Представление двух множеств с помощью связанного списка и их объединение

При использовании такого представления процедуры MAKE_SET и FIND_SET легко реализуются и время их работы равно $O(1)$. Процедура MAKE_SET(x) создает новый связанный список с единственным объектом x, а процедура FIND_SET просто возвращает указатель на представителя множества.

Простая реализация объединения

Простейшая реализация операции UNION при использовании связанных списков требует гораздо больше времени, чем процедуры MAKE_SET или FIND_SET. Как показано на рис. 21.2*б*, мы выполняем процедуру UNION(x, y), добавляя список с элементом x в конец списка, содержащего y. Указатель $tail$ списка с элементом y используется для того, чтобы быстро определить, куда следует добавить список, содержащий x. Представителем нового множества является элемент, который был представителем множества, содержащего y. К сожалению, при этом мы должны обновить указатели на представителя у каждого объекта, который первоначально находился в списке, содержащем x, что требует времени, линейно зависящего от длины списка, содержащего x.

В действительности, не трудно привести последовательность из m операций над n объектами, которая требует $\Theta\left(n^2\right)$ времени. Предположим, что у нас есть объекты $x_1, x_2 \ldots, x_n$. Мы выполняем последовательность из n операций MAKE_SET, за которой следует последовательность из $n-1$ операций UNION, показанных в табл. 21.2, так что $m = 2n-1$. На выполнение n операций MAKE_SET мы тратим время $\Theta(n)$. Поскольку i-я операция UNION обновляет i объектов, общее количество объектов, обновленных всеми $n-1$ операциями UNION, равно

$$\sum_{i=1}^{n-1} i = \Theta\left(n^2\right).$$

Таблица 21.2. Последовательность операций при использовании связанных списков

Операция	Количество обновленных объектов
MAKE_SET(x_1)	1
MAKE_SET(x_2)	1
⋮	⋮
MAKE_SET(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
⋮	⋮
UNION(x_{n-1}, x_n)	$n-1$

Общее количество операций равно $2n - 1$, так что каждая операция в среднем требует для выполнения времени $\Theta(n)$. Таким образом, амортизированное время выполнения операции составляет $\Theta(n)$.

Весовая эвристика

В худшем случае представленная реализация процедуры UNION требует в среднем $\Theta(n)$ времени на один вызов, поскольку может оказаться, что мы присоединяем длинный список к короткому и должны при этом обновить поля указателей на представителя всех членов длинного списка. Предположим теперь, что каждый список включает также поле длины списка (которое легко поддерживается) и что мы всегда добавляем меньший список к большему (при одинаковых длинах порядок добавления безразличен). При такой простейшей ***весовой эвристике*** (weighted-union heuristic) одна операция UNION может потребовать $\Omega(n)$ времени, если оба множества имеют по $\Omega(n)$ членов. Однако, как показывает следующая теорема, последовательность из m операций MAKE_SET, UNION и FIND_SET, n из которых составляют операции MAKE_SET, требует для выполнения $O(m + n \lg n)$ времени.

Теорема 21.1. При использовании связанных списков для представления непересекающихся множеств и применении весовой эвристики, последовательность из m операций MAKE_SET, UNION и FIND_SET, n из которых составляют операции MAKE_SET, требует для выполнения $O(m + n \lg n)$ времени.

Доказательство. Начнем с вычисления верхней границы количества обновлений указателя на представителя для каждого объекта множества из n элементов. Рассмотрим фиксированный объект n. Мы знаем, что всякий раз, когда происходит обновление указателя на представителя в объекте x, он должен находиться в меньшем из объединяемых множеств. Следовательно, когда происходит первое обновление указателя на представителя в объекте x, образованное в результате множество содержит не менее 2 элементов. При следующем обновлении указателя на представителя в объекте x полученное после объединения множество содержит не менее 4 членов. Продолжая рассуждения, приходим к выводу о том, что для произвольного $k \leqslant n$, после того как указатель на представителя в объекте x обновлен $\lceil \lg k \rceil$ раз, полученное в результате множество должно иметь не менее k элементов. Поскольку максимальное множество может иметь только n элементов, во всех операциях UNION указатель на представителя у каждого объекта может быть обновлен не более $\lceil \lg n \rceil$ раз. Мы должны также учесть обновления указателей $head$ и $tail$ и длины списка, для выполнения которых при каждой операции UNION требуется $\Theta(1)$ времени. Таким образом, общее время, необходимое для обновления n объектов, составляет $O(n \lg n)$.

Отсюда легко выводится время, необходимое для всей последовательности из m операций. Каждая операция MAKE_SET и FIND_SET занимает $O(1)$ времени, а всего их — $O(m)$. Таким образом, полное время выполнения всей последовательности операций составляет $O(m + n \lg n)$. ■

Упражнения

21.2-1. Напишите псевдокод процедур MAKE_SET, FIND_SET и UNION с использованием связанных списков и весовой эвристики. Считаем, что каждый объект x имеет атрибут $rep[x]$, указывающий на представителя множества, содержащего x, и что каждое множество S имеет атрибуты $head[S]$, $tail[S]$ и $size[S]$ (который равен длине списка).

21.2-2. Покажите, какая образуется структура данных и какие ответы дают процедуры FIND_SET в следующей программе. Используется представление при помощи связанных списков и весовая эвристика.

```
1  for i ← 1 to 16
2      do MAKE_SET(x_i)
3  for i ← 1 to 15 by 2
4      do UNION(x_i, x_{i+1})
5  for i ← 1 to 13 by 4
6      do UNION(x_i, x_{i+2})
7  UNION(x_1, x_5)
8  UNION(x_11, x_13)
9  UNION(x_1, x_10)
10 FIND_SET(x_2)
11 FIND_SET(x_9)
```

Считаем, что если множества, содержащие x_i и x_j, имеют одинаковый размер, то операция UNION(x_i, x_j) присоединяет список x_j к списку x_i.

21.2-3. Адаптируйте доказательство теоремы 21.1 для получения границ амортизированного времени выполнения процедур MAKE_SET и FIND_SET, равного $O(1)$, и амортизированного времени выполнения процедуры UNION, равного $O(\lg n)$, при использовании связанных списков и весовой эвристики.

21.2-4. Найдите точную асимптотическую границу времени работы последовательности операций из табл. 21.2 в предположении использования связанных списков и весовой эвристики.

21.2-5. Предложите изменение процедуры UNION для связанных списков, которое избавляет от необходимости поддерживать указатель $tail$ на последний объект каждого списка. Ваше предложение не должно изменить

асимптотическое время работы процедуры UNION независимо от того, используется ли в ней весовая эвристика. (*Указание*: вместо присоединения одного списка к другому, воспользуйтесь вставкой в начало списка.)

21.3 Лес непересекающихся множеств

В более быстрой реализации непересекающихся множеств мы представляем множества в виде корневых деревьев, где каждый узел содержит один член множества, а каждое дерево представляет одно множество. В ***лесу непересекающихся множеств*** (disjoint-set forest) (см. рис. 21.3*а*, где вы видите два дерева, представляющие множества, ранее представленные на рис. 21.2) каждый член указывает только на родительский узел. Корень каждого дерева содержит представителя и является родительским узлом для самого себя. Как мы увидим далее, хотя наивное программирование и не приводит к более эффективной работе по сравнению со связанными списками, введение двух эвристик — "объединение по рангу" и "сжатие пути" — позволяет достичь асимптотически более быстрого выполнения операций над непересекающимися множествами.

Три операции над непересекающимися множествами выполняются следующим образом. Операция MAKE_SET просто создает дерево с одним узлом. Поиск в операции FIND_SET выполняется простым передвижением до корня дерева по указателям на родительские узлы. Посещенные узлы на этом пути составляют ***путь поиска*** (find path). Операция UNION, показанная на рис. 21.3*б*, состоит в том, что корень одного дерева указывает на корень другого.

Эвристики для повышения эффективности

Пока что у нас нет никаких особых преимуществ перед реализацией с использованием связанных списков: последовательность из $n-1$ операций UNION может создать дерево, которое представляет собой линейную цепочку из n узлов. Однако

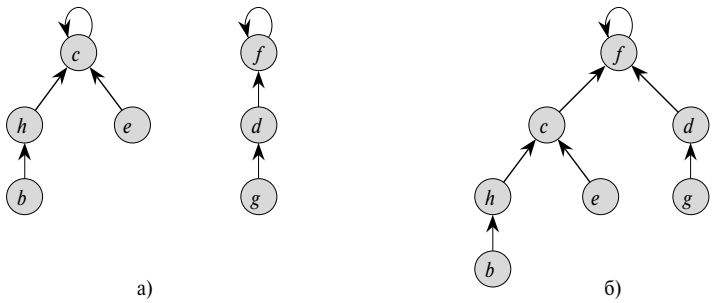


Рис. 21.3. Лес непересекающихся множеств

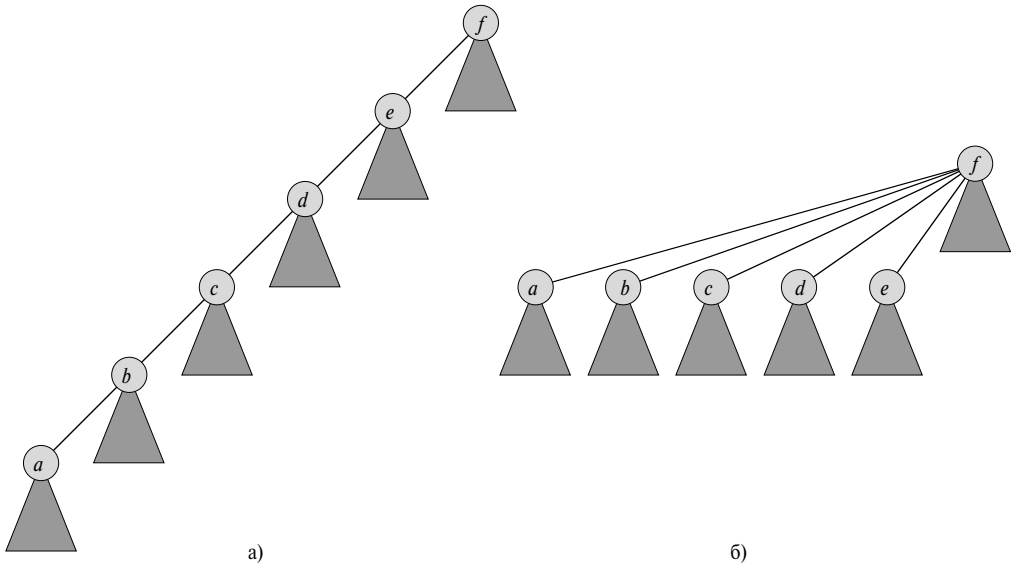


Рис. 21.4. Сжатие пути в процессе выполнения операции FIND_SET

мы можем воспользоваться двумя эвристиками и достичь практически линейного времени работы от общего количества операций m.

Первая эвристика, ***объединение по рангу*** (union by rank), аналогична весовой эвристике при использовании связанных списков. Ее идея заключается в том, чтобы корень дерева с меньшим количеством узлов указывал на корень дерева с большим количеством узлов. Вместо явной поддержки размера поддерева для каждого узла, можно воспользоваться подходом, который упростит анализ — использовать ***ранг*** (rank) каждого корня, который представляет собой верхнюю границу высоты узла. При объединении по рангу корень с меньшим рангом должен указывать на корень с бóльшим рангом.

Вторая эвристика, ***сжатие пути*** (path compression), также достаточно проста и эффективна. Как показано на рис. 21.4, она используется в процессе выполнения операции FIND_SET и делает каждый узел указывающим непосредственно на корень. Сжатие пути не изменяет ранги узлов.

Псевдокоды

Для реализации леса непересекающихся множеств с применением эвристики объединения по рангу, мы должны отслеживать ранг узлов. Для каждого узла x нами поддерживается целое значение $rank\,[x]$, которое представляет собой верхнюю границу высоты x (количество ребер на самом длинном пути от x к листу, являющемуся его потомком). При создании множества из одного элемента

процедурой MAKE_SET начальный ранг узла в соответствующем дереве устанавливается равным 0. Операции FIND_SET оставляют ранги неизменными. При применении процедуры UNION для объединения двух деревьев имеются две ситуации, зависящие от того, имеют объединяемые деревья одинаковый ранг, или нет. Если ранги деревьев разные, мы делаем корень с бóльшим рангом родительским узлом по отношению к корню с меньшим рангом, но сами ранги при этом остаются неизменными. Если же корни имеют одинаковые ранги, то мы произвольным образом выбираем один из корней в качестве родительского и увеличиваем его ранг.

В приведенных псевдокодах родительский по отношению к x узел обозначается как $p[x]$. Вспомогательная процедура LINK, использующаяся в процедуре UNION, получает в качестве входных параметров указатели на два корня.

```
MAKE_SET(x)
1  p[x] ← x
2  rank[x] ← 0

UNION(x, y)
1  LINK(FIND_SET(x), FIND_SET(y))

LINK(x, y)
1  if rank[x] > rank[y]
2      then p[y] ← x
3      else  p[x] ← y
4            if rank[x] = rank[y]
5               then rank[y] ← rank[y] + 1

FIND_SET(x)
1  if x ≠ p[x]
2      then p[x] ← FIND_SET(p[x])
3  return p[x]
```

Процедура FIND_SET является двухпроходной: при первом проходе она ищет путь к корню дерева, а при втором проходе в обратном направлении по найденному пути происходит обновление узлов, которые теперь указывают непосредственно на корень дерева. Каждый вызов FIND_SET возвращает в строке 3 $p[x]$. Если x — корень, то строка 2 не выполняется и возвращается значение $p[x] = x$, и на этом рекурсия завершается. В противном случае выполнение строки 2 приводит к рекурсивному вызову с параметром $p[x]$, который возвращает указатель на корень. В той же строке 2 происходит обновление узла x, после которого он указывает непосредственно на найденный корень, и этот указатель возвращается вызывающей процедуре в строке 3.

Влияние эвристик на время работы

Будучи примененными раздельно, объединение по рангу и сжатие пути приводят к повышению эффективности операций над лесом непересекающихся множеств. Еще больший выигрыш дает совместное применение этих эвристик. Объединение по рангу само по себе дает время работы $O(m \lg n)$ (см. упражнение 21.4-4), причем эта оценка не может быть улучшена (см. упражнение 21.3-3). Хотя мы не будем доказывать это здесь, если имеется n операций MAKE_SET (а следовательно, не более $n-1$ операций UNION) и f операций FIND_SET, то сжатие пути само по себе приводит ко времени работы в наихудшем случае $O\left(n + f \cdot \left(\log_{2+f/n} n\right)\right)$.

При совместном использовании обеих эвристик время работы в наихудшем случае составляет $O(m\,\alpha(n))$, где $\alpha(n)$ — *очень* медленно растущая функция, которую мы определим в разделе 21.4. Для любого мыслимого приложения с использованием непересекающихся множеств $\alpha(n) \leqslant 4$, так что можно рассматривать время работы во всех практических ситуациях как линейно зависящее от m. В разделе 21.4 мы докажем эту верхнюю границу.

Упражнения

21.3-1. Выполните упражнение 21.2-2 для леса непересекающихся множеств с использованием объединения по рангу и сжатия пути.

21.3-2. Разработайте нерекурсивную версию процедуры FIND_SET со сжатием пути.

21.3-3. Приведите последовательность из m операций MAKE_SET, UNION и FIND_SET, n из которых — операции MAKE_SET, время выполнения которой составляет $\Omega(m \lg n)$ при использовании только объединения по рангу.

⋆ 21.3-4. Покажите, что время выполнения произвольной последовательности m операций MAKE_SET, FIND_SET и LINK, в которой все операции LINK выполняются до первой операции FIND_SET, равно $O(m)$, если используются как объединение по рангу, так и сжатие пути. Что будет, если воспользоваться только сжатием пути?

⋆ 21.4 Анализ объединения по рангу со сжатием пути

Как отмечалось в разделе 21.3, время работы m операций над непересекающимися множествами из n элементов при использовании объединения по рангу и сжатия пути равно $O(m\,\alpha(n))$. В данном разделе мы рассмотрим функцию

$\alpha(n)$ и выясним, насколько медленно она растет. Затем мы докажем приведенное выше время работы с использованием метода потенциала из амортизационного анализа.

Очень быстро и очень медленно растущая функция

Определим функцию $A_k(j)$ для целых $k \geqslant 0$ и $j \geqslant 1$ как

$$A_k(j) = \begin{cases} j+1 & \text{при } k = 0, \\ A_{k-1}^{(j+1)}(j) & \text{при } k \geqslant 1, \end{cases}$$

где выражение $A_{k-1}^{(j+1)}(j)$ использует функционально-итеративные обозначения из раздела 3.2. В частности, $A_{k-1}^{(0)}(j) = j$ и $A_{k-1}^{(i)}(j) = A_{k-1}\left(A_{k-1}^{(i-1)}(j)\right)$ для $i \geqslant 1$. Параметр k будем называть ***уровнем*** (level) функции A.

Функция $A_k(j)$ — строго возрастающая по j и k. Для того чтобы увидеть, насколько быстро растет данная функция, начнем с записи функций $A_1(j)$ и $A_2(j)$ в явном виде.

Лемма 21.2. Для произвольного целого $j \geqslant 1$ $A_1(j) = 2j + 1$.

Доказательство. Воспользуемся индукцией по i, чтобы показать, что $A_0^{(i)}(j) = j + i$. Начнем с базы индукции: $A_0^{(0)}(j) = j = j + 0$. Далее предположим, что $A_0^{(i-1)}(j) = j+(i-1)$. Тогда $A_0^{(i)}(j) = A_0\left(A_0^{(i-1)}(j)\right) = (j + (i-1))+1 = j+i$. И наконец, $A_1(j) = A_0^{(j+1)}(j) = j + (j+1) = 2j + 1$. ■

Лемма 21.3. Для произвольного целого $j \geqslant 1$ $A_2(j) = 2^{j+1}(j+1) - 1$.

Доказательство. Воспользуемся индукцией по i, чтобы показать, что $A_1^{(i)}(j) = 2^i(j+1) - 1$. Начнем с базы индукции: $A_1^{(0)}(j) = j = 2^0(j+1) - 1$. Далее предположим, что $A_1^{(i-1)}(j) = 2^{i-1}(j+1) - 1$. Тогда

$$\begin{aligned} A_1^{(i)}(j) &= A_1\left(A_1^{(i-1)}(j)\right) = \\ &= A_1\left(2^{i-1}(j+1) - 1\right) = \\ &= 2 \cdot \left(2^{i-1}(j+1) - 1\right) + 1 = \\ &= 2^i(j+1) - 2 + 1 = 2^i(j+1) - 1. \end{aligned}$$

И наконец, $A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j+1) - 1$. ■

Теперь посмотрим, насколько быстро растет функция $A_k(j)$, просто вычисляя $A_k(1)$ для уровней k от 0 до 4. Из определения $A_0(j)$ и рассмотренных выше лемм мы имеем $A_0(1) = 1 + 1 = 2$, $A_1(1) = 2 \cdot 1 + 1 = 3$ и $A_2(1) = 2^{1+1} \cdot \cdot (1+1) - 1 = 7$. Продолжая вычисления, получаем:

$$A_3(1) = A_2^{(2)}(1) = A_2(A_2(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2^{11} - 1 = 2047$$

и

$$A_4(1) = A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047) = A_2^{(2048)}(2047) \gg$$
$$\gg A_2(2047) = 2^{2048} \cdot 2048 - 1 > 2^{2048} = \left(2^4\right)^{512} = 16^{512} \gg 10^{80},$$

что представляет собой оценочное количество атомов в наблюдаемой вселенной.

Определим обратную к $A_k(j)$ функцию для $n \geqslant 0$ следующим образом:

$$\alpha(n) = \min\{k : A_k(1) \geqslant n\}.$$

Другими словами, $\alpha(n)$ — наименьший уровень k, для которого $A_k(1)$ не меньше n. Исходя из найденных выше значений функции,

$$\alpha(n) = \begin{cases} 0 & \text{при } 0 \leqslant n \leqslant 2, \\ 1 & \text{при } n = 3, \\ 2 & \text{при } 4 \leqslant n \leqslant 7, \\ 3 & \text{при } 8 \leqslant n \leqslant 2047, \\ 4 & \text{при } 2048 \leqslant n \leqslant A_4(1). \end{cases}$$

Только для невероятно больших значений n (бóльших $A_4(1)$) значение функции $\alpha(n)$ становится больше 4, так что для практического применения $\alpha(n) \leqslant 4$.

Свойства рангов

В оставшейся части раздела мы докажем, что при использовании объединения по рангам и сжатия пути граница времени работы операций над непересекающимися множествами равна $O(m\,\alpha(n))$. Для этого нам потребуются некоторые простые свойства рангов.

Лемма 21.4. Для всех узлов x выполняется соотношение $rank[x] \leqslant rank[p[x]]$, причем равенство достигается только при $x = p[x]$. Значение $rank[x]$ изначально равно 0 и, пока $x \neq p[x]$, увеличивается со временем. По достижении равенства $x = p[x]$ значение $rank[x]$ больше не изменяется. Значение $rank[p[x]]$ монотонно возрастает со временем.

Доказательство. Доказательство выполняется простой индукцией по количеству операций, с использованием реализаций процедур MAKE_SET, UNION и FIND_SET в разделе 21.3. Подробности доказательства оставлены в качестве упражнения 21.4-1. ■

Следствие 21.5. При перемещении вдоль пути от произвольного узла к корню ранг строго возрастает. ■

Лемма 21.6. Ранг любого узла не превышает $n - 1$.

Доказательство. Начальный ранг каждого узла — 0, и он возрастает только при выполнении операции LINK. Поскольку выполняется не более $n - 1$ операции UNION, операций LINK также выполняется не более чем $n - 1$. Поскольку каждая операция LINK либо оставляет все ранги неизменными, либо увеличивает один ранг на 1, ни один ранг не может превышать $n - 1$. ■

Лемма 21.6 дает слабую оценку границы рангов. В действительности ранг любого узла не превосходит $\lfloor \lg n \rfloor$ (см. упражнение 21.4-2). Однако для наших целей достаточно границы, определяемой леммой 21.6.

Доказательство границы времени работы

Для доказательства границы $O(m\,\alpha(n))$ мы воспользуемся методом потенциала из амортизационного анализа (см. раздел 17.3). При выполнении амортизационного анализа удобно считать, что мы выполняем операцию LINK, а не UNION. Т.е., поскольку параметрами процедуры LINK являются указатели на два корня, мы считаем, что соответствующие операции FIND_SET выполняются отдельно. Приведенная ниже лемма показывает, что даже если учесть дополнительные операции FIND_SET, инициированные вызовами UNION, асимптотическое время работы останется неизменным.

Лемма 21.7. Предположим, что мы преобразуем последовательность S' из m' операций MAKE_SET, UNION и FIND_SET в последовательность из m операций MAKE_SET, LINK и FIND_SET путем преобразования каждой операции UNION в две операции FIND_SET, за которыми следует операция LINK. Тогда, если последовательность S выполняется за время $O(m\alpha(n))$, то последовательность S' выполняется за время $O(m'\alpha(n))$.

Доказательство. Поскольку каждая операция UNION в последовательности S' преобразуется в три операции в S, выполняется соотношение $m' \leqslant m \leqslant 3m'$. Так как $m = O(m')$, из границы времени работы $O(m\alpha(n))$ преобразованной последовательности S следует граница $O(m'\alpha(n))$ времени работы исходной последовательности S'. ■

В оставшейся части этого раздела мы полагаем, что исходная последовательность из m' операций MAKE_SET, UNION и FIND_SET преобразована в последовательность из m операций MAKE_SET, LINK и FIND_SET. Теперь мы докажем, что время выполнения полученной последовательности равно $O(m\alpha(n))$ и обратимся к лемме 21.7 для доказательства времени работы $O(m'\alpha(n))$ исходной последовательности из m' операций.

Потенциальная функция

Используемая нами потенциальная функция присваивает потенциал $\phi_q(x)$ каждому узлу x в лесу непересекающихся множеств после выполнения q операций. Для получения потенциала всего леса мы суммируем потенциалы его узлов: $\Phi_q = \sum_x \phi_q(x)$, где Φ_q — потенциал всего леса после выполнения q операций. До выполнения первой операции лес пуст, и мы полагаем, что $\Phi_0 = 0$. Потенциал Φ_q никогда не может стать отрицательным.

Значение $\phi_q(x)$ зависит от того, является ли x корнем дерева после q-й операции. Если это так или если $rank[x] = 0$, $\phi_q(x) = \alpha(n) \cdot rank[x]$.

Теперь предположим, что после q-й операции x не является корнем и что $rank[x] \geqslant 1$. Нам надо определить две вспомогательные функции от x перед тем, как мы определим $\phi_q(x)$. Сначала мы определим

$$\mathrm{level}(x) = \max\{k : rank[p[x]] \geqslant A_k(rank[x])\},$$

т.е. $\mathrm{level}(x)$ — наибольший уровень k, для которого A_k, примененное к рангу x, не превышает ранг родителя x.

Мы утверждаем, что

$$0 \leqslant \mathrm{level}(x) < \alpha(n). \tag{21.1}$$

Это можно подтвердить следующим образом. Мы имеем

$$rank[p[x]] \geqslant rank[x] + 1 = A_0(rank[x]),$$

где неравенство следует из леммы 21.4, а равенство — по определению $A_0(j)$. Отсюда вытекает, что $\mathrm{level}(x) \geqslant 0$. Тогда

$$A_{\alpha(n)}(rank[x]) \geqslant A_{\alpha(n)}(1) \geqslant n \geqslant rank[p[x]],$$

где первое неравенство следует из строго возрастающего характера функции $A_k(j)$, второе — из определения $\alpha(n)$, а третье — из леммы 21.6. Из полученного соотношения вытекает, что $\mathrm{level}(x) < \alpha(n)$. Заметим, что поскольку $rank[p[x]]$ монотонно увеличивается со временем, то же происходит и с $\mathrm{level}(x)$.

Вторую вспомогательную функцию определим следующим образом:

$$\mathrm{iter}(x) = \max\left\{i : rank[p[x]] \geqslant A^{(i)}_{\mathrm{level}(x)}(rank[x])\right\},$$

т.е. $\mathrm{iter}\,(x)$ — наибольшее число раз итеративного применения функции $A_{\mathrm{level}(x)}$ к исходному рангу x, до того как мы получим значение, превышающее ранг родителя x.

Мы утверждаем, что

$$1 \leqslant \mathrm{iter}\,(x) \leqslant rank\,[x]\,. \tag{21.2}$$

В этом можно убедиться следующим образом. Мы имеем

$$rank\,[p\,[x]] \geqslant A_{\mathrm{level}(x)}\,(rank\,[x]) = A^{(1)}_{\mathrm{level}(x)}\,(rank\,[x])\,,$$

где неравенство следует из определения $\mathrm{level}\,(x)$, а равенство — из определения функциональной итерации. Из приведенного соотношения вытекает, что $\mathrm{iter}\,(x) \geqslant$ $\geqslant 1$, и мы получаем в соответствии с определениями $A_k\,(j)$ и $\mathrm{level}\,(x)$

$$A^{(rank[x]+1)}_{\mathrm{level}(x)}\,(rank\,[x]) = A_{\mathrm{level}(x)+1}\,(rank\,[x]) > rank\,[p\,[x]]\,,$$

откуда $\mathrm{iter}\,(x) \leqslant rank\,[x]$. Заметим, что поскольку $rank\,[p[x]]$ монотонно увеличивается со временем, для того чтобы значение $\mathrm{iter}\,(x)$ уменьшалось, величина $\mathrm{level}\,(x)$ должна увеличиваться. Если же $\mathrm{level}\,(x)$ остается неизменной, то значение $\mathrm{iter}\,(x)$ должно либо увеличиваться, либо вообще не изменяться.

Имея описанные вспомогательные функции, мы можем записать потенциал узла x после q операций:

$$\phi_q\,(x) = \begin{cases} \alpha\,(n) \cdot rank\,[x] & \text{если } x \text{ корень или } rank\,[x] = 0, \\ (\alpha\,(n) - \mathrm{level}\,(x)) \times & \\ \times\, rank\,[x] - \mathrm{iter}\,(x) & \text{если } x \text{ не корень и } rank\,[x] \geqslant 1. \end{cases}$$

В следующих двух леммах представлены полезные свойства потенциала узла.

Лемма 21.8. Для всех q операций и произвольного узла x

$$0 \leqslant \phi_q\,(x) \leqslant \alpha\,(n) \cdot rank\,[x]\,.$$

Доказательство. Если x — корень или $rank\,[x] = 0$, то по определению $\phi_q\,(x) =$ $= \alpha\,(n) \cdot rank\,[x]$. Предположим теперь, что x — не корень и что $rank\,[x] \geqslant$ $\geqslant 1$. Нижнюю границу $\phi_q\,(x)$ можно получить, максимизируя $\mathrm{level}\,(x)$ и $\mathrm{iter}\,(x)$. Согласно (21.1), $\mathrm{level}\,(x) \leqslant \alpha\,(n) - 1$, а в соответствии с (21.2) — $\mathrm{iter}\,(x) \leqslant rank\,[x]$. Таким образом,

$$\phi_q\,(x) \geqslant (\alpha\,(n) - (\alpha\,(n) - 1)) \cdot rank\,[x] - rank\,[x] = rank\,[x] - rank\,[x] = 0.$$

Аналогично получаем верхнюю границу $\phi_q\,(x)$, минимизируя $\mathrm{level}\,(x)$ и $\mathrm{iter}\,(x)$. Согласно (21.1), $\mathrm{level}\,(x) \geqslant 0$, а в соответствии с (21.2) — $\mathrm{iter}\,(x) \geqslant 1$. Таким образом,

$$\phi_q\,(x) \leqslant (\alpha\,(n) - 0) \cdot rank\,[x] - 1 = \alpha\,(n) \cdot rank\,[x] - 1 < \alpha\,(n) \cdot rank\,[x]\,. \quad \blacksquare$$

Изменения потенциала и амортизированная стоимость операций

Теперь мы готовы к рассмотрению вопроса о влиянии операций над непересекающимися множествами на потенциалы узлов. Зная, как изменяется потенциал при той или иной операции, мы можем определить амортизированную стоимость каждой операции.

Лемма 21.9. Пусть x — узел, не являющийся корнем, и предположим, что q-я операция — либо LINK, либо FIND_SET. Тогда после выполнения q-й операции $\phi_q(x) \leqslant \phi_{q-1}(x)$. Более того, если $rank[x] \geqslant 1$ и из-за выполнения q-й операции происходит изменение либо $\mathrm{level}(x)$, либо $\mathrm{iter}(x)$, то $\phi_q(x) \leqslant \phi_{q-1}(x) - 1$. То есть потенциал x не может возрастать, и если он имеет положительное значение, а либо $\mathrm{level}(x)$, либо $\mathrm{iter}(x)$ изменяются, то потенциал x уменьшается как минимум на 1.

Доказательство. Поскольку x не является корнем, q-я операция не изменяет $rank[x]$, и так как n не изменяется после первых n операций MAKE_SET, $\alpha(n)$ остается неизменной величиной. Следовательно, эти компоненты формулы потенциальной функции при выполнении q-й операции не изменяются. Если $rank[x] = 0$, то $\phi_q(x) = \phi_{q-1}(x) = 0$. Теперь предположим, что $rank[x] \geqslant 1$.

Вспомним, что значение функции $\mathrm{level}(x)$ монотонно растет со временем. Если q-я операция оставляет $\mathrm{level}(x)$ неизменной, то функция $\mathrm{iter}(x)$ либо возрастает, либо остается неизменной. Если и $\mathrm{level}(x)$, и $\mathrm{iter}(x)$ не изменяются, то $\phi_q(x) = \phi_{q-1}(x)$. Если же $\mathrm{level}(x)$ не изменяется, а $\mathrm{iter}(x)$ возрастает, то $\mathrm{iter}(x)$ увеличивается как минимум на 1, так что $\phi_q(x) \leqslant \phi_{q-1}(x) - 1$.

И наконец, если q-я операция увеличивает $\mathrm{level}(x)$, то это увеличение составляет как минимум 1, так что значение члена $(\alpha(n) - \mathrm{level}(x)) \cdot rank[x]$ уменьшается как минимум на величину $rank[x]$. Так как значение $\mathrm{level}(x)$ возрастает, значение $\mathrm{iter}(x)$ может уменьшаться, но в соответствии с (21.2), это уменьшение не может превышать $rank[x] - 1$. Таким образом, увеличение потенциала, вызванное изменением функции $\mathrm{iter}(x)$, меньше, чем уменьшение потенциала из-за изменения функции $\mathrm{level}(x)$, так что мы можем заключить, что $\phi_q(x) \leqslant \phi_{q-1}(x) - 1$. ■

Наши последние три леммы показывают, что амортизированная стоимость каждой из операций MAKE_SET, LINK и FIND_SET составляет $O(\alpha(n))$. Вспомним, что, согласно (17.2), амортизированная стоимость каждой операции равна ее фактической стоимости плюс увеличение потенциала, вызванное ее выполнением.

Лемма 21.10. Амортизированная стоимость каждой операции MAKE_SET равна $O(1)$.

Доказательство. Предположим, что q-я операция — MAKE_SET. Эта операция создает узел x с рангом 0, так что $\phi_q(x) = 0$. Никакие иные ранги и потенциалы

не изменяются, так что $\Phi_q = \Phi_{q-1}$. То, что фактическая стоимость операции MAKE_SET равна $O(1)$, завершает доказательство. ■

Лемма 21.11. Амортизированная стоимость каждой операции LINK равна $O(\alpha(n))$.

Доказательство. Предположим, что q-я операция — LINK(x, y). Фактическая стоимость операции LINK равна $O(1)$. Без потери общности предположим, что LINK делает y родительским узлом x.

Для определения изменения потенциала из-за выполнения операции LINK заметим, что множество узлов, чьи потенциалы могут измениться, ограничено узлами x, y и дочерними узлами y непосредственно перед операцией. Мы покажем, что единственным узлом, потенциал которого может увеличиться в результате выполнения операции LINK, — это y, и это увеличение не превышает $\alpha(n)$.

- Согласно лемме 21.9, потенциал любого узла, являющегося дочерним узлом y перед выполнением операции LINK, не может увеличиться в результате выполнения этой операции.
- По определению $\phi_q(x)$ мы видим, что поскольку x перед q-й операцией был корнем, $\phi_{q-1}(x) = \alpha(n) \cdot rank[x]$. Если $rank[x] = 0$, то $\phi_q(x) = \phi_{q-1}(x) = 0$. В противном случае, в соответствии с неравенствами (21.1) и (21.2),
- $\phi_q(x) = (\alpha(n) - \text{level}(x)) \cdot rank[x] - \text{iter}(x) < \alpha(n) \cdot rank[x]$.
  Поскольку $\phi_{q-1}(x) = \alpha(n) \cdot rank[x]$, мы видим, что потенциал x уменьшается.
- Так как y непосредственно перед выполнением операции LINK был корнем, $\phi_{q-1}(y) = \alpha(n) \cdot rank[y]$. Операция LINK оставляет y корнем и либо оставляет ранг y неизменным, либо увеличивает его на 1. Таким образом, либо $\phi_q(y) = \phi_{q-1}(y)$, либо $\phi_q(y) = \phi_{q-1}(y) + \alpha(n)$.

Итак, увеличение потенциала вследствие операции LINK не превышает $\alpha(n)$, и амортизированная стоимость операции LINK равна $O(1) + \alpha(n) = O(\alpha(n))$. ■

Лемма 21.12. Амортизированная стоимость каждой операции FIND_SET равна $O(\alpha(n))$.

Доказательство. Предположим, что q-й операцией является FIND_SET и что путь поиска содержит s узлов. Фактическая стоимость операции FIND_SET составляет $O(s)$. Покажем, что отсутствуют узлы, потенциал которых возрастает вследствие операции FIND_SET, и что как минимум у $\max(0, s - (\alpha(n) + 2))$ узлов на пути поиска потенциал уменьшается по меньшей мере на 1.

Чтобы увидеть отсутствие узлов с возрастающим потенциалом, обратимся к лемме 21.9 для узлов, не являющихся корнем. Если же узел x — корень, то его потенциал равен $\alpha(n) \cdot rank[x]$ и остается неизменным.

Теперь мы покажем, что как минимум у $\max(0, s-(\alpha(n)+2))$ узлов потенциал уменьшается по меньшей мере на 1. Пусть x — узел на пути поиска, такой что $rank[x] > 0$, и за x на пути поиска следует другой узел y, не являющийся корнем, где непосредственно перед выполнением операции FIND_SET $\mathrm{level}(y) = \mathrm{level}(x)$ (узел y не обязательно следует *непосредственно* за x). Этим ограничениям на x удовлетворяют все узлы на пути поиска, кроме $\alpha(n)+2$ узлов. Приведенным условиям не удовлетворяет первый узел на пути поиска (если он имеет нулевой ранг), последний узел пути (т.е. корень), а также $\alpha(n)$ узлов w, которые для данного k, где $k = 0, 1, \ldots, \alpha(n)-1$, являются последними узлами на пути, удовлетворяющими условию $\mathrm{level}(w) = k$.

Зафиксировав такой узел x, покажем, что потенциал x уменьшается как минимум на 1. Пусть $k = \mathrm{level}(x) = \mathrm{level}(y)$. Непосредственно перед сжатием пути в процедуре FIND_SET мы имеем:

$$\begin{aligned} rank[p[x]] &\geqslant A_k^{(\mathrm{iter}(x))}(rank[x]) && \text{по определению } \mathrm{iter}(x), \\ rank[p[y]] &\geqslant A_k(rank[y]) && \text{по определению } \mathrm{level}(x), \\ rank[y] &\geqslant rank[p[x]] && \text{согласно следствию 21.5,} \\ & && \text{поскольку } y \text{ следует за } x. \end{aligned}$$

Объединяя приведенные неравенства и обозначая через i значение $\mathrm{iter}(x)$ перед сжатием пути, получаем:

$$\begin{aligned} rank[p[y]] &\geqslant A_k(rank[y]) \geqslant A_k(rank[p[x]]) \geqslant \\ &\geqslant A_k\left(A_k^{(\mathrm{iter}(x))}(rank[x])\right) = A_k^{(i+1)}(rank[x]), \end{aligned}$$

где второе неравенство следует из того, что $A_k(j)$ — строго возрастающая функция.

Поскольку после сжатия пути и x, и y имеют один и тот же родительский узел, мы знаем, что после сжатия пути $rank[p[x]] = rank[p[y]]$ и что сжатие пути не уменьшает $rank[p[y]]$. Поскольку $rank[x]$ не изменяется, после сжатия пути $rank[p[x]] \geqslant A_k^{(i+1)}(rank[x])$. Таким образом, сжатие пути приводит к тому, что либо увеличивается $\mathrm{iter}(x)$ (как минимум до $i+1$), либо увеличивается $\mathrm{level}(x)$ (как минимум до $rank[x]+1$). В любом случае, в соответствии с леммой 21.9, $\phi_q(x) \leqslant \phi_{q-1}(x) - 1$. Следовательно, потенциал x уменьшается как минимум на 1.

Амортизированная стоимость операции FIND_SET равна фактической стоимости плюс изменение потенциала. Фактическая стоимость равна $O(s)$, и мы показали, что общий потенциал уменьшается как минимум на $\max(0, s-(\alpha(n)+2))$. Амортизированная стоимость, таким образом, не превышает $O(s) - (s-(\alpha(n)+2)) = O(s) - s + O(\alpha(n)) = O(\alpha(n))$, так как мы можем масштабировать потенциал таким образом, чтобы можно было пренебречь константой, скрытой в $O(s)$. ■

Теперь, после того как мы доказали все приведенные выше леммы, мы можем перейти к следующей теореме.

Теорема 21.13. Последовательность из m операций MAKE_SET, UNION и FIND_SET, n из которых — операции MAKE_SET, может быть выполнена над лесом непересекающихся множеств с использованием объединения по рангу и сжатия пути за время $O(m\alpha(n))$ в наихудшем случае.

Доказательство. Непосредственно следует из лемм 21.7, 21.10, 21.11 и 21.12. ■

Упражнения

21.4-1. Докажите лемму 21.4.

21.4-2. Докажите, что ранг произвольного узла не превышает $\lfloor \lg n \rfloor$.

21.4-3. С учетом решения упражнения 21.4-2, сколько битов требуется для хранения поля $rank[x]$ узла x?

21.4-4. Используя решение упражнения 21.4-2, приведите простое доказательство того факта, что операции над лесом непересекающихся множеств с использованием объединения по рангу, но без сжатия пути, выполняются за время $O(m \lg n)$.

21.4-5. Профессор полагает, что поскольку ранг узла строго возрастает вдоль пути к корню, уровни узлов должны монотонно возрастать вдоль этого пути. Другими словами, если $rank[x] > 0$ и $p[x]$ — не корень, то $\text{level}(x) \leqslant \text{level}(p[x])$. Прав ли профессор?

⋆ 21.4-6. Рассмотрим функцию $\alpha'(n) = \min\{k : A_k(1) \geqslant \lg(n+1)\}$. Покажите, что для всех практических применений $\alpha'(n) \leqslant 3$. Покажите также, воспользовавшись решением упражнения 21.4-2, каким образом следует модифицировать аргумент потенциальной функции для доказательства того, что последовательность из m операций MAKE_SET, UNION и FIND_SET, n из которых — операции MAKE_SET, может быть выполнена над лесом непересекающихся множеств с использованием объединения по рангу и сжатия пути за время $O(m\alpha'(n))$ в наихудшем случае.

Задачи

21-1. **Минимум в автономном режиме**

Задача ***поиска минимума в автономном режиме*** (off-line minimum problem) состоит в следующем. Пусть имеется динамическое множество T с элементами из области определения $\{1, 2, \ldots, n\}$, поддерживающее операции INSERT и EXTRACT_MIN. Задана последовательность S из n

вызовов INSERT и m вызовов EXTRACT_MIN, в которой каждый ключ из $\{1, 2, \ldots, n\}$ вставляется ровно один раз. Мы хотим определить, какой ключ возвращается каждым вызовом EXTRACT_MIN, т.е. мы хотим заполнить массив $extracted\,[1..m]$, где $extracted\,[i]$ $(i = 1, 2, \ldots, m)$ представляет собой ключ, возвращаемый i-м вызовом EXTRACT_MIN. Задача "автономна" в том смысле, что перед тем как приступить к вычислениям, ей известна вся последовательность S полностью.

а) В следующей последовательности каждый вызов INSERT представлен вставляемым числом, а каждый вызов EXTRACT_MIN представлен буквой E: 4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5. Заполните массив $extracted$.

Для разработки алгоритма для решения этой задачи разобьем последовательность S на гомогенные подпоследовательности, т.е. представим S в таком виде:

$$I_1, E, I_2, E, I_3, \ldots, I_m, E, I_{m+1}$$

где каждая буква E представляет один вызов EXTRACT_MIN, а I_j — последовательность (возможно, пустую) вызовов INSERT. Для каждой подпоследовательности I_j мы сначала помещаем ключи, вставленные данными операциями, в множество K_j (которое является пустым, если подпоследовательность I_j пуста). Затем сделаем следующее:

```
OFF_LINE_MINIMUM(m, n)
1  for i ← 1 to n
2      do Определяем j, такое что i ∈ K_j
3         if j ≠ m + 1
4            then extracted[j] ← i
5                 Пусть l — наименьшее значение, большее j
                        для которого существует множество K_l
6                 K_l ← K_j ∪ K_l, уничтожаем K_j
7  return extracted
```

б) Покажите корректность массива $extracted$, возвращаемого процедурой OFF_LINE_MINIMUM.

в) Опишите, как эффективно реализовать процедуру OFF_LINE_MINIMUM с использованием структур данных для непересекающихся множеств. Дайте точную оценку времени работы вашей реализации в наихудшем случае.

21-2. **Определение глубины**

В ***задаче по определению глубины*** (depth-determination problem) мы работаем с лесом $\mathcal{T} = \{T_i\}$ корневых деревьев, в котором поддерживаются следующие три операции.

MAKE_TREE(v) создает дерево, состоящее из единственного узла v.

FIND_DEPTH(v) возвращает глубину узла v в его дереве.

GRAFT(r, v) ("прививка") делает узел r, являющийся корнем, дочерним по отношению к узлу v, который находится в дереве, корень которого отличен от r, но при этом не обязательно должен сам быть корнем.

а) Предположим, что мы используем представление дерева такое же, как и в случае непересекающихся множеств: $p[v]$ является родительским узлом v, и если v — корень, то $p[v] = v$. Реализуем процедуру GRAFT(r, v) путем присвоения $p[r] \leftarrow v$, а FIND_DEPTH(v) — как поиск пути к корню с подсчетом всех узлов (кроме v), встречающихся на этом пути. Покажите, что время работы последовательности из m процедур MAKE_TREE, FIND_DEPTH и GRAFT в худшем случае равно $\Theta(m^2)$.

Используя объединение по рангу и сжатие пути, мы можем снизить время работы в наихудшем случае. Воспользуемся лесом непересекающихся множеств $\mathcal{S} = \{S_i\}$, где каждое множество S_i (представляющее собой дерево) соответствует дереву T_i в лесу $\mathcal{T}$. Структура дерева в S_i не обязательно соответствует структуре дерева T_i. Однако хотя реализация S_i не хранит точные отношения родитель-потомок, но тем не менее она позволяет нам определить глубину любого узла в T_i.

Ключевая идея состоит в хранении в каждом узле v "псевдодистанции" $d[v]$, которая определена таким образом, что сумма псевдодистанций вдоль пути от v к корню его множества S_i равна глубине v в T_i; т.е. если путь от v к корню S_i представляет собой последовательность $v_0, v_1, \ldots, v_k$, где $v_0 = v_k$ и v_k — корень S_i, то глубина v в T_i равна $\sum_{j=0}^{k} d[v_j]$.

б) Приведите реализацию процедуры MAKE_TREE.

в) Покажите, как следует изменить процедуру FIND_SET для реализации процедуры FIND_DEPTH. Ваша реализация должна осуществлять сжатие пути, а время ее работы линейно зависеть от длины пути. Убедитесь в корректности обновления псевдодистанций вашей процедурой.

г) Покажите, как можно реализовать процедуру GRAFT(r, v), которая объединяет множества, содержащие r и v, модифицируя процедуры UNION и LINK. Убедитесь в корректности обновления псевдодистанций вашей процедурой. Не забывайте, что корень множества S_i не обязательно должен быть корнем соответствующего дерева T_i.

д) Дайте точную оценку времени работы в наихудшем случае последовательности из m операций MAKE_TREE, FIND_DEPTH и GRAFT, n из которых — операции MAKE_TREE.

21-3. **Алгоритм Таржана для автономного поиска наименьшего общего предка**

Наименьший общий предок (least common ancestor) двух узлов u и v в корневом дереве T представляет собой узел w, который среди узлов, являющихся предками как u, так и v, имеет наибольшую глубину. В ***задаче автономного поиска наименьшего общего предка*** (off-line least-common-ancestor problem) дано корневое дерево T и произвольное множество $P = \{\{u, v\}\}$ неупорядоченных пар узлов в T. Для каждой пары узлов из P требуется найти их наименьшего общего предка.

Для решения задачи автономного поиска наименьшего общего предка осуществляется обход дерева T путем вызова приведенной ниже процедуры LCA$(root\,[T])$. Предполагается, что перед вызовом процедуры все узлы помечены как WHITE (белые):

```
LCA(u)
 1  MAKE_SET(u)
 2  ancestor[FIND_SET(u)] ← u
 3  for (для) каждого v, дочернего по отношению к u
 4       do LCA(v)
 5          UNION(u, v)
 6          ancestor[FIND_SET(u)] ← u
 7  color[u] ← BLACK
 8  for (для) каждого v, такого что {u, v} ∈ P
 9       do if color[v] = BLACK
10             then print "Наименьшим общим предком"
                          u " и " v " является " ancestor[FIND_SET(v)]
```

а) Покажите, что строка 10 выполняется в точности по одному разу для каждой пары $\{u, v\} \in P$.

б) Покажите, что при вызове LCA(u) количество множеств в структуре данных для непересекающихся множеств равно глубине u в T.

в) Докажите, что процедура LCA правильно выводит наименьших общих предков для каждой пары $\{u, v\} \in P$.

г) Проанализируйте время работы процедуры LCA в предположении, что используется реализация непересекающихся множеств из раздела 21.3.

Заключительные замечания

Многие важные результаты о структурах данных для непересекающихся множеств в той или иной степени принадлежат Таржану (Tarjan). В частности, именно им [290, 292] установлена точная верхняя граница с применением очень медленно растущей инверсии $\widehat{\alpha}(m,n)$ функции Аккермана (Ackermann). (Функция $A_k(j)$, используемая в разделе 21.4, подобна функции Аккермана, а функция $\alpha(n)$ — ее инверсии. Для всех мыслимых значений m и n как $\alpha(n)$, так и $\widehat{\alpha}(m,n)$ не превышают 4.) Верхняя граница $O(m \lg^* n)$ была доказана несколько ранее Хопкрофтом (Hopcroft) и Ульманом (Ullman) [5, 155]. Материал раздела 21.4 основан на более позднем анализе Таржана [294], который, в свою очередь, опирается на анализ Козена (Kozen) [193]. Харфст (Harfst) и Рейнгольд (Reingold) [139] разработали версию доказательства полученных Таржаном оценок при помощи потенциалов.

Таржан и ван Леувен (van Leeuwen) [295] рассмотрели разные варианты эвристики со сжатием пути, включая однопроходные варианты, которые эффективнее двухпроходных в силу меньшего постоянного множителя. Позже Харфст и Рейнгольд [139] показали, какие (небольшие) изменения следует внести в потенциальную функцию для адаптации их анализа сжатия пути к однопроходному варианту Таржана. Габов (Gabow) и Таржан [103] показали, что в некоторых приложениях операции над непересекающимися множествами могут выполняться за время $O(m)$.

Таржан [291] показал, что для операций над произвольными структурами данных для непересекающихся множеств, удовлетворяющими определенным техническим условиям, нижняя граница времени работы равна $\Omega(m\,\widehat{\alpha}(m,n))$. Позже эта нижняя граница была обобщена Фредманом (Fredman) и Саксом (Saks) [97], которые показали, что в наихудшем случае эти операции требуют обращения к $\Omega(m\,\widehat{\alpha}(m,n))$ словам памяти длиной $\lg n$ битов.

ЧАСТЬ VI

Алгоритмы для работы с графами

Введение

Графы представляют собой распространенные структуры в информатике, и алгоритмы для работы с графами очень важны. Имеются сотни интересных вычислительных задач, сформулированных с использованием графов. В этой части мы коснемся только некоторых наиболее важных из них.

В главе 22 рассматриваются вопросы представления графов в компьютере и обсуждаются алгоритмы, основанные на обходе графа в ширину и в глубину. Приведены два применения обхода в глубину — топологическая сортировка ориентированного ациклического графа и разложение ориентированного графа на сильно связные компоненты.

В главе 23 описывается вычисление остовного дерева минимального веса. Такое дерево представляет собой набор ребер, связывающий все вершины графа с наименьшим возможным весом (каждое ребро обладает некоторым весом). Эта задача решается при помощи жадного алгоритма (см. главу 16).

В главах 24 и 25 рассматривается задача вычисления кратчайшего пути между вершинами, когда каждому ребру присвоена некоторая длина или вес. Глава 24 посвящена вычислению кратчайшего пути из одной вершины во все остальные, а в главе 25 рассматривается поиск кратчайших путей для всех пар вершин.

И наконец, в главе 26 рассматривается задача о максимальном потоке материала в сети (ориентированном графе) с определенным источником вещества, стоком и пропускной способностью каждого ребра. К этой общей задаче сводятся многие другие, так что хороший алгоритм ее решения может использоваться во многих приложениях.

При описании времени работы алгоритма над данным графом $G = (V, E)$ мы обычно определяем размер входного графа в терминах количества его вершин $|V|$ и количества ребер $|E|$ графа, т.е. размер входных данных описывается двумя, а не одним параметром. Для удобства и краткости в асимптотических обозначениях (таких, как O и Θ-обозначения), и *только* в них, символ V будет означать $|V|$, а символ E — $|E|$, т.е. когда мы будем говорить "время работы алгоритма равно $O(V\ E)$", то это означает "время работы алгоритма равно $O(|V|\ |E|)$". Такое соглашение позволяет сделать формулы для времени работы более удобочитаемыми без риска неоднозначности.

Еще одно соглашение принято для псевдокода. Мы обозначаем множество вершин графа G как $V[G]$, а множество ребер — как $E[G]$, т.е. в псевдокоде множества вершин и ребер рассматриваются как атрибуты графа.

ГЛАВА 22

Элементарные алгоритмы для работы с графами

В этой главе рассматриваются методы представления графов, а также обход графа. Под обходом графа понимается систематическое перемещение по ребрам графа, при котором посещаются все его вершины. Алгоритм обхода графа может многое сказать о его структуре, поэтому многие другие алгоритмы начинают свою работу с получения информации о структуре путем обхода графа. Многие алгоритмы для работы с графами организованы как простое усовершенствование базовых алгоритмов обхода графа.

В разделе 22.1 рассматриваются два наиболее распространенных способа представления графов — списки смежных вершин и матрица смежности. В разделе 22.2 представлен простой алгоритм обхода графа, который называется поиском в ширину, и соответствующее этому алгоритму дерево. В разделе 22.3 представлен алгоритм поиска в глубину и доказываются некоторые свойства этого вида обхода графа. В разделе 22.4 вы познакомитесь с первым реальным применением поиска в глубину — топологической сортировкой ориентированного ациклического графа. Еще одно применение поиска в глубину — поиск сильно связных компонентов графа — приводится в разделе 22.5.

22.1 Представление графов

Имеется два стандартных способа представления графа $G = (V, E)$: как набора списков смежных вершин или как матрицы смежности. Оба способа представления применимы как для ориентированных, так и для неориентированных

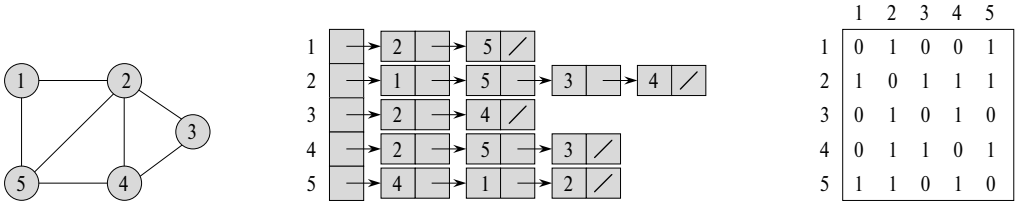


Рис. 22.1. Два представления неориентированного графа

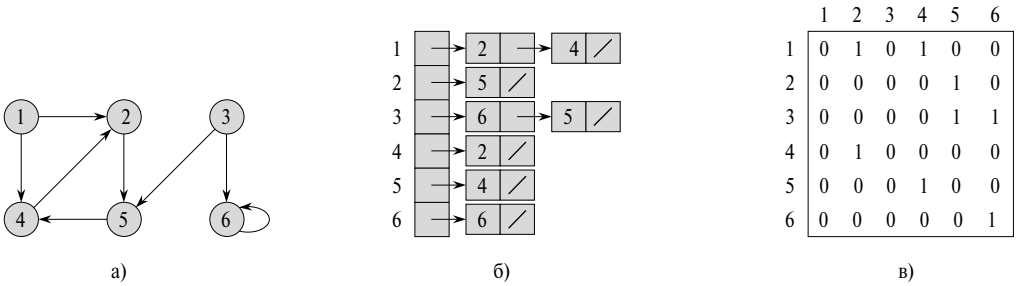


Рис. 22.2. Два представления ориентированного графа

графов. Обычно более предпочтительно представление с помощью списков смежности, поскольку оно обеспечивает компактное представление ***разреженных*** (sparse) графов, т.е. таких, для которых $|E|$ гораздо меньше $|V|^2$. Большинство алгоритмов, представленных в данной книге, предполагают, что входной граф представлен именно в виде списка смежности. Представление при помощи матрицы смежности предпочтительнее в случае ***плотных*** (dense) графов, т.е. когда значение $|E|$ близко к $|V|^2$, или когда нам надо иметь возможность быстро определить, имеется ли ребро, соединяющие две данные вершины. Например, два алгоритма поиска кратчайшего пути для всех пар вершин, представленные в главе 25, используют представление графов именно в виде матриц смежности.

Представление графа $G = (V, E)$ в виде ***списка смежности*** (adjacency-list representation) использует массив Adj из $|V|$ списков, по одному для каждой вершины из V. Для каждой вершины $u \in V$ список $Adj\,[u]$ содержит все вершины v, такие что $(u, v) \in E$, т.е. $Adj\,[u]$ состоит из всех вершин, смежных с u в графе G (список может содержать и не сами вершины, а указатели на них). Вершины в каждом списке обычно хранятся в произвольном порядке. На рис. 22.1*б* показано такое представление графа, изображенного на рис. 22.1*а* (на рис. 22.1*в* представлена его матрица смежности). Аналогично, на рис. 22.2*б* и рис. 22.2*в* показаны список и матрица смежности ориентированного графа, изображенного на рис. 22.2*а*.

Если G — ориентированный граф, то сумма длин всех списков смежности равна $|E|$, поскольку ребру (u, v) однозначно соответствует элемент v в списке $Adj\,[u]$. Если G — неориентированный граф, то сумма длин всех списков смежности равна $2\,|E|$, поскольку ребро (u, v), будучи неориентированным, появляется в списке $Adj\,[v]$ как u, и в списке $Adj\,[u]$ — как v. Как для ориентированных, так и для неориентированных графов представление в виде списков требует объем памяти, равный $\Theta\,(V + E)$.

Списки смежности легко адаптируются для представления ***взвешенных графов*** (weighted graph), т.е. графов, с каждым ребром которых связан определенный ***вес*** (weight), обычно определяемый ***весовой функцией*** (weight function) $w : E \to \mathbf{R}$. Например, пусть $G = (V, E)$ — взвешенный граф с весовой функцией w. Вес $w\,(u, v)$ ребра $(u, v) \in E$ просто хранится вместе с вершиной v в списке смежности u. Представление с помощью списков смежности достаточно гибко в том смысле, что легко адаптируется для поддержки многих других вариантов графов.

Потенциальный недостаток представления при помощи списков смежности заключается в том, что при этом нет более быстрого способа определить, имеется ли данное ребро (u, v) в графе, чем поиск v в списке $Adj\,[u]$. Этот недостаток можно устранить ценой использования асимптотически большего количества памяти и представления графа с помощью матрицы смежности (в упражнении 22.1-8 предлагается вариант списков смежности, позволяющий ускорить поиск ребер).

Представление графа $G = (V, E)$ с помощью ***матрицы смежности*** (adjacency-matrix representation) предполагает, что вершины перенумерованы в некотором порядке числами $1, 2, \ldots, |V|$. В таком случае представление графа G с использованием матрицы смежности представляет собой матрицу $A = (a_{ij})$ размером $|V| \times |V|$, такую что

$$a_{ij} = \begin{cases} 1 & \text{если } (i, j) \in E, \\ 0 & \text{в противном случае.} \end{cases}$$

На рис. 22.1*в* и 22.2*в* показаны представления с использованием матрицы смежности неориентированного и ориентированного графов, показанных на рис. 22.1*а* и 22.2*а* соответственно. Матрица смежности графа требует объем памяти, равный $\Theta\left(V^2\right)$, независимо от количества ребер графа.

Обратите внимание на симметричность матрицы смежности на рис. 22.1*в* относительно главной диагонали. Определим ***транспонированную*** (transpose) матрицу $A = (a_{ij})$ как матрицу $A^{\mathrm{T}} = \left(a_{ij}^{\mathrm{T}}\right)$, у которой $a_{ij}^{\mathrm{T}} = a_{ji}$. Поскольку граф неориентирован, (u, v) и (v, u) представляют одно и то же ребро, так что матрица смежности неориентированного графа совпадает с транспонированной матрицей смежности, т.е. $A = A^{\mathrm{T}}$. В ряде приложений это свойство позволяет хранить только элементы матрицы, находящиеся на главной диагонали и выше нее, что позволяет почти в два раза сократить необходимое количество памяти.

Так же, как и представление со списками смежности, представление с матрицами смежности можно использовать для взвешенных графов. Например, если $G = (V, E)$ — взвешенный граф с весовой функцией w, то вес $w(u, v)$ ребра $(u, v) \in E$ хранится в записи в строке u и столбце v матрицы смежности. Если ребро не существует, то в соответствующем элементе матрицы хранится значение NIL, хотя для многих приложений удобнее использовать некоторое значение, такое как 0 или ∞.

Хотя список смежности асимптотически, как минимум, столь же эффективен в плане требуемой памяти, как и матрица смежности, простота последней делает ее предпочтительной при работе с небольшими графами. Кроме того, в случае невзвешенных графов для представления одного ребра достаточно одного бита, что позволяет существенно сэкономить необходимую для представления память.

Упражнения

22.1-1. Имеется представление ориентированного графа с использованием списков смежности. Как долго будут вычисляться исходящие степени всех вершин графа? А входящие степени?

22.1-2. Имеется представление с использованием списков смежности полного бинарного дерева с 7 вершинами. Приведите его представление с использованием матрицы смежности (считаем, что вершины пронумерованы от 1 до 7, как в бинарной пирамиде).

22.1-3. При ***транспонировании*** (transpose) графа $G = (V, E)$ мы получаем граф $G^{\mathrm{T}} = \left(V, E^{\mathrm{T}}\right)$, где $E^{\mathrm{T}} = \{(v, u) \in V \times V : (u, v) \in E\}$, т.е. граф G^{T} представляет собой граф G с обратным направлением ребер. Опишите эффективный алгоритм транспонирования графа, как для представления с использованием списков смежности, так и для матриц смежности. Проанализируйте время работы ваших алгоритмов.

22.1-4. Имеется представление мультиграфа $G = (V, E)$ с использованием списков смежности. Опишите алгоритм со временем работы $O(V + E)$ для вычисления представления со списками смежности "эквивалентного" неориентированного графа $G' = (V, E')$, где E' состоит из ребер из E, где кратные ребра заменены обычными и удалены ребра-циклы.

22.1-5. ***Квадратом*** (square) ориентированного графа $G = (V, E)$ является граф $G^2 = \left(V, E^2\right)$, такой что $(u, w) \in E^2$ тогда и только тогда, когда для некоторой вершины $v \in V$ и $(u, v) \in E$, и $(v, w) \in E$ (т.е. G^2 содержит ребро между u и w, если в G между u и w имеется путь, состоящий из двух ребер). Опишите эффективный алгоритм вычисления квадрата графа как для представления с использованием списков смежности, так и для матриц смежности. Проанализируйте время работы ваших алгоритмов.

22.1-6. При использовании матриц смежности большинство алгоритмов для работы с графами требуют времени $\Omega\left(V^2\right)$, но имеются и некоторые исключения. Покажите, что определение того, содержит ли граф G ***всеобщий сток*** (universal sink), т.е. вершину со входящей степенью, равной $|V|-1$, и с исходящей степенью 0, возможно выполнить за время $O\left(V\right)$, если использовать представление графа при помощи матрицы смежности.

22.1-7. ***Матрицей инциденций*** (incidence matrix) ориентированного графа $G = (V, E)$ является матрица $B = (b_{ij})$ размером $|V| \times |E|$, такая что

$$b_{ij} = \begin{cases} -1 & \text{если ребро } j \text{ выходит из вершины } i, \\ 1 & \text{если ребро } j \text{ входит в вершину } i, \\ 0 & \text{в противном случае.} \end{cases}$$

Поясните, что представляют собой элементы матрицы $B\,B^{\mathrm{T}}$, где B^{T} — транспонированная матрица B.

22.1-8. Предположим, что вместо связанного списка каждый элемент массива $Adj\,[u]$ представляет собой хеш-таблицу, содержащую вершины v, для которых $(u, v) \in E$. Чему равно математическое ожидание времени определения наличия ребра в графе, если проверка всех ребер выполняется с одинаковой вероятностью. Какие недостатки имеет данная схема? Предложите другие структуры данных для списков ребер, которые позволяют решать поставленную задачу. Имеет ли ваша схема преимущества или недостатки по сравнению с хеш-таблицами?

22.2 Поиск в ширину

Поиск в ширину (breadth-first search) представляет собой один из простейших алгоритмов для обхода графа и является основой для многих важных алгоритмов для работы с графами. Например, алгоритм Прима (Prim) поиска минимального остовного дерева (раздел 23.2) или алгоритм Дейкстры (Dijkstra) поиска кратчайшего пути из одной вершины (раздел 24.3) используют идеи, сходные с идеями, используемыми при поиске в ширину.

Пусть задан граф $G = (V, E)$ и выделена ***исходная*** (source) вершина s. Алгоритм поиска в ширину систематически обходит все ребра G для "открытия" всех вершин, достижимых из s, вычисляя при этом расстояние (минимальное количество ребер) от s до каждой достижимой из s вершины. Кроме того, в процессе обхода строится "дерево поиска в ширину" с корнем s, содержащее все достижимые вершины. Для каждой достижимой из s вершины v путь в дереве поиска в ширину соответствует кратчайшему (т.е. содержащему наименьшее количество

ребер) пути от s до v в G. Алгоритм работает как для ориентированных, так и для неориентированных графов.

Поиск в ширину имеет такое название потому, что в процессе обхода мы идем вширь, т.е. перед тем как приступить к поиску вершин на расстоянии $k+1$, выполняется обход всех вершин на расстоянии k.

Для отслеживания работы алгоритма поиск в ширину раскрашивает вершины графа в белый, серый и черный цвета. Изначально все вершины белые, и позже они могут стать серыми, а затем черными. Когда вершина ***открывается*** (discovered) в процессе поиска, она окрашивается. Таким образом, серые и черные вершины — это вершины, которые уже были открыты, но алгоритм поиска в ширину поразному работает с ними, чтобы обеспечить объявленный порядок обхода. Если $(u, v) \in E$ и вершина u черного цвета, то вершина v либо серая, либо черная, т.е. все вершины, смежные с черной, уже открыты. Серые вершины могут иметь белых соседей, представляя собой границу между открытыми и неоткрытыми вершинами.

Поиск в ширину строит дерево поиска в ширину, которое изначально состоит из одного корня, которым является исходная вершина s. Если в процессе сканирования списка смежности уже открытой вершины u открывается белая вершина v, то вершина v и ребро (u, v) добавляются в дерево. Мы говорим, что u является ***предшественником*** (predecessor), или ***родителем*** (parent), v в дереве поиска вширь. Поскольку вершина может быть открыта не более одного раза, она имеет не более одного родителя. Взаимоотношения предков и потомков определяются в дереве поиска в ширину как обычно — если u находится на пути от корня s к вершине v, то u является предком v, а v — потомком u.

Приведенная ниже процедура поиска в ширину BFS предполагает, что входной граф $G = (V, E)$ представлен при помощи списков смежности. Кроме того, поддерживаются дополнительные структуры данных в каждой вершине графа. Цвет каждой вершины $u \in V$ хранится в переменной $color\,[u]$, а предшественник — в переменной $\pi\,[u]$. Если предшественника у u нет (например, если $u = s$ или u не открыта), то $\pi\,[u] = \text{NIL}$. Расстояние от s до вершины u, вычисляемое алгоритмом, хранится в поле $d\,[u]$. Алгоритм использует очередь Q (см. раздел 10.1) для работы с множеством серых вершин:

```
BFS(G, s)
 1  for (для) каждой вершины u ∈ V[G] − s
 2      do color[u] ← WHITE
 3         d[u] ← ∞
 4         π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
```

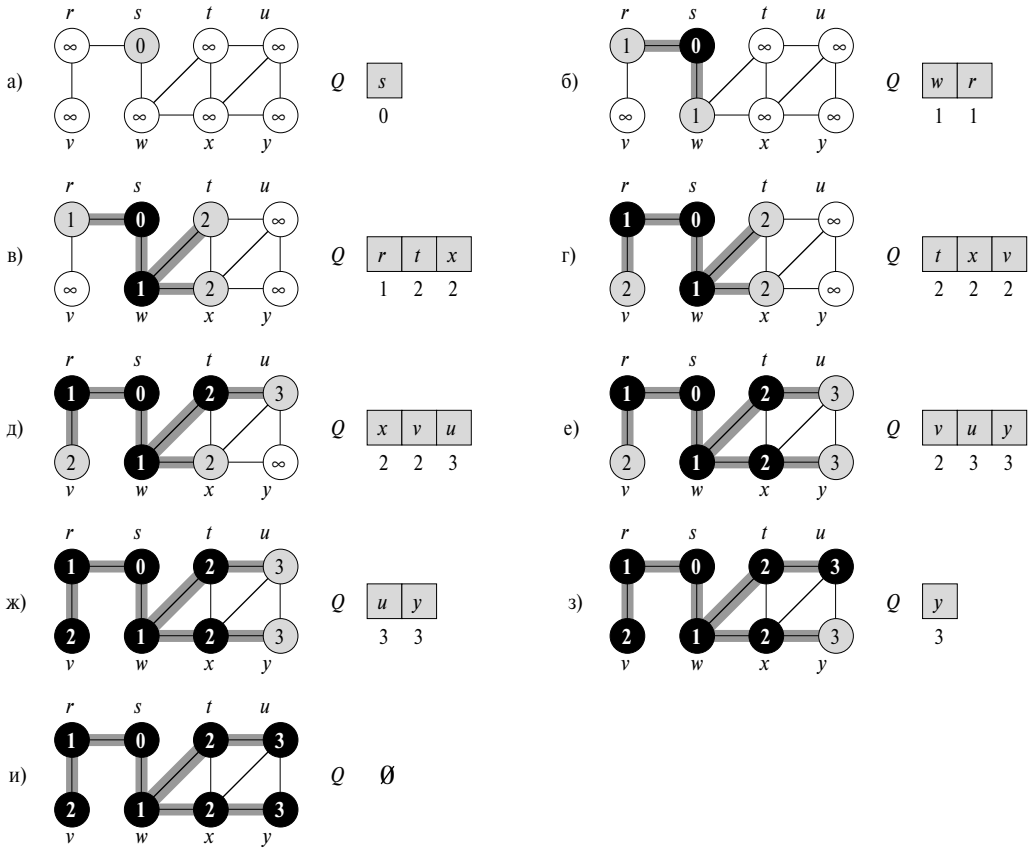


Рис. 22.3. Выполнение процедуры BFS над неориентированным графом

```
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11        do u ← DEQUEUE(Q)
12           for (для) каждой v ∈ Adj[u]
13                do if color[v] = WHITE
14                      then color[v] ← GRAY
15                           d[v] ← d[u] + 1
16                           π[v] ← u
17                           ENQUEUE(Q, v)
18           color[u] ← BLACK
```

На рис. 22.3 проиллюстрирована работа процедуры BFS. Внутри каждой вершины графа u приведено значение $d[u]$, а состояние очереди Q показано на

момент начала каждой итерации цикла **while** в строках 10–18. Рядом с элементами очереди показаны расстояния до корня.

Процедура BFS работает следующим образом. В строках 1–4 все вершины, за исключением исходной вершины s, окрашиваются в белый цвет, для каждой вершины u полю $d[u]$ присваивается значение ∞, а в качестве родителя для каждой вершины устанавливается NIL. В строке 5 исходная вершина s окрашивается в серый цвет, поскольку она рассматривается как открытая в начале процедуры. В строке 6 ее полю $d[s]$ присваивается значение 0, а в строке 7 ее родителем становится NIL. В строках 8–9 создается пустая очередь Q, в которую помещается один элемент s.

Цикл **while** в строках 10–18 выполняется до тех пор, пока остаются серые вершины (т.е. открытые, но списки смежности которых еще не просмотрены). Инвариант данного цикла выглядит следующим образом:

> При выполнении проверки в строке 10 очередь Q состоит из множества серых вершин.

Хотя мы не намерены использовать этот инвариант для доказательства корректности алгоритма, легко увидеть, что он выполняется перед первой итерацией и что каждая итерация цикла сохраняет инвариант. Перед первой итерацией единственной серой вершиной и единственной вершиной в очереди Q, является исходная вершина s. В строке 11 определяется серая вершина u в голове очереди Q, которая затем удаляется из очереди. Цикл **for** в строках 12–17 просматривает все вершины v в списке смежности u. Если вершина v белая, значит, она еще не открыта, и алгоритм открывает ее, выполняя строки 14–17. Вершине назначается серый цвет, дистанция $d[v]$ устанавливается равной $d[u] + 1$, а в качестве ее родителя указывается вершина u. После этого вершина помещается в хвост очереди Q. После того как все вершины из списка смежности u просмотрены, вершине u присваивается черный цвет. Инвариант цикла сохраняется, так как все вершины, которые окрашиваются в серый цвет (строка 14), вносятся в очередь (строка 17), а вершина, которая удаляется из очереди (строка 11), окрашивается в черный цвет (строка 18).

Результат поиска в ширину может зависеть от порядка просмотра вершин, смежных с данной вершиной, в строке 12. Дерево поиска в ширину может варьироваться, но расстояния d, вычисленные алгоритмом, не зависят от порядка просмотра (см. упражнение 22.2-4).

Анализ

Перед тем как рассматривать различные свойства поиска в ширину, начнем с самого простого — оценки времени работы алгоритма для входного графа $G = (V, E)$. Мы воспользуемся групповым анализом, описанным в разделе 17.1.

После инициализации ни одна вершина не окрашивается в белый цвет, поэтому проверка в строке 13 гарантирует, что каждая вершина вносится в очередь не более одного раза, а следовательно, и удаляется из очереди она не более одного раза. Операции внесения в очередь и удаления из нее требуют $O(1)$ времени, так что общее время операций с очередью составляет $O(V)$. Поскольку каждый список смежности сканируется только при удалении соответствующей вершины из очереди, каждый список сканируется не более одного раза. Так как сумма длин всех списков смежности равна $\Theta(E)$, общее время, необходимое для сканирования списков, равно $O(E)$. Накладные расходы на инициализацию равны $O(V)$, так что общее время работы процедуры BFS составляет $O(V+E)$. Таким образом, время поиска в ширину линейно зависит от размера представления графа G с использованием списков смежности.

Кратчайшие пути

В начале этого раздела мы говорили о том, что поиск в ширину находит расстояния до каждой достижимой вершины в графе $G=(V,E)$ от исходной вершины $s \in V$. Определим ***длину кратчайшего пути*** (shortest-path distance) $\delta(s,v)$ от s до v как минимальное количество ребер на каком-либо пути от s к v. Если пути от s к v не существует, то $\delta(s,v)=\infty$. Путь длиной $\delta(s,v)$ от s к v называется ***кратчайшим путем***[1] (shortest path) от s к v. Перед тем как показать, что поиск в ширину вычисляет длины кратчайших путей, рассмотрим важное свойство длин кратчайших путей.

Лемма 22.1. Пусть $G=(V,E)$ — ориентированный или неориентированный граф, а $s \in V$ — произвольная вершина. Тогда для любого ребра $(u,v) \in E$ справедливо соотношение $\delta(s,v) \leqslant \delta(s,u)+1$.

Доказательство. Если вершина u достижима из s, то достижима и вершина v. В этом случае кратчайший путь из s в v не может быть длиннее, чем кратчайший путь из s в u, за которым следует ребро (u,v), так что указанное неравенство выполняется. Если же вершина u недостижима из вершины s, то $\delta(s,u)=\infty$ и неравенство выполняется и в этом случае. ■

Мы хотим показать, что процедура BFS корректно вычисляет $d[v]=\delta(s,v)$ для каждой вершины $v \in V$. Сначала покажем, что $d[v]$ ограничивает $\delta(s,v)$ сверху.

[1] В главах 24 и 25 мы обобщим понятие кратчайшего пути для взвешенных графов, в которых каждое ребро имеет вес, выраженный действительным числом, а вес пути равен весу составляющих его ребер. Графы, рассматриваемые в данной главе, являются невзвешенными (или, что то же самое, все ребра имеют единичный вес).

Лемма 22.2. Пусть $G = (V, E)$ — ориентированный или неориентированный граф, и пусть процедура BFS выполняется над ним с исходной вершиной $s \in V$. Тогда по завершении процедуры для каждой вершины $v \in V$ значение $d[v]$, вычисленное процедурой BFS, удовлетворяет неравенству $d[v] \geqslant \delta(s, v)$.

Доказательство. Используем индукцию по количеству операций ENQUEUE. Наша гипотеза индукции заключается в том, что для всех $v \in V$ выполняется условие $d[v] \geqslant \delta(s, v)$.

Базисом индукции является ситуация непосредственно после внесения s в очередь в строке 9 процедуры BFS. В этой ситуации гипотеза индукции справедлива, поскольку $d[s] = 0 = \delta(s, s)$, а для всех $v \in V - \{s\}$ $d[v] = \infty \geqslant \delta(s, v)$.

На каждом шаге индукции рассмотрим белую вершину v, которая открывается в процессе поиска из вершины u. Согласно гипотезе индукции, $d[u] \geqslant \delta(s, u)$. На основании присвоения в строке 15 и леммы 22.1, получаем:

$$d[v] = d[u] + 1 \geqslant \delta(s, u) + 1 \geqslant \delta(s, v).$$

После этого вершина v вносится в очередь. Поскольку она при этом окрашивается в серый цвет, а строки 14–17 выполняются только для белых вершин, рассмотренная вершина v больше в очередь поставлена не будет. Таким образом, ее значение $d[v]$ также больше не изменяется, так что гипотеза индукции выполняется. ■

Для доказательства того что $d[v] = \delta(s, v)$, мы должны сначала более точно разобраться, как работает очередь Q в процедуре BFS. Следующая лемма показывает, что в любой момент времени в очереди находится не более двух различных значений d.

Лемма 22.3. Предположим, что в процессе выполнения процедуры BFS над графом $G = (V, E)$ очередь Q содержит вершины $\langle v_1, v_2, \ldots, v_r \rangle$, где v_1 — голова очереди Q, а v_r — ее хвост. Тогда для всех $i = 1, 2, \ldots, r-1$ справедливы соотношения $d[v_r] \leqslant d[v_1] + 1$ и $d[v_i] \leqslant d[v_{i+1}]$.

Доказательство. Доказательство использует индукцию по числу операций с очередью. Изначально, когда в очереди содержится только одна вершина s, лемма определенно выполняется.

На каждом шаге индукции мы должны доказать, что лемма выполняется как после помещения вершины в очередь, так и после извлечения ее оттуда. Если из очереди извлекается ее голова v_1, новой головой очереди становится вершина v_2 (если очередь после извлечения очередной вершины становится пустой, то лемма выполняется исходя из того, что очередь пуста). В соответствии с гипотезой индукции, $d[v_1] \leqslant d[v_2]$. Но тогда мы имеем $d[v_r] \leqslant d[v_1] + 1 \leqslant d[v_2] + 1$, а все остальные неравенства не изменяются. Следовательно, лемма справедлива, когда новой головой очереди становится v_2.

Внесем в очередь вершину в строке 17 процедуры BFS. При этом она становится вершиной v_{r+1}. В этот момент времени вершина u, список смежности которой сканируется, уже удалена из очереди, так что, согласно гипотезе индукции, для новой головы очереди v_1 выполняется неравенство $d[v_1] \geqslant d[u]$. Следовательно, $d[v_{r+1}] = d[v] = d[u] + 1 \leqslant d[v_1] + 1$. Кроме того, согласно гипотезе индукции, $d[v_r] \leqslant d[u] + 1$, так что $d[v_r] \leqslant d[u] + 1 = d[v] = d[v_{r+1}]$, а остальные неравенства остаются неизменными. Следовательно, лемма выполняется при внесении в очередь новой вершины. ■

Приведенное ниже следствие показывает, что значения d вносимых в очередь вершин монотонно возрастают.

Следствие 22.4. Предположим, что вершины v_i и v_j вносятся в очередь в процессе работы процедуры BFS и что v_i вносится в очередь до v_j. Тогда в момент внесения в очередь вершины v_j выполняется неравенство $d[v_i] \leqslant d[v_j]$.

Доказательство. Доказательство вытекает непосредственно из леммы 22.3 и того факта, что каждая вершина получает конечное значение d в процессе выполнения процедуры BFS не более одного раза. ■

Теперь мы можем доказать, что методом поиска в ширину корректно определяются длины кратчайших путей.

Теорема 22.5 (Корректность поиска в ширину). Пусть $G = (V, E)$ — ориентированный или неориентированный граф, и пусть процедура BFS выполняется над графом G с определенной исходной вершиной s. Тогда в процессе работы BFS открывает все вершины $v \in V$, достижимые из s, и по окончании работы для всех $v \in V$ $d[v] = \delta(s, v)$. Кроме того, для всех достижимых из s вершин $v \neq s$, один из кратчайших путей от s к v — это путь от s к $\pi[v]$, за которым следует ребро $(\pi[v], v)$.

Доказательство. Предположим, с целью использовать доказательство от обратного, что у некоторой вершины значение d не равно длине кратчайшего пути. Пусть v — вершина с минимальной длиной $\delta(s, v)$ среди вершин, у которых оказывается неверным вычисленное значение d. Очевидно, что $v \neq s$. Согласно лемме 22.2, $d[v] \geqslant \delta(s, v)$, так что, в силу нашего исходного предположения, $d[v] > \delta(s, v)$. Вершина v должна быть достижима из s, потому что если это не так, то $\delta(s, v) = \infty \geqslant d[v]$. Пусть u — вершина, непосредственно предшествующая v на кратчайшем пути от s к v, так что $\delta(s, v) = \delta(s, u) + 1$. Так как $\delta(s, u) < \delta(s, v)$ и в силу нашего выбора вершины, v $d[u] = \delta(s, u)$. Объединяя полученные результаты, имеем:

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1. \tag{22.1}$$

Теперь рассмотрим момент, когда процедура BFS удаляет узел u из очереди Q в строке 11. В этот момент вершина v может быть белой, серой или черной. Мы покажем, что для каждого из этих случаев мы получим противоречие с неравенством (22.1). Если v белая, то в строке 15 выполняется присвоение $d[v] = d[u]+1$, противоречащее (22.1). Если v — черная, то она уже удалена из очереди и, согласно следствию 22.4, $d[v] \leqslant d[u]$, что опять-таки противоречит (22.1). Если v — серая, то она была окрашена в этот цвет при удалении из очереди некоторой вершины w, которая была удалена ранее вершины u и для которой выполняется равенство $d[v] = d[w] + 1$. Однако из следствия 22.4 вытекает, что $d[w] \leqslant d[u]$, поэтому $d[v] \leqslant d[u] + 1$, что тоже противоречит (22.1).

Итак, мы заключили, что $d[v] = \delta(s, v)$ для всех $v \in V$. Процедурой должны быть открыты все достижимые из s вершины, потому что если это не так, то они будут иметь бесконечное значение d. Для завершения доказательства теоремы заметим, что если $\pi[v] = u$, то $d[v] = d[u]+1$. Следовательно, мы можем получить кратчайший путь из s в v, взяв кратчайший путь из s в $\pi[v]$ и затем проходя по ребру $(\pi[v], v)$. ■

Деревья поиска в ширину

Процедура BFS строит в процессе обхода графа дерево поиска в ширину, как показано на рис. 22.3. Дерево представлено при помощи поля π в каждой вершине. Говоря более формально, для графа $G = (V, E)$ с исходной вершиной s мы определяем ***подграф предшествования*** (predecessor subgraph) графа G как $G_\pi = (V_\pi, E_\pi)$, где

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

и

$$E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}.$$

Подграф предшествования G_π является ***деревом поиска в ширину*** (breadth-first tree), если V_π состоит из вершин, достижимых из s, и для всех $v \in V_\pi$ в G_π имеется единственный простой путь из s в v, такой что он одновременно является кратчайшим путем из s в v в G. Дерево поиска в ширину является деревом, поскольку оно является связным и $|E_\pi| = |V_\pi| - 1$ (см. теорему Б.2). Ребра в E_π называются ***ребрами дерева*** (tree edges).

Следующая лемма показывает, что после выполнения процедуры BFS над графом G с исходной вершиной s подграф предшествования представляет собой дерево поиска в ширину.

Лемма 22.6. Будучи примененной к ориентированному или неориентированному графу $G = (V, E)$, процедура BFS строит π так, что подграф предшествования $G_\pi = (V_\pi, E_\pi)$ является деревом поиска в ширину.

Доказательство. В строке 16 процедуры BFS присвоение $\pi[v] = u$ выполняется тогда и только тогда, когда $(u, v) \in E$ и $\delta(s, v) < \infty$, т.е. если v достижимо из s. Следовательно, V_π состоит из вершин V, достижимых из s. Поскольку G_π образует дерево, согласно теореме Б.2 оно содержит единственный путь из s в каждую вершину множества V_π. Индуктивно применяя теорему 22.5, мы заключаем, что каждый такой путь является кратчайшим. ■

Приведенная далее процедура выводит все вершины на пути из s в v исходя из предположения, что дерево поиска в ширину уже построено процедурой BFS.

```
PRINT_PATH(G, s, v)
1  if v = s
2     then print s
3     else if π[v] = NIL
4             then print "Путь из" s "в" v "отсутствует"
5             else PRINT_PATH(G, s, π[v])
6                  print v
```

Время работы процедуры линейно зависит от количества выводимых вершин, так как каждый рекурсивный вызов процедуры осуществляется для пути, который на одну вершину короче текущего.

Упражнения

22.2-1. Покажите, какие значения d и π получатся в результате поиска в ширину в ориентированном графе, показанном на рис. 22.2*a*, если в качестве исходной взять вершину 3.

22.2-2. Покажите, какие значения d и π получатся в результате поиска в ширину в неориентированном графе, показанном на рис. 22.3, если в качестве исходной взять вершину u.

22.2-3. Чему будет равно время работы процедуры BFS, адаптированной для работы с матричным представлением графа?

22.2-4. Докажите, что при выполнении поиска в ширину значение $d[u]$, назначаемое вершине u, не зависит от порядка перечисления вершин в списках смежности. Используя рис. 22.3 в качестве примера, покажите, что вид дерева поиска в ширину, вычисленного с помощью процедуры BFS, может зависеть от порядка перечисления вершин в списках смежности.

22.2-5. Приведите пример ориентированного графа $G = (V, E)$, исходной вершины $s \in V$ и множества ребер дерева $E_\pi \subseteq E$, таких что для каждой вершины $v \in V$ единственный путь в графе (V, E_π) от s к v является кратчайшим путем в графе G, но множество ребер E_π невозможно получить

при помощи процедуры BFS ни при каком порядке вершин в списках смежности.

22.2-6. Предположим, что у нас есть n борцов, между любой парой которых может состояться, (а может и не состояться) поединок, и список r поединков. Разработайте алгоритм, который за время $O(n+r)$ определяет, можно ли разделить всех борцов на две команды так, чтобы в поединках встречались только борцы из разных команд, и если это возможно, то алгоритм должен выполнять такое разделение.

⋆ 22.2-7. ***Диаметр*** дерева $T=(V,E)$ определяется как

$$\max_{u,v\in V}\delta(u,v),$$

т.е. диаметр — это наибольшая длина кратчайшего пути в дереве. Разработайте эффективный алгоритм вычисления диаметра дерева и проанализируйте время его работы.

22.2-8. Пусть $G=(V,E)$ — связный неориентированный граф. Разработайте алгоритм для вычисления за время $O(V+E)$ пути в G, который проходит по каждому ребру из E по одному разу в каждом направлении. Придумайте способ выйти из лабиринта, если у вас в карманах имеется много монет.

22.3 Поиск в глубину

Стратегия поиска в глубину, как следует из ее названия, состоит в том, чтобы идти "вглубь" графа, насколько это возможно. При выполнении поиска в глубину исследуются все ребра, выходящие из вершины, открытой последней, и покидает вершину, только когда не остается неисследованных ребер — при этом происходит возврат в вершину, из которой была открыта вершина v. Этот процесс продолжается до тех пор, пока не будут открыты все вершины, достижимые из исходной. Если при этом остаются неоткрытые вершины, то одна из них выбирается в качестве новой исходной вершины и поиск повторяется уже из нее. Этот процесс повторяется до тех пор, пока не будут открыты все вершины.

Как и в случае поиска в ширину, когда вершина v открывается в процессе сканирования списка смежности уже открытой вершины u, процедура поиска записывает это событие, устанавливая поле предшественника $v\,\pi[v]$ равным u. В отличие от поиска в ширину, где подграф предшествования образует дерево, при поиске в глубину подграф предшествования может состоять из нескольких деревьев, так как поиск может выполняться из нескольких исходных вершин[2].

[2]Может показаться несколько странным то, что поиск в ширину ограничен только одной исходной вершиной, в то время как поиск в глубину может выполняться из нескольких исходных

Подграф предшествования (predecessor subgraph) поиска в глубину, таким образом, несколько отличается от такового для поиска в ширину. Мы определяем его как граф $G_\pi = (V, E_\pi)$, где

$$E_\pi = \{(\pi[v], v) : v \in V \text{ и } \pi[v] \neq \text{NIL}\}.$$

Подграф предшествования поиска в глубин образует ***лес поиска в глубину*** (depth-first forest), который состоит из нескольких ***деревьев поиска в глубину*** (depth-first trees). Ребра в E_π называются ***ребрами дерева*** (tree edges).

Как и в процессе выполнения поиска в ширину, вершины графа раскрашиваются в разные цвета, свидетельствующие о их состоянии. Каждая вершина изначально белая, затем при ***открытии*** (discover) в процессе поиска она окрашивается в серый цвет, и по ***завершении*** (finish), когда ее список смежности полностью сканирован, она становится черной. Такая методика гарантирует, что каждая вершина в конечном счете находится только в одном дереве поиска в глубину, так что деревья не пересекаются.

Помимо построения леса поиска в глубину, поиск в глубину также проставляет в вершинах ***метки времени*** (timestamp). Каждая вершина имеет две такие метки — первую $d[v]$, в которой указывается, когда вершина v открывается (и окрашивается в серый цвет), и вторая — $f[v]$, которая фиксирует момент, когда поиск завершает сканирование списка смежности вершины v и она становится черной. Эти метки используются многими алгоритмами и полезны при рассмотрении поведения поиска в глубину.

Приведенная ниже процедура DFS записывает в поле $d[u]$ момент, когда вершина u открывается, а в поле $f[u]$ — момент завершения работы с вершиной u. Эти метки времени представляют собой целые числа в диапазоне от 1 до $2|V|$, поскольку для каждой из $|V|$ вершин имеется только одно событие открытия и одно — завершения. Для каждой вершины u

$$d[u] < f[u]. \tag{22.2}$$

До момента времени $d[u]$ вершина имеет цвет WHITE, между $d[u]$ и $f[u]$ — цвет GRAY, а после $f[u]$ — цвет BLACK.

Далее представлен псевдокод алгоритма поиска в глубину. Входной граф G может быть как ориентированным, так и неориентированным. Переменная $time$ — глобальная и используется нами для меток времени.

вершин. Хотя концептуально поиск в ширину может выполняться из нескольких исходных вершин, а поиск в глубину — быть ограниченным одной исходной вершиной, такой подход отражает типичное использование результатов поиска. Поиск в ширину обычно используется для определения длин кратчайших путей (и связанного с ними графа предшествования) из данной вершины. Поиск в глубину, как мы увидим позже в этой главе, чаще используется в качестве подпрограммы в других алгоритмах.

```
DFS(G)
1  for (Для) каждой вершины u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for (Для) каждой вершины u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS_VISIT(u)
```

```
DFS_VISIT(u)
1  color[u] ← GRAY                          ▷ Открыта белая вершина u
2  time ← time +1
3  d[u] ← time
4  for (Для) каждой вершины v ∈ Adj[u]      ▷ Исследование ребра (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS_VISIT(v)
8  color[u] ← BLACK                         ▷ Завершение
9  f[u] ← time ← time +1
```

На рис. 22.4 проиллюстрировано выполнение процедуры DFS над графом, приведенном на рис. 22.2. Ребра, исследованные алгоритмом, либо закрашены (если этот ребра деревьев), либо помечены пунктиром (в противном случае). Ребра, не являющиеся ребрами деревьев, помечены на рисунке буквами B (обратные — back), F (прямые — forward) и C (перекрестные — cross). В вершинах указаны метки времени в формате открытие/завершение.

Процедура DFS работает следующим образом. В строках 1–3 все вершины окрашиваются в белый цвет, а их поля π инициализируются значением NIL. В строке 4 выполняется сброс глобального счетчика времени. В строках 5–7 поочередно проверяются все вершины из V, и когда обнаруживается белая вершина, она обрабатывается при помощи процедуры DFS_VISIT. Каждый раз при вызове процедуры DFS_VISIT(u) в строке 7, вершина u становится корнем нового дерева леса поиска в глубину. При возврате из процедуры DFS каждой вершине u сопоставляются два момента времени — ***время открытия*** (discovery time) $d\,[u]$ и ***время завершения*** (finishing time) $f\,[u]$.

При каждом вызове DFS_VISIT(u) вершина u изначально имеет белый цвет. В строке 1 она окрашивается в серый цвет, в строке 2 увеличивается глобальная переменная $time$, а в строке 3 выполняется запись нового значения переменной $time$ в поле времени открытия $d\,[u]$. В строках 4–7 исследуются все вершины, смежные с u, и выполняется рекурсивное посещение белых вершин. При рассмотрении в строке 4 вершины $v \in Adj\,[u]$, мы говорим, что ребро (u, v) ***исследуется*** (explored) поиском в глубину. И наконец, после того как будут исследованы все

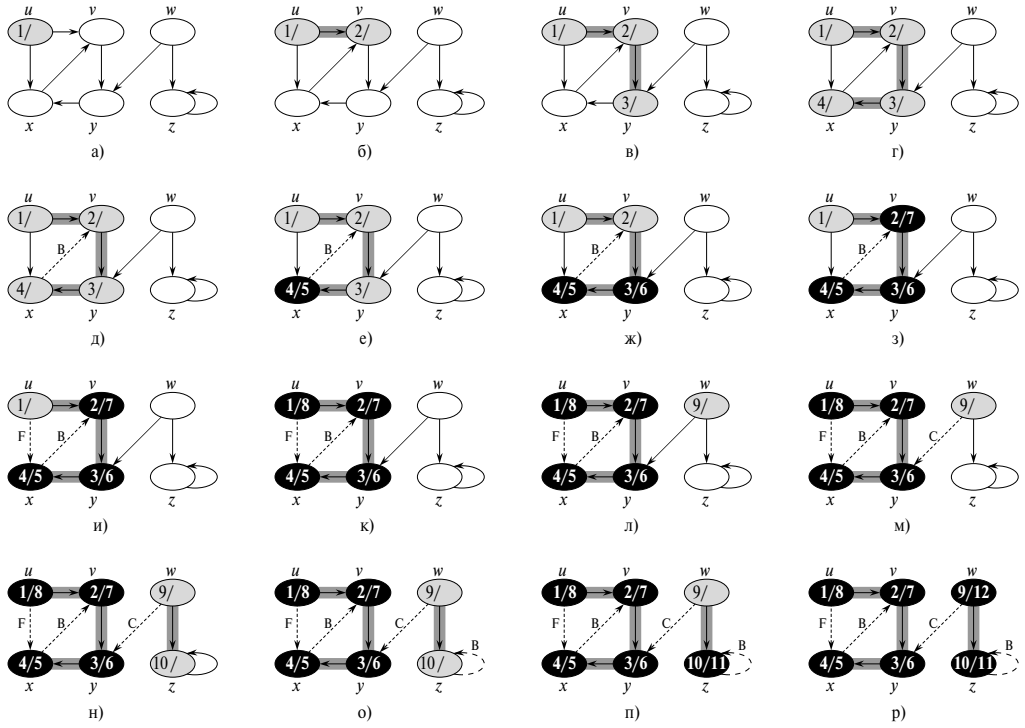


Рис. 22.4. Выполнение алгоритма поиска в глубину DFS над ориентированным графом

ребра, покидающие u, в строках 8–9 вершина u окрашивается в черный цвет, а в поле $f\,[u]$ записывается время завершения работы с ней.

Заметим, что результат поиска в глубину может зависеть от порядка, в котором выполняется рассмотрение вершин в строке 5 процедуры DFS, а также от порядка посещения смежных вершин в строке 4 процедуры DFS_VISIT. На практике это обычно не вызывает каких-либо проблем, так как обычно эффективно использован может быть *любой* результат поиска в глубину, приводя по сути к одинаковым результатам работы алгоритма, опирающегося на поиск в глубину.

Чему равно время работы процедуры DFS? Циклы в строках 1–3 и 5–7 процедуры DFS выполняются за время $\Theta\,(V)$, исключая время, необходимое для вызова процедуры DFS_VISIT. Как и в случае поиска в ширину, воспользуемся групповым анализом. Процедура DFS_VISIT вызывается ровно по одному разу для каждой вершины $v \in V$, так как она вызывается только для белых вершин, и первое, что она делает, — это окрашивает переданную в качестве параметра вершину в серый цвет. В процессе выполнения DFS_VISIT(v) цикл в строках 4–7

выполняется $|Adj\,[v]|$ раз. Поскольку

$$\sum_{v \in V} |Adj\,[v]| = \Theta\,(E)\,,$$

общая стоимость выполнения строк 4–7 процедуры DFS_VISIT равна $\Theta\,(E)$. Время работы процедуры DFS, таким образом, равно $\Theta\,(V + E)$.

Свойства поиска в глубину

Поиск в глубину дает нам важную информацию о структуре графа. Вероятно, наиболее фундаментальное свойство поиска в глубину заключается в том, что подграф предшествования G_π образует лес деревьев, поскольку структура деревьев поиска в глубину в точности отражает структуру рекурсивных вызовов процедуры DFS_VISIT. То есть $u = \pi\,[v]$ тогда и только тогда, когда DFS_VISIT(v) была вызвана при сканировании списка смежности вершины u. Кроме того, вершина v является потомком вершины u в лесу поиска в глубину тогда и только тогда, когда вершина u была серой в момент открытия вершины v.

Еще одно важное свойство поиска в глубину заключается в том, что времена открытия и завершения образуют ***скобочную структуру*** (parenthesis structure). Если мы представим открытие вершины u при помощи отрывающейся скобки "(u", а завершение — при помощи закрывающейся скобки "u)", то перечень открытий и завершений образует корректное выражение в смысле вложенности скобок. Например, поиск в глубину на рис. 22.5*а* соответствует скобочному выражению на рис. 22.5*б*. Еще одно утверждение о скобочной структуре приведено в следующей теореме.

Теорема 22.7 (Теорема о скобках). В любом поиске в глубину в (ориентированном или неориентированном) графе $G = (V, E)$ для любых двух вершин u и v выполняется ровно одно из трех следующих утверждений.

- Отрезки $[d[u], f[u]]$ и $[d[v], f[v]]$ не пересекаются, и ни u не является потомком v в лесу поиска в глубину, ни v не является потомком u.
- Отрезок $[d[u], f[u]]$ полностью содержится в отрезке $[d[v], f[v]]$, и u является потомком v в дереве поиска в глубину.
- Отрезок $[d[v], f[v]]$ полностью содержится в отрезке $[d[u], f[u]]$, и v является потомком u в дереве поиска в глубину.

Доказательство. Начнем со случая, когда $d\,[u] < d\,[v]$. При этом нам надо рассмотреть два подслучая, в зависимости от того, справедливо неравенство $d\,[v] < f\,[u]$ или нет. Первый подслучай соответствует справедливости неравенства $d\,[v] < f\,[u]$, так что вершина v была открыта, когда вершина u была окрашена

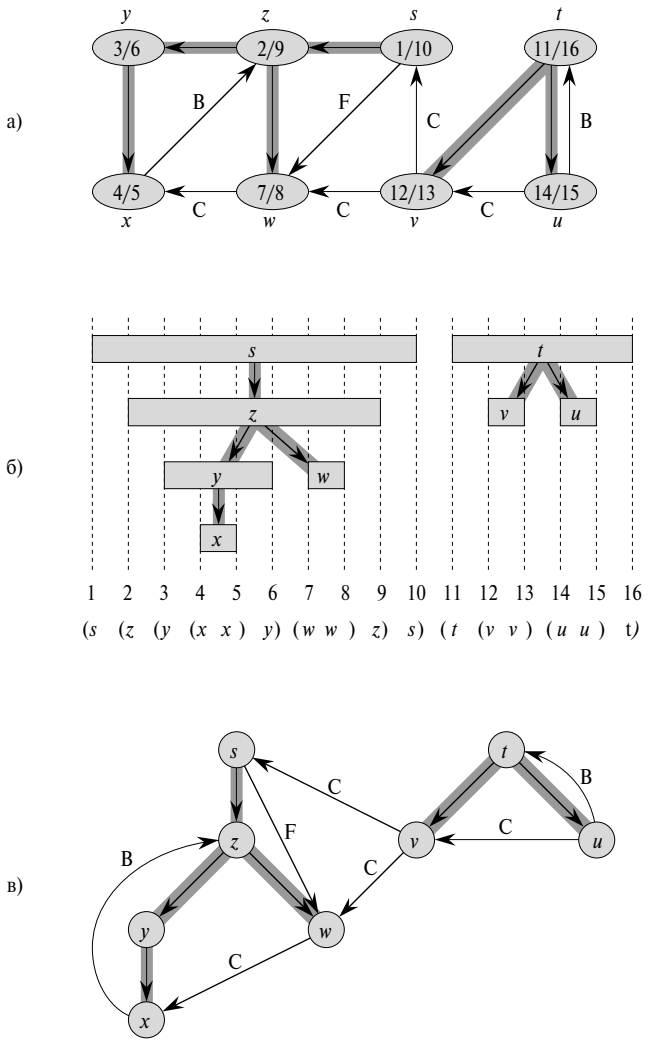


Рис. 22.5. Свойства поиска в глубину

в серый цвет. Отсюда следует, что v является потомком u. Кроме того, поскольку вершина v открыта позже, чем u, перед тем как вернуться для завершения поиска к вершине u, алгоритмом будут исследованы все выходящие из v ребра. В таком случае, следовательно, отрезок $[d[v], f[v]]$ полностью содержится в отрезке $[d[u], f[u]]$. В другом подслучае $f[u] < d[v]$, и из неравенства следует, что отрезки $[d[u], f[u]]$ и $[d[v], f[v]]$ не пересекаются. Поскольку отрезки не пересекаются, не происходит открытие одной вершины, пока другая имеет серый цвет, так что ни одна вершина не является потомком другой.

Случай, когда $d[v] < d[u]$, рассматривается аналогично, происходит только смена ролей u и v. ■

Следствие 22.8 (Вложенность интервалов потомков). Вершина v является потомком u (отличным от самого u) в лесу поиска в глубину в (ориентированном или неориентированном) графе G тогда и только тогда, когда $d[u] < d[v] < f[v] < f[u]$.

Доказательство. Непосредственно вытекает из теоремы 22.7. ■

Следующая теорема дает еще одну важное указание о том, когда одна вершина в лесу поиска в глубину является потомком другой.

Теорема 22.9 (Теорема о белом пути). В лесу поиска в глубину (ориентированного или неориентированного) графа $G = (V, E)$ вершина v является потомком вершины u тогда и только тогда, когда в момент времени $d[u]$ (открытие вершины u) вершина v достижима из u по пути, состоящему только из белых вершин.

Доказательство. $\Rightarrow$: Предположим, что v является потомком u. Пусть w — произвольная вершина на пути между u и v в дереве поиска в глубину, так что w является потомком u. Согласно следствию 22.8, $d[u] < d[w]$, так что в момент времени $d[u]$ вершина w — белая.

$\Leftarrow$: Предположим, что в момент времени $d[u]$ вершина v достижима из u вдоль пути, состоящего из белых вершин, но v не становится потомком u в дереве поиска в глубину. Без потери общности предположим, что все остальные вершины вдоль пути становятся потомками u (в противном случае в качестве v выберем на пути ближайшую к u вершину, которая не становится потомком u). Пусть w — предшественник v на пути, так что w является потомком u (w и u могут быть в действительности одной вершиной) и, согласно следствию 22.8, $f[w] \leqslant f[u]$. заметим, что вершина v должна быть открыта после того, как открыта u, но перед тем, как завершена обработка w. Таким образом, $d[u] < d[v] < f[w] < f[u]$. Из теоремы 22.7 следует, что отрезок $[d[v], f[v]]$ полностью содержится внутри отрезка $[d[u], f[u]]$. Согласно следствию 22.8, вершина v должна в конечном итоге быть потомком вершины u. ■

Классификация ребер

Еще одно интересное свойство поиска в глубину заключается в том, что поиск может использоваться для классификации ребер входного графа $G = (V, E)$. Эта классификация ребер может использоваться для получения важной информации о графе. Например, в следующем разделе мы увидим, что ориентированный граф ацикличен тогда и только тогда, когда при поиске в глубину в нем не обнаруживается “обратных” ребер (лемма 22.11).

Мы можем определить четыре типа ребер с использованием леса G_π, полученного при поиске в глубину в графе G.

1. ***Ребра деревьев*** (tree edges) — это ребра графа G_π. Ребро (u, v) является ребром дерева, если при исследовании этого ребра открыта вершина v.
2. ***Обратные ребра*** (back edges) — это ребра (u, v), соединяющие вершину u с ее предком v в дереве поиска в глубину. Ребра-циклы, которые могут встречаться в ориентированных графах, рассматриваются как обратные ребра.
3. ***Прямые ребра*** (forward edges) — это ребра (u, v), не являющиеся ребрами дерева и соединяющие вершину u с ее потомком v в дереве поиска в глубину.
4. ***Перекрестные ребра*** (cross edges) — все остальные ребра графа. Они могут соединять вершины одного и того же дерева поиска в глубину, когда ни одна из вершин не является предком другой, или соединять вершины в разных деревьях.

На рис. 22.4 и рис. 22.5 ребра помечены в соответствии с их типом. На рис. 22.5*в* показан граф на рис. 22.5*а*, нарисованный так, что все прямые ребра и ребра деревьев направлены вниз, а обратные ребра — вверх. Любой граф можно перерисовать таким способом.

Алгоритм DFS можно модифицировать так, что он будет классифицировать встречающиеся при работе ребра. Ключевая идея заключается в том, что каждое ребро (u, v) можно классифицировать при помощи цвета вершины v при первом его исследовании (правда, при этом не различаются прямые и перекрестные ребра).

1. Белый цвет говорит о том, что это ребро дерева.
2. Серый цвет определяет обратное ребро.
3. Черный цвет указывает на прямое или перекрестное ребро.

Первый случай следует непосредственно из определения алгоритма. Рассматривая второй случай, заметим, что серые вершины всегда образуют линейную цепочку потомков, соответствующую стеку активных вызовов процедуры DFS_VISIT; количество серых вершин на единицу больше глубины последней открытой вершины в дереве поиска в глубину. Исследование всегда начинается от самой глубокой серой вершины, так что ребро, которое достигает другой серой вершины, достигает предка исходной вершины. В третьем случае мы имеем дело с остальными ребрами, не подпадающими под первый или второй случай. Можно показать, что ребро (u, v) является прямым, если $d[u] < d[v]$, и перекрестным, если $d[u] > d[v]$ (см. упражнение 22.3-4).

В неориентированном графе при классификации ребер может возникнуть определенная неоднозначность, поскольку (u, v) и (v, u) в действительности являются одним ребром. В таком случае, когда ребро соответствует нескольким

категориям, оно классифицируется в соответствии с *первой* категорией в списке, применимой для данного ребра. Тот же результат получается, если выполнять классификацию в соответствии с тем, в каком именно виде — (u, v) или (v, u) — ребро встречается в первый раз в процессе выполнения алгоритма (см. упражнение 22.3-5).

Теперь мы покажем, что прямые и перекрестные ребра никогда не встречаются при поиске в глубину в неориентированном графе.

Теорема 22.10. При поиске в глубину в неориентированном графе G любое ребро G является либо ребром дерева, либо обратным ребром.

Доказательство. Пусть (u, v) — произвольное ребро графа G, и предположим без потери общности, что $d[u] < d[v]$. Тогда вершина v должна быть открыта и завершена до того, как будет завершена работа с вершиной u (пока u — серая), так как v находится в списке смежности u. Если ребро (u, v) исследуется сначала в направлении от u к v, то v до этого момента была неоткрытой (белой) — так как в противном случае мы бы уже исследовали это ребро в направлении от v к u. Таким образом, (u, v) становится ребром дерева. Если же ребро (u, v) исследуется сначала в направлении от v к u, то оно является обратным, поскольку вершина u при первом исследовании ребра — серая. ■

Мы ознакомимся с некоторыми применениями этой теоремы в последующих разделах.

Упражнения

22.3-1. Нарисуйте таблицу размером 3×3, со строками и столбцами, помеченными как WHITE (белый), GRAY (серый) и BLACK (черный). В каждой ячейке (i, j) пометьте, может ли быть обнаружено в процессе поиска в глубину в ориентированном графе ребро из вершины цвета i в вершину цвета j. Для каждого возможного ребра укажите его возможный тип. Постройте такую же таблицу для неориентированного графа.

22.3-2. Покажите, как работает поиск в глубину для графа, изображенного на рис. 22.6. Считаем, что цикл **for** в строках 5–7 процедуры DFS сканирует вершины в алфавитном порядке, а также что все списки смежности упорядочены по алфавиту. Для каждой вершины укажите время открытия и завершения, а для каждого ребра — его тип.

22.3-3. Напишите скобочное выражение, соответствующее поиску в глубину, показанному на рис. 22.4.

22.3-4. Покажите, что ребро (u, v) является

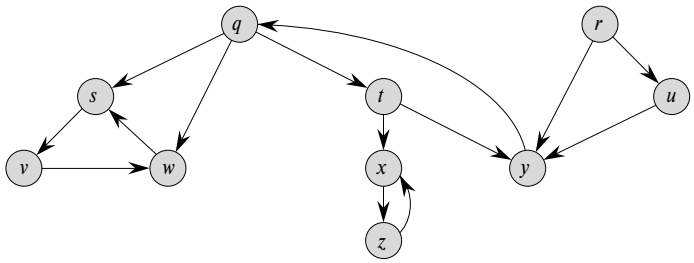


Рис. 22.6. Ориентированный граф для упражнений 22.3-2 и 22.5-2

а) ребром дерева или прямым ребром тогда и только тогда, когда $d[u] < d[v] < f[v] < f[u]$;

б) обратным ребром тогда и только тогда, когда $d[v] \leqslant d[u] < f[u] \leqslant$ $\leqslant f[v]$;

в) перекрестным ребром тогда и только тогда, когда $d[v] < f[v] <$ $< d[u] < f[u]$.

22.3-5. Покажите, что в неориентированном графе классификация ребра (u, v) как ребра дерева или обратного ребра в зависимости от того, встречается ли первым ребро (u, v) или (v, u) при поиске в глубину, эквивалентна классификации в соответствии с приоритетами типов в схеме классификации.

22.3-6. Перепишите процедуру DFS с использованием стека для устранения рекурсии.

22.3-7. Приведите контрпример к гипотезе, заключающейся в том, что если в ориентированном графе G имеется путь от u к v и что если $d[u] < d[v]$ при поиске в глубину в G, то v — потомок u в лесу поиска в глубину.

22.3-8. Приведите контрпример к гипотезе, заключающейся в том, что если в ориентированном графе G имеется путь от u к v, то любой поиск в глубину должен дать в результате $d[v] \leqslant f[u]$.

22.3-9. Модифицируйте псевдокод поиска в глубину так, чтобы он выводил все ребра ориентированного графа G вместе с их типами. Какие изменения следует внести в псевдокод (если таковые требуются) для работы с неориентированным графом?

22.3-10. Объясните, как вершина ориентированного графа может оказаться единственной в дереве поиска в глубину, несмотря на наличие у нее как входящих, так и исходящих ребер.

22.3-11. Покажите, что поиск в глубину в неориентированном графе G может использоваться для определения связных компонентов графа G и что

лес поиска в глубину содержит столько деревьев, сколько в графе связных компонентов. Говоря более точно, покажите, как изменить поиск в глубину так, чтобы каждой вершине v присваивалась целая метка $cc[v]$ в диапазоне от 1 до k (где k — количество связных компонентов), такая что $cc[u] = cc[v]$ тогда и только тогда, когда u и v принадлежат одному связному компоненту.

⋆ 22.3-12. Ориентированный граф $G = (V, E)$ обладает свойством ***односвязности*** (singly connected), если из $u \rightsquigarrow v$ следует, что имеется не более одного пути от u к v для всех вершин $u, v \in V$. Разработайте эффективный алгоритм для определения, является ли ориентированный граф односвязным.

22.4 Топологическая сортировка

В этом разделе показано, каким образом можно использовать поиск в глубину для топологической сортировки ориентированного ациклического графа (directed acyclic graph, для которого иногда используется аббревиатура "dag"). ***Топологическая сортировка*** (topological sort) ориентированного ациклического графа $G = (V, E)$ представляет собой такое линейное упорядочение всех его вершин, что если граф G содержит ребро (u, v), то u при таком упорядочении располагается до v (если граф не является ацикличным, такая сортировка невозможна). Топологическую сортировку графа можно рассматривать как такое упорядочение его вершин вдоль горизонтальной линии, что все ребра направлены слева направо. Таким образом, топологическая сортировка существенно отличается от обычных видов сортировки, рассмотренных в части III.

Ориентированные ациклические графы используются во многих приложениях для указания последовательности событий. На рис. 22.7 приведен пример графа, построенного профессором Рассеянным для утреннего одевания. Некоторые вещи надо обязательно одевать раньше других, например, сначала носки, а затем туфли. Другие вещи могут быть одеты в произвольном порядке (например, носки и рубашка). Ребро (u, v) в ориентированном ациклическом графе на рис. 22.7*а* показывает, что вещь u должна быть одета раньше вещи v. Топологическая сортировка этого графа дает нам порядок одевания. На рис. 22.7*б* показан отсортированный ориентированный ациклический граф, вершины которого расположены вдоль горизонтальной линии так, что все ребра направлены слева направо.

Вот простой алгоритм топологической сортировки ориентированного ациклического графа:

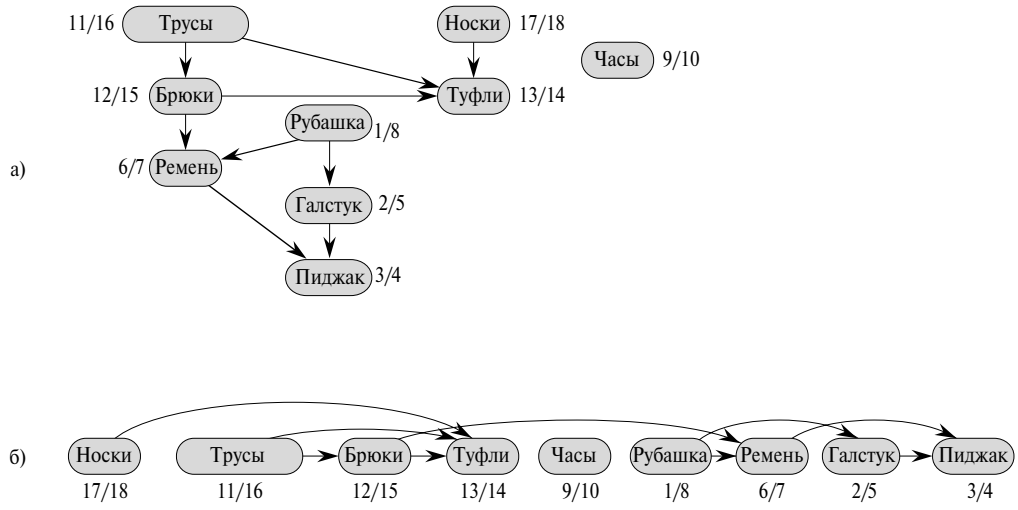


Рис. 22.7. Топологическая сортировка одежды

TOPOLOGICAL_SORT(G)

1 Вызов DFS(G) для вычисления времени завершения $f[v]$
для каждой вершины v
2 По завершении работы над вершиной внести ее в начало связанного списка
3 **return** Связанный список вершин

На рис. 22.7*б* видно, что топологически отсортированные вершины располагаются в порядке убывания времени завершения.

Мы можем выполнить топологическую сортировку за время $\Theta(V+E)$, поскольку поиск в глубину выполняется именно за это время, а вставка каждой из $|V|$ вершин в начало связанного списка занимает время $O(1)$.

Докажем корректность этого алгоритма с использованием следующей ключевой леммы, характеризующей ориентированный ациклический граф.

Лемма 22.11. Ориентированный граф G является ациклическим тогда и только тогда, когда поиск в глубину в G не находит в нем обратных ребер.

Доказательство. $\Rightarrow$: Предположим, что имеется обратное ребро (u,v). Тогда вершина v является предком u в лесу поиска в глубину. Таким образом, в графе G имеется путь из v в u, и ребро (u,v) завершает цикл.

$\Leftarrow$: Предположим, что граф G содержит цикл c. Покажем, что поиск в глубину обнаружит обратное ребро. Пусть v — первая вершина, открытая в c, и пусть (u,v) — предыдущее ребро в c. В момент времени $d[v]$ вершины c образуют путь из белых вершин от v к u. В соответствии с теоремой о белом пути, вершина u

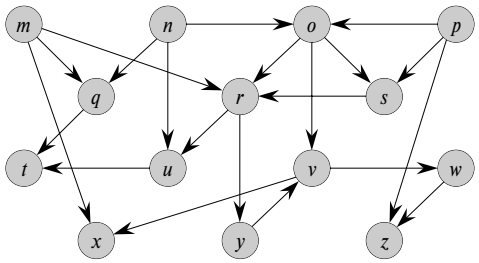


Рис. 22.8. Ориентированный ациклический граф для топологической сортировки

становится потомком v в лесу поиска в глубину. Следовательно, (u, v) — обратное ребро. ■

Теорема 22.12. Процедура TOPOLOGICAL_SORT(G) выполняет топологическую сортировку ориентированного ациклического графа G.

Доказательство. Предположим, что над данным ориентированным ациклическим графом $G = (V, E)$ выполняется процедура DFS, которая вычисляет время завершения его вершин. Достаточно показать, что если для произвольной пары разных вершин $u, v \in V$ в графе G имеется ребро от u к v, то $f[v] < f[u]$. Рассмотрим произвольное ребро (u, v), исследуемое процедурой DFS(G). При исследовании вершина v не может быть серой, поскольку тогда v была бы предком u и (u, v) представляло бы собой обратное ребро, что противоречит лемме 22.11. Следовательно, вершина v должна быть белой либо черной. Если вершина v — белая, то она становится потомком u, так что $f[v] < f[u]$. Если v — черная, значит, работа с ней уже завершена и значение $f[v]$ уже установлено. Поскольку мы все еще работаем с вершиной u, значение $f[u]$ еще не определено, так что, когда это будет сделано, будет выполняться неравенство $f[v] < f[u]$. Следовательно, для любого ребра (u, v) ориентированного ациклического графа выполняется условие $f[v] < f[u]$, что и доказывает данную теорему. ■

Упражнения

22.4-1. Покажите, в каком порядке расположит вершины представленного на рис. 22.8 ориентированного ациклического графа процедура TOPOLOGICAL_SORT, если считать, что цикл **for** в строках 5–7 процедуры DFS сканирует вершины в алфавитном порядке, а также что все списки смежности упорядочены по алфавиту.

22.4-2. Разработайте алгоритм с линейным временем работы, который для данного ориентированного ациклического графа $G = (V, E)$ и двух вершин

s и t определяет количество путей из s к t в графе G. Например, в графе на рис. 22.8 имеется ровно 4 пути от вершины p к вершине v: pov, $poryv$, $posryv$ и $psryv$. Ваш алгоритм должен только указать количество путей, не перечисляя их.

22.4-3. Разработайте алгоритм для определения, содержит ориентированный граф $G = (V, E)$ цикл или нет. Ваш алгоритм должен выполняться за время $O(V)$, независимо от $|E|$.

22.4-4. Докажите или опровергните следующее утверждение: если ориентированный граф G содержит циклы, то процедура TOPOLOGICAL_SORT(G) упорядочивает вершины таким образом, что при этом количество "плохих" ребер (идущих в противоположном направлении) минимально.

22.4-5. Еще один алгоритм топологической сортировки ориентированного ациклического графа $G = (V, E)$ состоит в поиске вершины со входящей степенью 0, выводе ее, удалении из графа этой вершины и всех исходящих из нее ребер, и повторении этих действий до тех пор, пока в графе остается хоть одна вершина. Покажите, как реализовать описанный алгоритм, чтобы время его работы составляло $O(V + E)$. Что произойдет, если в графе G будут иметься циклы?

22.5 Сильно связные компоненты

Теперь мы рассмотрим классическое применение поиска в глубину: разложение ориентированного графа на сильно связные компоненты. В этом разделе будет показано, как выполнить такое разложение при помощи двух поисков в глубину. Ряд алгоритмов для работы с графами начинаются с выполнение такого разложения графа, и после разложения алгоритм отдельно работает с каждым сильно связным компонентом. Полученные решения затем комбинируются в соответствии со структурой связей между сильно связными компонентами.

В приложении Б сказано, что сильно связный компонент ориентированного графа $G = (V, E)$ представляет собой максимальное множество вершин $C \subseteq V$, такое что для каждой пары вершин u и v из C справедливо как $u \rightsquigarrow v$, так и $v \rightsquigarrow u$, т.е. вершины u и v достижимы друг из друга. На рис. 22.9 приведены примеры сильно связных компонентов.

Наш алгоритм поиска сильно связных компонентов графа $G = (V, E)$ использует транспонирование G, которое определяется в упражнении 22.1-3 как граф $G^{\mathrm{T}} = \left(V, E^{\mathrm{T}}\right)$, где $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$, т.е. E^{T} состоит из ребер G путем изменения их направления на обратное. Для представления с помощью списков смежности данного графа G получить граф G^{T} можно за время $O(V + E)$. Интересно заметить, что графы G и G^{T} имеют одни и те же сильно связные компоненты: u и v достижимы друг из друга в G тогда и только тогда, когда они

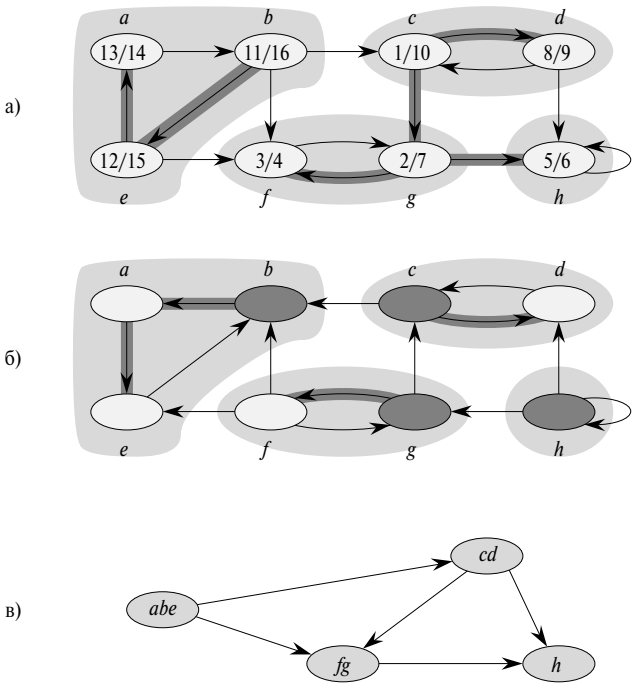


Рис. 22.9. Сильно связные компоненты ориентированного графа

достижимы друг из друга в G^{T}. На рис. 22.9*б* показан граф, представляющий собой результат транспонирования графа на рис. 22.9*а* (сильно связные компоненты выделены штриховкой).

Далее приведен алгоритм, который за линейное время $\Theta(V+E)$ находит сильно связные компоненты ориентированного графа $G=(V,E)$ благодаря двойному поиску в глубину, одному — в графе G, и второму — в графе G^{T}.

STRONGLY_CONNECTED_COMPONENTS(G)

1 Вызов DFS(G) для вычисления времен завершения $f[u]$
  для каждой вершины u
2 Построение G^{T}
3 Вызов DFS(G^{T}), но в главном цикле процедуры DFS,
  вершины рассматриваются в порядке убывания значений $f[u]$
  вычисленных в строке 1
4 Деревья леса поиска в глубину, полученного в строке 3
  представляют собой сильно связные компоненты

Идея, лежащая в основе этого алгоритма, опирается на ключевое свойство ***графа компонентов*** (component graph) $G^{\mathrm{SCC}}=\left(V^{\mathrm{SCC}},E^{\mathrm{SCC}}\right)$, который определяется

следующим образом. Предположим, что G имеет сильно связные компоненты $C_1, C_2, \ldots, C_k$. Множество вершин $V^{\mathrm{SCC}} = \{v_1, v_2, \ldots, v_k\}$ состоит из вершин v_i для каждого сильно связного компонента C_i графа G. Если в G имеется ребро (x, y) для некоторых двух вершин $x \in C_i$ и $y \in C_j$, то в графе компонент имеется ребро $(v_i, v_j) \in E^{\mathrm{SCC}}$. Другими словами, если сжать все ребра между смежными вершинами в каждом сильно связном компоненте графа G, мы получим граф G^{SCC} (вершинами которого являются сильно связные компоненты графа G). На рис. 22.9*в* показан граф компонентов для графа, приведенного на рис. 22.9*а*.

Ключевое свойство графа компонентов заключается в том, что он представляет собой ориентированный ациклический граф, как следует из приведенной ниже леммы.

Лемма 22.13. Пусть C и C' — различные сильно связные компоненты в ориентированном графе $G = (V, E)$, и пусть $u, v \in C$ и $u', v' \in C'$, а кроме того, предположим, что в G имеется путь $u \rightsquigarrow u'$. В таком случае в G не может быть пути $v' \rightsquigarrow v$.

Доказательство. Если в G имеется путь $v' \rightsquigarrow v$, то в G имеются и пути $u \rightsquigarrow u' \rightsquigarrow v'$ и $v' \rightsquigarrow v \rightsquigarrow u$. Следовательно, вершины u и v' достижимы одна из другой, что противоречит предположению о том, что C и C' — различные сильно связные компоненты. ■

Мы увидим, что при рассмотрении вершин в процессе второго поиска в глубину в порядке убывания времен завершения работы с вершинами, которые были вычислены при первом поиске в глубину, мы по сути посещаем вершины графа компонентов (каждая из которых соответствует сильно связному компоненту G) в порядке топологической сортировки.

Поскольку процедура STRONGLY_CONNECTED_COMPONENTS выполняет два поиска в глубину, имеется потенциальная неоднозначность при рассмотрении значений $d[u]$ и $f[u]$. В этом разделе данные значения будут относиться исключительно к времени открытия и времени завершения, вычисленным при первом вызове процедуры DFS в строке 1.

Мы распространим обозначения для времени открытия и времени завершения на множества вершин. Если $U \subseteq V$, то мы определим $d(U) = \min_{u \in U}\{d[u]\}$ и $f(U) = \max_{u \in U}\{f[u]\}$, т.е. $d(U)$ и $f(U)$ представляют собой самое раннее время открытия и самое позднее время завершения для всех вершин в U.

Следующая лемма и следствие из нее описывают ключевое свойство, связывающее сильно связные компоненты и времена завершения, полученные при первом поиске в глубину.

Лемма 22.14. Пусть C и C' — различные сильно связные компоненты в ориентированном графе $G = (V, E)$. Предположим, что имеется ребро $(u, v) \in E$, где $u \in C$ и $v \in C'$. Тогда $f(C) > f(C')$.

Доказательство. Имеется два возможных случая в зависимости от того, какой из сильно связных компонентов, C или C', содержит первую открытую в процессе поиска в глубину вершину.

Если $d(C) < d(C')$, то обозначим первую открытую в C вершину как x. В момент времени $d[x]$ все вершины в C и C' — белые. В G имеется путь от x к каждой вершине в C, состоящий только из белых вершин. Поскольку $(u, v) \in E$, для любой вершины $w \in C'$ в момент времени $d[x]$ в графе G имеется также путь от x к w, состоящий только из белых вершин: $x \rightsquigarrow u \to v \rightsquigarrow w$. Согласно теореме о белом пути, все вершины в C и C' становятся потомками x в дереве поиска в глубину. Согласно следствию 22.8, $f[x] = f(C) > f(C')$.

Если же $d(C) > d(C')$, то обозначим первую открытую вершину в C' как y. В момент $d[y]$ все вершины в C' белые, и в G имеется путь от y к каждой вершине C', состоящий только из белых вершин. В соответствии с теоремой о белом пути, все вершины в C' становятся потомками y в дереве поиска в глубину, так что согласно следствию 22.8, $f[y] = f(C')$. В момент времени $d[y]$ все вершины в C белые. Поскольку имеется ребро (u, v) из C в C', из леммы 22.13 вытекает, что не существует пути из C' в C. Следовательно, в C не имеется вершин, достижимых из y. Таким образом, в момент времени $f[y]$ все вершины в C остаются белыми. Значит, для любой вершины $w \in C$ имеем $f[w] > f[y]$, откуда следует, что $f(C) > f(C')$. ■

Приведенное ниже следствие говорит нам, что каждое ребро в G^{T}, соединяющее два сильно связных компонента, идет от компонента с более ранним временем завершения (при поиске в глубину) к компоненту с более поздним временем завершения.

Следствие 22.15. Пусть C и C' — различные сильно связные компоненты в ориентированном графе $G = (V, E)$. Предположим, что имеется ребро $(u, v) \in E^{\mathrm{T}}$, где $u \in C$ и $v \in C'$. Тогда $f(C) < f(C')$.

Доказательство. Поскольку $(u, v) \in E^{\mathrm{T}}$, $(v, u) \in E$. Так как сильно связные компоненты G и G^{T} одни и те же, из леммы 22.14 следует, что $f(C) < f(C')$. ■

Следствие 22.15 дает нам ключ к пониманию того, как работает процедура STRONGLY_CONNECTED_COMPONENTS. Рассмотрим, что происходит, когда мы выполняем второй поиск в глубину над графом G^{T}. Мы начинаем с сильно связного компонента C, время завершения которого $f(C)$ максимально. Поиск начинается с некоторой вершины $x \in C$, при этом посещаются все вершины в C. Согласно

следствию 22.15, в G^{T} нет ребер от C к другому сильно связному компоненту, так что при поиске из x не посещается ни одна вершина в других компонентах. Следовательно, дерево, корнем которого является x, содержит только вершины из C. После того как будут посещены все вершины в C, поиск в строке 3 выбирает в качестве корня вершину из некоторого другого сильно связного компонента C', время завершения $f(C')$ которого максимально среди всех компонентов, исключая C. Теперь поиск посещает все вершины в C'. Согласно следствию 22.15, в G^{T} единственным ребром из C' в другие компоненты может быть ребро в C, но обработка компонента C уже завершена. В общем случае, когда поиск в глубину в G^{T} в строке 3 посещает некоторый сильно связный компонент, все ребра, исходящие из этого компонента, идут в уже обработанные компоненты. Следовательно, каждый поиск в глубину обрабатывает ровно один сильно связный компонент. Приведенная далее теорема формализует это доказательство.

Теорема 22.16. Процедура STRONGLY_CONNECTED_COMPONENTS(G) корректно вычисляет сильно связные компоненты ориентированного графа G.

Доказательство. Воспользуемся индукцией по количеству найденных деревьев при поиске в глубину в G^{T} в строке 3, и докажем, что вершины каждого дерева образуют сильно связный компонент. Гипотеза индукции состоит в том, что первые k деревьев, полученные в строке 3, являются сильно связными компонентами. Для случая $k = 0$ это утверждение тривиально.

Для выполнения шага индукции предположим, что каждое из первых k деревьев поиска в глубину в строке 3, представляет собой сильно связный компонент, и рассмотрим $(k+1)$-е дерево. Пусть корнем этого дерева является вершина u, и пусть u принадлежит сильно связному компоненту C. В соответствии с тем как мы выбираем корни при поиске в глубину в строке 3, для любого сильно связного компонента C', который еще не был посещен и отличен от C, справедливо соотношение $f[u] = f(C) > f(C')$. В соответствии с гипотезой индукции в момент времени, когда поиск посещает вершину u, все остальные вершины C — белые. Согласно теореме о белом пути, все вершины C, кроме u, являются потомками u в дереве поиска в глубину. Кроме того, в соответствии с гипотезой индукции и со следствием 22.15, все ребра в G^{T}, которые покидают C, должны идти в уже посещенные сильно связные компоненты. Таким образом, ни в одном сильно связном компоненте, отличном от C, нет вершины, которая бы стала потомком u в процессе поиска в глубину в G^{T}. Следовательно, вершины дерева поиска в глубину в G^{T}, корнем которого является u, образуют ровно один сильно связный компонент, что и завершает шаг индукции и доказательство данной теоремы. ■

Вот еще одна точка зрения на работу второго поиска в глубину. Рассмотрим граф компонентов $(G^{\mathrm{T}})^{\mathrm{SCC}}$ графа G^{T}. Если мы отобразим каждый сильно связный компонент, посещенный при втором поиске в глубину, на вершину $(G^{\mathrm{T}})^{\mathrm{SCC}}$,

то вершины этого графа компонентов посещаются в порядке, обратном топологической сортировке. Если мы обратим все ребра графа $(G^{\mathrm{T}})^{\mathrm{SCC}}$, то получим граф $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}}$. Так как $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$ (см. упражнение 22.5-4), при втором поиске в глубину вершины G^{SCC} посещаются в порядке топологической сортировки.

Упражнения

22.5-1. Как может измениться количество сильно связных компонентов графа при добавлении в граф нового ребра?

22.5-2. Рассмотрите работу процедуры STRONGLY_CONNECTED_COMPONENTS над графом, показанным на рис. 22.6. В частности, определите время завершения, вычисляемое в строке 1, и лес, полученный в строке 3. Считаем, что цикл в строках 5–7 процедуры DFS рассматривает вершины в алфавитном порядке и что так же упорядочены и списки смежности.

22.5-3. Профессор считает, что алгоритм определения сильно связных компонентов можно упростить, если во втором поиске в глубину использовать исходный, а не транспонированный граф, и сканировать вершины в порядке *возрастания* времени завершения. Прав ли профессор?

22.5-4. Докажите, что для любого ориентированного графа G справедливо соотношение $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$, т.е. что транспонирование графа компонентов G^{T} дает граф компонентов графа G.

22.5-5. Разработайте алгоритм, который за время $O(V + E)$ находит граф компонентов ориентированного графа $G = (V, E)$. Убедитесь, что в полученном графе компонентов между двумя вершинами имеется не более одного ребра.

22.5-6. Поясните, как для данного ориентированного графа $G = (V, E)$ создать другой граф $G' = (V, E')$, такой что *а*) G' имеет те же сильно связные компоненты, что и G, *б*) G' имеет тот же граф компонентов, что и G, и *в*) E' имеет минимально возможный размер. Разработайте быстрый алгоритм для вычисления G'.

22.5-7. Ориентированный граф $G = (V, E)$ называется ***полусвязным*** (semiconnected), если для всех пар вершин $u, v \in V$ $u \rightsquigarrow v$ или $v \rightsquigarrow u$ (или и то, и другое одновременно). Разработайте эффективный алгоритм для определения, является ли данный граф G полусвязным. Докажите корректность разработанного алгоритма и проанализируйте время его работы.

Задачи

22-1. **Классификация ребер при поиске в ширину**

Лес поиска в глубину позволяет классифицировать ребра графа как ребра деревьев, обратные, прямые и перекрестные. Дерево поиска в ширину также можно использовать для аналогичной классификации ребер, достижимых из исходной вершины.

а) Докажите, что при поиске в ширину в неориентированном графе выполняются следующие свойства.

1) Не существует прямых и обратных ребер.
2) Для каждого ребра дерева (u, v) имеем $d[v] = d[u] + 1$.
3) Для каждого перекрестного ребра (u, v) имеем $d[v] = d[u]$ или $d[v] = d[u] + 1$.

б) Докажите, что при поиске в ширину в ориентированном графе выполняются следующие свойства.

1) Не существует прямых ребер.
2) Для каждого ребра дерева (u, v) имеем $d[v] = d[u] + 1$.
3) Для каждого перекрестного ребра (u, v) имеем $d[v] \leqslant d[u] + 1$.
4) Для каждого обратного ребра (u, v) имеем $0 \leqslant d[v] \leqslant d[u]$.

22-2. **Точки сочленения, мосты и двусвязные компоненты**

Пусть $G = (V, E)$ — связный неориентированный граф. ***Точкой сочленения*** (articulation point) G называется вершина, удаление которой делает граф несвязным. ***Мостом*** (bridge) графа G называется ребро, удаление которого делает граф несвязным. ***Двусвязный компонент*** (biconnected component) графа G представляет собой максимальное множество ребер, такое что любые два ребра этого множества принадлежат общему простому циклу. На рис. 22.10 проиллюстрированы приведенные определения. Темным цветом на рисунке выделены точки сочленения и мосты, двусвязные компоненты — наборы ребер в пределах одной серой области (внутри которой указан номер двусвязного компонента, о котором идет речь в задании *з*) данной задачи). Точки сочленения, мосты и двусвязные компоненты можно найти при помощи поиска в глубину. Пусть $G_\pi = (V, E_\pi)$ — дерево поиска в глубину графа G.

а) Докажите, что корень G_π — точка сочленения графа G тогда и только тогда, когда он имеет как минимум два дочерних узла в G_π.

б) Пусть v — некорневая вершина G_π. Докажите, что v является точкой сочленения G тогда и только тогда, когда v имеет потомка s, такого что не существует обратного ребра от s или любого его потомка к истинному предку v.

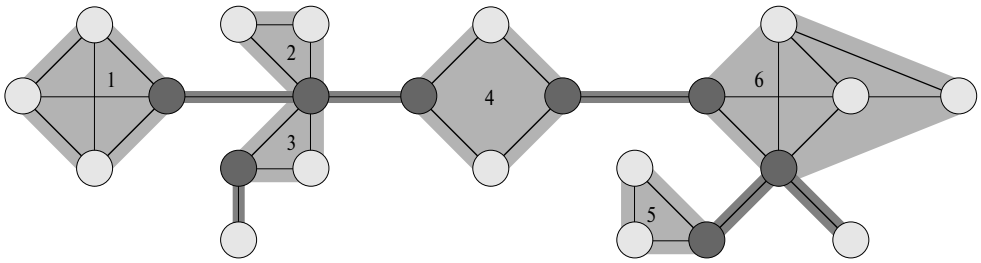


Рис. 22.10. Точки сочленения, мосты и двусвязные компоненты связного неориентированного графа

в) Пусть $low\,[v]$ — минимальное значение среди $d\,[v]$ и всех $d\,[w]$, где w — вершины, для которых имеется обратное ребро (u, w), где u — некоторый потомок v. Покажите, как можно вычислить $low\,[v]$ для всех вершин $v \in V$ за время $O\,(E)$.

г) Покажите, как найти все точки сочленения за время $O\,(E)$.

д) Докажите, что ребро в G является мостом тогда и только тогда, когда оно не принадлежит ни одному простому циклу G.

е) Покажите, как найти все мосты за время $O\,(E)$.

ж) Докажите, что двусвязные компоненты графа G составляют разбиение множества всех ребер графа, не являющихся мостами.

з) Разработайте алгоритм, который за время $O\,(E)$ помечает каждое ребро e графа G натуральным числом $bcc\,[e]$, таким что $bcc\,[e] = bcc\,[e']$ тогда и только тогда, когда e и e' находятся в одном и том же двусвязном компоненте.

22-3. **Эйлеров цикл**

Эйлеров цикл (Euler tour) сильно связного ориентированного графа $G = (V, E)$ представляет собой цикл, который проходит по всем ребрам G ровно по одному разу, хотя через вершины он может проходить по несколько раз.

а) Покажите, что в G имеется Эйлеров цикл тогда и только тогда, когда входящая степень каждой вершины равна ее исходящей степени.

б) Разработайте алгоритм, который за время $O\,(E)$ находит Эйлеров цикл графа G (если таковой цикл существует). (*Указание*: объединяйте циклы, у которых нет общих ребер.)

22-4. **Достижимость**

Пусть $G = (V, E)$ — ориентированный граф, в котором каждая вершина $u \in V$ помечена уникальным целым числом $L\,(u)$ из множества

$\{1, 2, \ldots, |V|\}$. Для каждой вершины $u \in V$ рассмотрим множество $R(u) = \{v \in V : u \rightsquigarrow v\}$ вершин, достижимых из u. Определим $\min(u)$ как вершину в $R(u)$, метка которой минимальна, т.е. $\min(u)$ — это такая вершина v, что $L(v) = \min\{L(w) : w \in R(u)\}$. Разработайте алгоритм, который за время $O(V + E)$ вычисляет $\min(u)$ для всех вершин $u \in V$.

Заключительные замечания

Превосходные руководства по алгоритмам для работы с графами написаны Ивеном (Even) [87] и Таржаном (Tarjan) [292].

Поиск в ширину был открыт Муром (Moore) [226] в контексте задачи поиска пути через лабиринт. Ли (Lee) [198] независимо открыл тот же алгоритм при работе над разводкой печатных плат.

Хопкрофт (Hopcroft) и Таржан [154] указали на преимущества использования представления графов в виде списков смежности над матричным представлением для разреженных графов, и были первыми, кто оценил алгоритмическую важность поиска в глубину. Поиск в глубину широко используется с конца 1950-х годов, в особенности в программах искусственного интеллекта.

Таржан [289] разработал алгоритм поиска сильно связных компонентов за линейное время. Алгоритм из раздела 22.5 взят у Ахо (Aho), Хопкрофта и Ульмана (Ullman) [6], которые ссылаются на неопубликованную работу Косараю (S.R. Kosaraju) и работу Шарира (Sharir) [276]. Габов (Gabow) [101] разработал алгоритм для поиска сильно связных компонентов, который основан на сжатых циклах и использует два стека для обеспечения линейного времени работы. Кнут (Knuth) [182] первым разработал алгоритм топологической сортировки за линейное время.

ГЛАВА 23

Минимальные остовные деревья

При разработке электронных схем зачастую необходимо электрически соединить контакты нескольких компонентов. Для соединения множества из n контактов мы можем использовать некоторую компоновку из $n-1$ проводов, каждый из которых соединяет два контакта. Обычно желательно получить компоновку, которая использует минимальное количество провода.

Мы можем смоделировать эту задачу при помощи связного неориентированного графа $G=(V,E)$, где V — множество контактов, E — множество возможных соединений между парами контактов, и для каждого ребра $(u,v)\in E$ задан вес $w(u,v)$, определяющий стоимость (количество необходимого провода) соединения u и v. Мы хотим найти ациклическое подмножество $T\subseteq E$, которое соединяет все вершины и чей общий вес

$$w(T)=\sum_{(u,v)\in T} w(u,v)$$

минимален. Поскольку множество T ациклическое и связывает все вершины, оно должно образовывать дерево, которое мы назовем ***остовным деревом*** (spanning tree) графа G (иногда используется термин "покрывающее дерево"). Задачу поиска дерева T мы назовем ***задачей поиска минимального остовного дерева*** (minimum-spanning-tree problem)[1]. На рис. 23.1 показан пример связного графа и его минимального остовного дерева. На ребрах указан их вес, а ребра минимального остовного дерева отдельно выделены цветом. Общий вес показанного дерева равен 37.

[1]По сути, термин "минимальное остовное дерево" означает "остовное дерево с минимальным весом". Мы не минимизируем, например, количество ребер в T, поскольку все остовные деревья имеют ровно $|V|-1$ ребер согласно теореме Б.2.

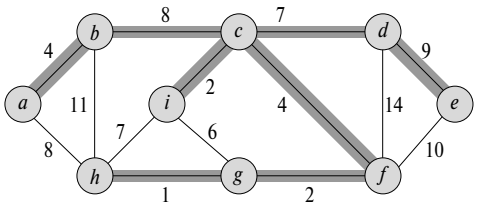


Рис. 23.1. Минимальное остовное дерево связного графа

Приведенное дерево не единственное: удалив ребро (b, c) и заменив его ребром (a, h), мы получим другое остовное дерево с тем же весом 37.

В этой главе мы рассмотрим два алгоритма решения задачи поиска минимального остовного дерева — алгоритм Крускала (Kruskal) и Прима (Prim). Каждый из них легко реализовать с помощью обычных бинарных пирамид, получив время работы $O(E \lg V)$. При использовании фибоначчиевых пирамид алгоритм Прима можно ускорить до $O(E + V \lg V)$, что является весьма существенным ускорением при $|V| \ll |E|$.

Оба эти алгоритма — жадные (см. главу 16). На каждом шаге алгоритма мы выбираем один из возможных вариантов. Жадная стратегия предполагает выбор варианта, наилучшего в данный момент. В общем случае такая стратегия не гарантирует глобально оптимального решения задачи, однако для задачи поиска минимального остовного дерева можно доказать, что определенные жадные стратегии дают нам остовное дерево минимального веса. Хотя настоящую главу можно читать независимо от главы 16, жадные алгоритмы, представленные здесь, наглядно демонстрируют классическое применение изложенных в этой главе теоретических основ.

В разделе 23.1 описана общая схема построения минимального остовного дерева, которая наращивает остовное дерево по одному ребру. В разделе 23.2 приведены два пути реализации обобщенного алгоритма. Первый алгоритм (Крускала) похож на алгоритм поиска связных компонентов из раздела 21.1. Второй алгоритм (Прима) подобен алгоритму поиска кратчайшего пути Дейкстры (Dijkstra) из раздела 24.3.

23.1 Построение минимального остовного дерева

Предположим, что у нас есть связный неориентированный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbf{R}$, и мы хотим найти минимальное остовное дерево

для G. В этой главе мы рассмотрим два жадных алгоритма решения поставленной задачи.

Оба рассматриваемых алгоритма имеют общую схему, согласно которой минимальное остовное дерево растет путем добавления к нему ребер по одному. Алгоритм работает с множеством ребер A, и инвариант цикла алгоритма выглядит следующим образом:

> Перед каждой очередной итерацией A представляет собой подмножество некоторого минимального остовного дерева.

На каждом шаге алгоритма мы определяем ребро (u, v), которое можно добавить к A без нарушения этого инварианта, в том смысле, что $A \cup \{(u, v)\}$ также является подмножеством минимального остовного дерева. Мы назовем такое ребро ***безопасным*** (safe edge) для A, поскольку его можно добавить к A, не опасаясь нарушить инвариант.

```
GENERIC_MST(G, w)
1  A ← ∅
2  while A не является минимальным остовным деревом
3        do Найти безопасное для A ребро (u, v)
4           A ← A ∪ {(u, v)}
5  return A
```

Мы используем инвариант цикла следующим образом.

Инициализация. После выполнения строки 1 множество A тривиально удовлетворяет инварианту цикла.

Сохранение. Цикл в строках 2–4 сохраняет инвариант путем добавления только безопасных ребер.

Завершение. Все ребра, добавленные в A, находятся в минимальном остовном дереве, так что множество A, возвращаемое в строке 5, должно быть минимальным остовным деревом.

Самое важное, само собой разумеется, заключается в том, как именно найти безопасное ребро в строке 3. Оно должно существовать, поскольку когда выполняется строка 3, инвариант требует, чтобы существовало такое остовное дерево T, что $A \subseteq T$. Внутри тела цикла **while** A должно быть истинным подмножеством T, поэтому должно существовать ребро $(u, v) \in T$, такое что $(u, v) \notin A$ и (u, v) — безопасное для A ребро.

В оставшейся части этого раздела мы приведем правило (теорема 23.1) для распознавания безопасных ребер. В следующем разделе описаны два алгоритма, которые используют это правило для эффективного поиска безопасных ребер.

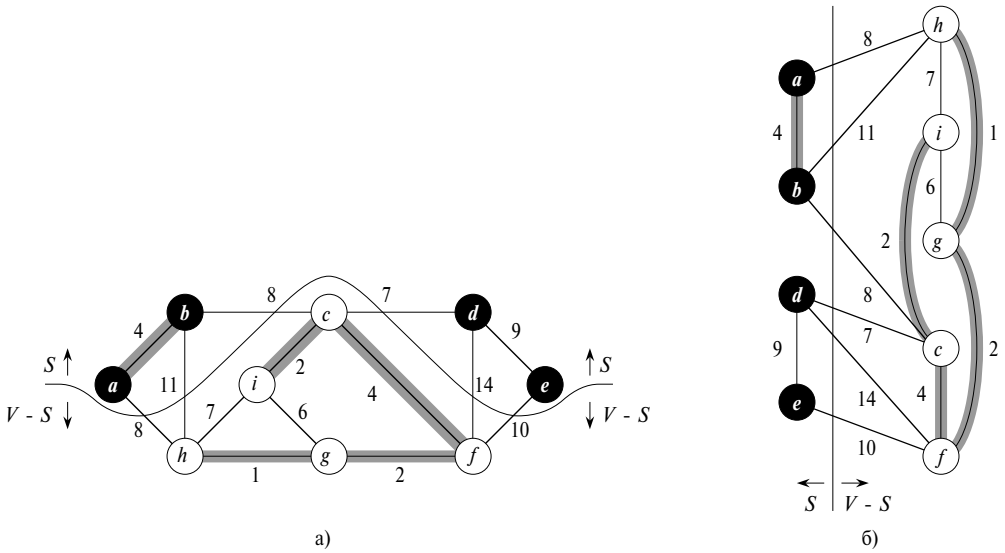


Рис. 23.2. Два варианта представления разреза $(S, V - S)$ графа, приведенного на рис. 23.1

Начнем с определений. ***Разрезом*** (cut) $(S, V - S)$ неориентированного графа $G = (V, E)$ называется разбиение V, что проиллюстрировано на рис. 23.2 (вершины в множестве S окрашены в черный цвет, а в множестве $V - S$ — в белый). Мы говорим, что ребро $(u, v) \in E$ ***пересекает*** (crosses) разрез $(S, V - S)$, если один из его концов оказывается в множестве S, а второй — в $V - S$. Разрез ***согласован*** (respect) с множеством A по ребрам, если ни одно ребро из A не пересекает разрез. Ребро, пересекающее разрез, является ***легким*** (light), если оно имеет минимальный вес среди всех ребер, пересекающих разрез. Заметим, что может быть несколько легких ребер одновременно. В общем случае мы говорим, что ребро является легким ребром, удовлетворяющим некоторому свойству, если оно имеет минимальный вес среди всех ребер, удовлетворяющих данному свойству.

Наше правило для распознавания безопасных ребер дает следующая теорема.

Теорема 23.1. Пусть $G = (V, E)$ — связный неориентированный граф с действительной весовой функцией w, определенной на E. Пусть A — подмножество E, которое входит в некоторое минимальное остовное дерево G, $(S, V - S)$ — разрез G, согласованный с A по ребрам, а (u, v) — легкое ребро, пересекающее разрез $(S, V - S)$. Тогда ребро (u, v) является безопасным для A.

Доказательство. Пусть T — минимальное остовное дерево, которое включает A, и предположим, что T не содержит ребро (u, v), поскольку в противном случае теорема доказана. Мы построим другое минимальное остовное дерево T',

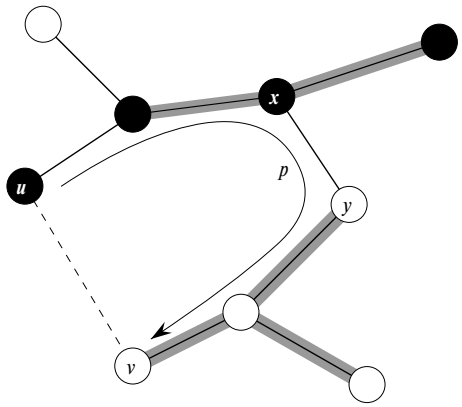


Рис. 23.3. Иллюстрация к доказательству теоремы 23.1

которое включает $A \cup \{(u, v)\}$, путем использования метода вырезания и вставки, показывая таким образом, что ребро (u, v) является безопасным для A.

Ребро (u, v) образует цикл с ребрами на пути p от u к v в T, как показано на рис. 23.3. Поскольку u и v находятся на разных сторонах разреза $(S, V - S)$, на пути p имеется как минимум одно ребро из T, которое пересекает разрез. Пусть таким ребром является ребро (x, y). Ребро (x, y) не входит в A, поскольку разрез согласован с A по ребрам. Так как (x, y) является единственным путем от u к v в T, его удаление разбивает T на два компонента. Добавление (u, v) восстанавливает разбиение, образуя новое остовное дерево $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

Теперь мы покажем, что T' — минимальное остовное дерево. Поскольку (u, v) — легкое ребро, пересекающее разбиение $(S, V - S)$, и (x, y) также пересекает это разбиение, $w(u, v) \leqslant w(x, y)$. Следовательно,

$$w\left(T'\right) = w(T) - w(x, y) + w(u, v) \leqslant w(T).$$

Однако T — минимальное остовное дерево, так что $w(T) \leqslant w(T')$. Следовательно, T' также должно быть минимальным остовным деревом.

Остается показать, что (u, v) действительно безопасное ребро для A. Мы имеем $A \subseteq T'$, поскольку $A \subseteq T$ и $(x, y) \notin A$. Таким образом, $A \cup \{(u, v)\} \subseteq T'$ и, поскольку T' — минимальное остовное дерево, ребро (u, v) безопасно для A. ■

Теорема 23.1 помогает нам лучше понять, как работает процедура GENERIC_MST. В процессе работы алгоритма множество A всегда ациклическое; в противном случае минимальное остовное дерево, включающее A, содержало бы цикл, что приводит к противоречию. В любой момент выполнения алгоритма граф $G_A = (V, E)$ представляет собой лес и каждый из связных компонентов G_A является

деревом. (Некоторые из деревьев могут содержать всего одну вершину, например, в случае, когда алгоритм начинает работу: множество A в этот момент пустое, а лес содержит $|V|$ деревьев, по одному для каждой вершины.) Кроме того, любое безопасное для A ребро (u, v) соединяет различные компоненты G_A, поскольку множество $A \cup \{(u, v)\}$ должно быть ациклическим.

Цикл в строках 2–4 процедуры GENERIC_MST выполняется $|V| - 1$ раз, так как должны быть найдены все $|V| - 1$ ребер минимального остовного дерева. Изначально, когда $A = \emptyset$, в G_A имеется $|V|$ деревьев и каждая итерация уменьшает их количество на единицу. Когда лес содержит только одно дерево, алгоритм завершается.

Два алгоритма, рассмотренные в разделе 23.2, используют следствие из теоремы 23.1.

Следствие 23.2. Пусть $G = (V, E)$ — связный неориентированный граф с действительной весовой функцией w, определенной на E. Пусть A — подмножество E, которое входит в некоторое минимальное остовное дерево G, и пусть $C = (V_C, E_C)$ — связный компонент (дерево) в лесу $G_A = (V, A)$. Если (u, v) — легкое ребро, соединяющее C с некоторым другим компонентом в G_A, то ребро (u, v) безопасно для A.

Доказательство. Разрез $(V_C, V - V_C)$ согласован с A, и (u, v) — легкое ребро для данного разреза. Следовательно, ребро (u, v) безопасно для A. ■

Упражнения

23.1-1. Пусть (u, v) — ребро минимального веса в графе G. Покажите, что (u, v) принадлежит некоторому минимальному остовному дереву G.

23.1-2. Профессор утверждает, что верно следующее обращение теоремы 23.1. Пусть $G = (V, E)$ — связный неориентированный граф с действительной весовой функцией w, определенной на E. Пусть A — подмножество E, входящее в некоторое минимальное остовное дерево G, $(S, V - S)$ — произвольный разрез G, согласованный с A, и пусть (u, v) — безопасное для A ребро, пересекающее разрез $(S, V - S)$. Тогда (u, v) — легкое ребро для данного разреза. Приведите контрпример, демонстрирующий некорректность профессорского обращения теоремы.

23.1-3. Покажите, что если ребро (u, v) содержится в некотором минимальном остовном дереве, то оно является легким ребром, пересекающим некоторый разрез графа.

23.1-4. Рассмотрим множество всех ребер, каждый элемент которого является легким ребром для какого-то из возможных разрезов графа. Приведите

простой пример, когда такое множество не образует минимального остовного дерева.

23.1-5. Пусть e — ребро с максимальным весом в некотором цикле связного графа $G = (V, E)$. Докажите, что имеется минимальное остовное дерево графа $G' = (V, E - \{e\})$, которое одновременно является минимальным остовным деревом G, т.е. что существует минимальное остовное дерево G, не включающее e.

23.1-6. Покажите, что граф имеет единственное минимальное остовное дерево, если для каждого разреза графа имеется единственное легкое ребро, пересекающее этот разрез. Покажите при помощи контрпримера, что обратное утверждение не верно.

23.1-7. Покажите, что если вес любого из ребер графа положителен, то любое подмножество ребер, объединяющее все вершины и имеющее минимальный общий вес, должно быть деревом. Приведите пример, показывающий, что это не так, если ребра могут иметь отрицательный вес.

23.1-8. Пусть T — минимальное остовное дерево графа G, и пусть L — отсортированный список весов ребер T. Покажите, что для любого другого минимального остовного дерева T' графа G отсортированный список весов ребер будет тем же.

23.1-9. Пусть T — минимальное остовное дерево графа $G = (V, E)$, а V' — подмножество V. Пусть T' — подграф T, порожденный V', а G' — подграф G, порожденный V'. Покажите, что если T' — связный граф, то он является минимальным остовным деревом G'.

23.1-10. Пусть дан граф G и минимальное остовное дерево T. Предположим, что мы уменьшаем вес одного из ребер в T. Покажите, что T при этом останется минимальным остовным деревом G. Более строго, пусть T — минимальное остовное дерево G с весами ребер, заданными весовой функцией w. Выберем одно ребро $(x, y) \in T$ и положительное число k, и определим весовую функцию w' следующим образом:

$$w'(u, v) = \begin{cases} w(u, v) & \text{если } (u, v) \neq (x, y), \\ w(x, y) - k & \text{если } (u, v) = (x, y). \end{cases}$$

Покажите, что если веса ребер определяются функцией w', T остается минимальным остовным деревом G.

⋆ 23.1-11. Пусть дан граф G и минимальное остовное дерево T. Предположим, что мы уменьшаем вес одного из ребер, не входящих в T. Разработайте алгоритм для поиска минимального остовного дерева модифицированного графа.

23.2 Алгоритмы Крускала и Прима

Описанные в этом разделе алгоритмы следуют общей схеме поиска минимального остовного дерева. Каждый из них использует свое правило для определения безопасных ребер в строке 3 процедуры GENERIC_MST. В алгоритме Крускала множество A является лесом. В A добавляются безопасные ребра, которые являются ребрами минимального веса, объединяющими два различных компонента. В алгоритме Прима множество A образует единое дерево. В A добавляются безопасные ребра, которые являются ребрами минимального веса, соединяющими дерево с вершиной вне дерева.

Алгоритм Крускала

Алгоритм Крускала непосредственно основан на обобщенном алгоритме поиска минимального остовного дерева, приведенном в разделе 23.1. Он находит безопасное ребро для добавления в растущий лес путем поиска ребра (u, v) с минимальным весом среди всех ребер, соединяющих два дерева в лесу. Обозначим два дерева, соединяемые ребром (u, v), как C_1 и C_2. Поскольку (u, v) должно быть легким ребром, соединяющим C_1 с некоторым другим деревом, из следствия 23.2 вытекает, что (u, v) — безопасное для C_1 ребро. Алгоритм Крускала является жадным, поскольку на каждом шаге он добавляет к лесу ребро с минимально возможным весом.

Наша реализация алгоритма Крускала напоминает алгоритм для вычисления связных компонентов в разделе 21.1. Она использует структуру для представления непересекающихся множеств. Каждое множество содержит вершины дерева в текущем лесу. Операция FIND_SET(u) возвращает представительный элемент множества, содержащего u. Таким образом, мы можем определить, принадлежат ли две вершины u и v одному и тому же дереву, проверяя равенство FIND_SET(u) и FIND_SET(v). Объединение деревьев выполняется при помощи процедуры UNION.

```
MST_KRUSKAL(G, w)
1  A ← ∅
2  for (Для) каждой вершины v ∈ V[G]
3       do MAKE_SET(v)
4  Сортируем ребра из E в неубывающем порядке их весов w
5  for (Для) каждого (u, v) ∈ E (в порядке возрастания веса)
6       do if FIND_SET(u) ≠ FIND_SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

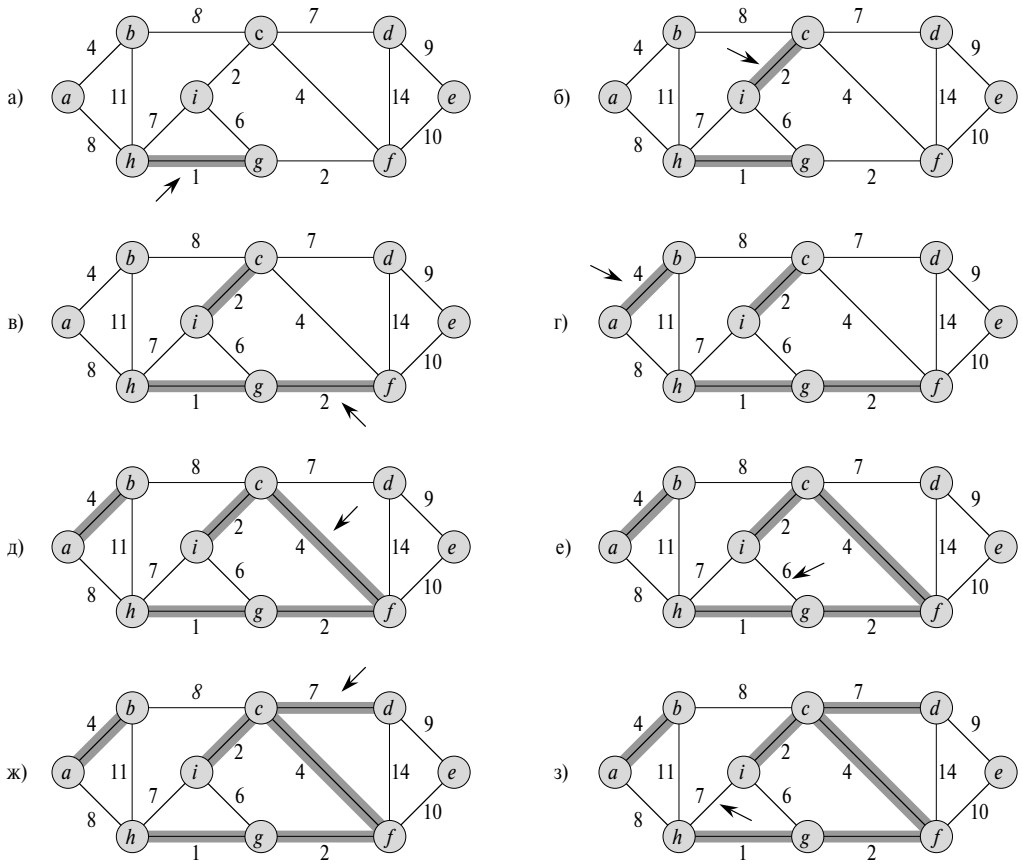
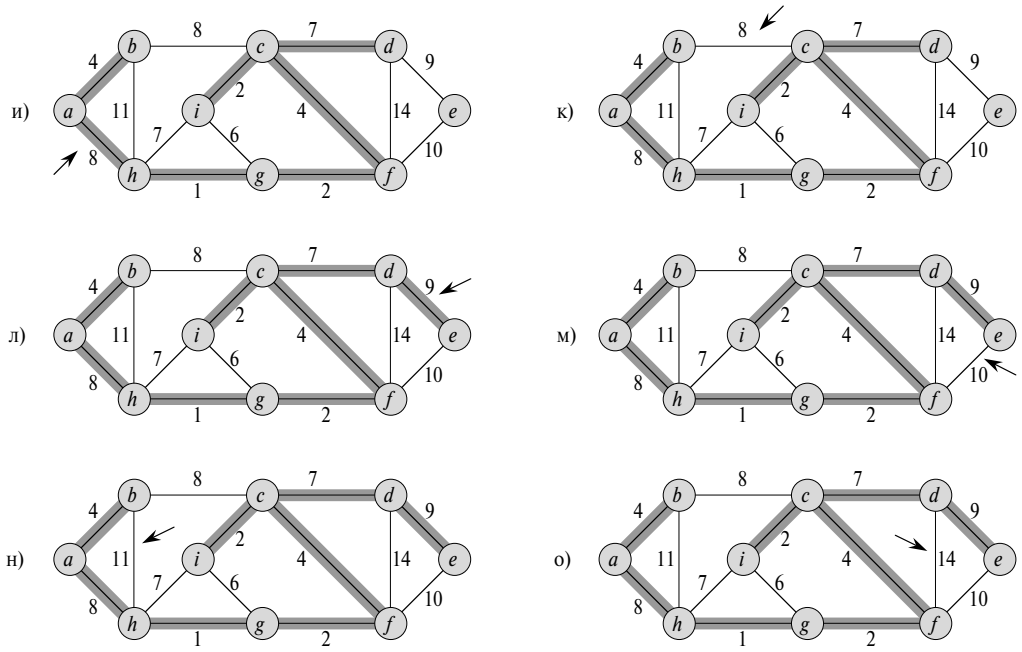


Рис. 23.4. Выполнение алгоритма Крускала для графа, приведенного на рис. 23.1. Заштрихованные ребра принадлежат растущему лесу A

Алгоритм Крускала работает так, как показано на рис. 23.4. В строках 1–3 выполняется инициализация множества A пустым множеством и создается $|V|$ деревьев, каждое из которых содержит по одной вершине. Ребра в E в строке 4 сортируются согласно их весу в неубывающем порядке. Цикл **for** в строках 5–8 проверяет для каждого ребра (u, v), принадлежат ли его концы одному и тому же дереву. Если это так, то данное ребро не может быть добавлено к лесу без того, чтобы создать при этом цикл, поэтому в таком случае ребро отбрасывается. В противном случае, когда концы ребра принадлежат разным деревьям, в строке 7 ребро (u, v) добавляется в множество A, и вершины двух деревьев объединяются в строке 8.

Время работы алгоритма Крускала для графа $G = (V, E)$ зависит от реализации структуры данных для непересекающихся множеств. Мы будем считать, что лес непересекающихся множеств реализован так, как описано в разделе 21.3, с эв-



ристиками объединения по рангу и сжатия пути, поскольку асимптотически это наиболее быстрая известная реализация. Инициализация множества A в строке 1 занимает время $O(1)$, а время, необходимое для сортировки множества в строке 4, равно $O(E \lg E)$ (стоимость $|V|$ операций MAKE_SET в цикле **for** в строках 2–3 мы учтем чуть позже). Цикл **for** в строках 5–8 выполняет $O(E)$ операций FIND_SET и UNION над лесом непересекающихся множеств. Вместе с $|V|$ операциями MAKE_SET эта работа требует времени $O((V + E)\alpha(V))$, где α — очень медленно растущая функция, определенная в разделе 21.4. Поскольку мы предполагаем, что G — связный граф, справедливо соотношение $|E| \geqslant |V| - 1$, так что операции над непересекающимися множествами требуют $O(E\alpha(V))$ времени. Кроме того, поскольку $\alpha(|V|) = O(\lg V) = O(\lg E)$, общее время работы алгоритма Крускала равно $O(E \lg E)$. Заметим, что $|E| < |V|^2$, поэтому $\lg |E| = O(\lg V)$ и время работы алгоритма Крускала можно записать как $O(E \lg V)$.

Алгоритм Прима

Аналогично алгоритму Крускала, алгоритм Прима представляет собой частный случай обобщенного алгоритма поиска минимального остовного дерева из раздела 23.1. Алгоритм Прима очень похож на алгоритм Дейкстры для поиска кратчайшего пути в графе, который будет рассмотрен нами в разделе 24.3. Алгоритм Прима обладает тем свойством, что ребра в множестве A всегда образуют единое дерево. Как показано на рис. 23.5, дерево начинается с произвольной

корневой вершины r и растет до тех пор, пока не охватит все вершины в V. На каждом шаге к дереву A добавляется легкое ребро, соединяющее дерево и отдельную вершину из оставшейся части графа. В соответствии со следствием 23.2, данное правило добавляет только безопасные для A ребра; следовательно, по завершении алгоритма ребра в A образуют минимальное остовное дерево. Данная стратегия является жадной, поскольку на каждом шаге к дереву добавляется ребро, которое вносит минимально возможный вклад в общий вес.

Ключевым моментом в эффективной реализации алгоритма Прима является выбор нового ребра для добавления в дерево. В приведенном ниже псевдокоде в качестве входных данных алгоритму передаются связный граф G и корень r минимального остовного дерева. В процессе работы алгоритма все вершины, которые *не* входят в дерево, располагаются в очереди с приоритетами Q, основанной на значении поля key, причем меньшее значение этого поля означает более высокий приоритет в очереди. Для каждой вершины v значение поля $key\,[v]$ представляет собой минимальный вес среди всех ребер, соединяющих v с вершиной в дереве. Если ни одного такого ребра нет, считаем, что $key\,[v] = \infty$. Поле $\pi\,[v]$ указывает родителя v в дереве. В процессе работы алгоритма множество A из процедуры GENERIC_MST неявно поддерживается как

$$A = \{(v, \pi\,[v]) : v \in V - \{r\} - Q\}\,.$$

Когда алгоритм завершает работу, очередь с приоритетами Q пуста и минимальным остовным деревом для G является дерево

$$A = \{(v, \pi\,[v]) : v \in V - \{r\}\}\,.$$

```
MST_PRIM(G, w, r)
 1  for (Для) каждой вершины u ∈ V[G]
 2       do key[u] ← ∞
 3          π[u] ← NIL
 4  key[r] ← 0
 5  Q ← V[G]
 6  while Q ≠ ∅
 7       do u ← EXTRACT_MIN(Q)
 8          for (Для) каждой вершины v ∈ Adj[u]
 9               do if v ∈ Q и w(u, v) < key[v]
10                    then π[v] ← u
11                         key[v] ← w(u, v)
```

Работа алгоритма Прима проиллюстрирована на рис. 23.5. В строках 1–5 ключи всех вершин устанавливаются равными ∞ (за исключением корня r, ключ которого равен 0, так что он оказывается первой обрабатываемой вершиной),

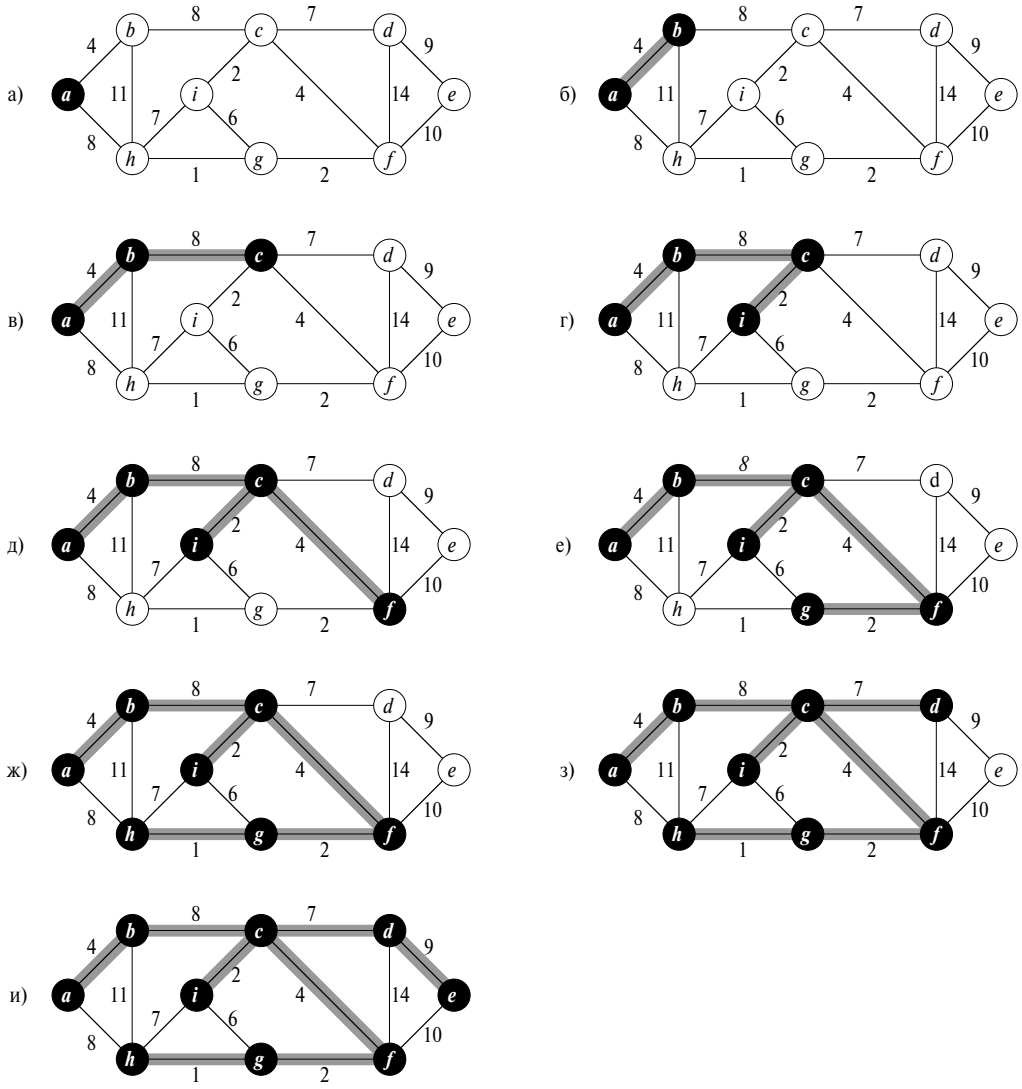


Рис. 23.5. Выполнение алгоритма Прима для графа, приведенного на рис. 23.1. Корневая вершина — a. Заштрихованные ребра принадлежат растущему дереву

указателям на родителей для всех узлов присваиваются значения NIL и все вершины вносятся в очередь с приоритетами Q. Алгоритм поддерживает следующий инвариант цикла, состоящий из трех частей.

Перед каждой итерацией цикла **while** в строках 6–11

1. $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$;

2. вершины, уже помещенные в минимальное остовное дерево, принадлежат множеству $V - Q$;
3. для всех вершин $v \in Q$ справедливо следующее: если $\pi[v] \neq$ NIL, то $key[v] < \infty$ и $key[v]$ — вес легкого ребра $(v, \pi[v])$, соединяющего v с некоторой вершиной, уже находящейся в минимальном остовном дереве.

В строке 7 определяется вершина u, принадлежащая легкому ребру, пересекающему разрез $(V - Q, Q)$ (за исключением первой итерации, когда $u = r$ в соответствии с присвоением в строке 4). Удаление u из множества Q добавляет его в множество $V - Q$ вершин дерева. Цикл **for** в строках 8–11 обновляет поля key и π для каждой вершины v, смежной с u и не находящейся в дереве. Это обновление сохраняет третью часть инварианта.

Производительность алгоритма Прима зависит от выбранной реализации очереди с приоритетами Q. Если реализовать ее как бинарную пирамиду (см. главу 6), то для выполнения инициализации в строках 1–5 можно использовать процедуру BUILD_MIN_HEAP, что потребует времени $O(V)$. Тело цикла **while** выполняется $|V|$ раз, а поскольку каждая операция EXTRACT_MIN занимает время $O(\lg V)$, общее время всех вызовов процедур EXTRACT_MIN составляет $O(V \lg V)$. Цикл **for** в строках 8–11 выполняется всего $O(E)$ раз, поскольку сумма длин всех списков смежности равна $2|E|$. Внутри цикла **for** проверка на принадлежность Q в строке 9 может быть реализована за постоянное время, если воспользоваться для каждой вершины битом, указывающим, находится ли она в Q, и обновлять этот бит при удалении вершины из Q. Присвоение в строке 11 неявно включает операцию DECREASE_KEY над пирамидой. Время выполнения этой операции — $O(\lg V)$. Таким образом, общее время работы алгоритма Прима составляет $O(V \lg V + E \lg V) = O(E \lg V)$, что асимптотически совпадает со временем работы рассмотренного ранее алгоритма Крускала.

Однако асимптотическое время работы алгоритма Прима можно улучшить за счет применения фибоначчиевых пирамид. В главе 20 показано, что если $|V|$ элементов организованы в фибоначчиеву пирамиду, то операцию EXCTARCT_MIN можно выполнить за амортизированное время $O(\lg V)$, а операцию DECREASE_KEY — за амортизированное время $O(1)$. Следовательно, при использовании фибоначчиевой пирамиды для реализации очереди с приоритетами Q общее время работы алгоритма Прима улучшается до $O(E + V \lg V)$.

Упражнения

23.2-1. Алгоритм Крускала может возвращать разные остовные деревья для одного и того же входного графа G в зависимости от расположения ребер с одинаковым весом при сортировке. Покажите, что для любого минимального остовного дерева T графа G можно указать способ сортировки

ребер G, для которого алгоритм Крускала даст минимальное остовное дерево T.

23.2-2. Предположим, что граф $G = (V, E)$ представлен при помощи матрицы смежности. Разработайте простую реализацию алгоритма Прима для этого случая, время работы которой равно $O\left(V^2\right)$.

23.2-3. Будет ли реализация алгоритма Прима с использованием фибоначчиевых пирамид асимптотически быстрее реализации с использованием бинарных пирамид для разреженного графа $G = (V, E)$, где $|E| = \Theta(V)$? А для плотного графа, в котором $|E| = \Theta\left(V^2\right)$? Каким образом должны быть связаны $|E|$ и $|V|$, чтобы реализация с использованием фибоначчиевых пирамид была быстрее реализации с использованием бинарных пирамид?

23.2-4. Предположим, что все веса ребер графа представляют собой целые числа в диапазоне от 1 до $|V|$. Насколько быстрым можно сделать алгоритм Крускала в этом случае? А в случае, когда вес каждого ребра представляет собой целое число в диапазоне от 1 до W для некоторой константы W?

23.2-5. Предположим, что вес каждого из ребер графа представляет собой целое число в диапазоне от 1 до $|V|$. Насколько быстрым можно сделать алгоритм Прима в этом случае? А в случае, когда вес любого ребра представляет собой целое число в диапазоне от 1 до W для некоторой константы W?

⋆ 23.2-6. Предположим, что веса ребер графа равномерно распределены на полуоткрытом интервале $[0, 1)$. Какой алгоритм — Крускала или Прима — будет работать быстрее в этом случае?

⋆ 23.2-7. Предположим, что граф G имеет уже вычисленное минимальное остовное дерево. Насколько быстро можно обновить минимальное остовное дерево при добавлении в G новой вершины и инцидентных ребер?

23.2-8. Профессор предложил новый алгоритм декомпозиции для вычисления минимальных остовных деревьев, заключающийся в следующем. Для данного графа $G = (V, E)$ разбиваем множество вершин V на два подмножества V_1 и V_2, таких что $|V_1|$ и $|V_2|$ отличаются не более, чем на 1. Пусть E_1 — множество ребер, инцидентных только над вершинами в V_1, а E_2 — множество ребер, инцидентных только над вершинами в V_2. Рекурсивно решаем задачу поиска минимальных остовных деревьев в каждом их подграфов $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$, а затем выбираем среди ребер E ребро с минимальным весом, пересекающее разрез (V_1, V_2), и используем его для объединения двух полученных минимальных остовных деревьев в одно.

Докажите или опровергните корректность описанного алгоритма поиска минимального остовного дерева.

Задачи

23-1. **Второе минимальное остовное дерево**

$G = (V, E)$ — неориентированный связный граф с весовой функцией $w : E \to \mathbf{R}$; предположим, что $|E| \geqslant |V|$ и что веса всех ребер различны.

Определим второе минимальное остовное дерево следующим образом. Пусть $\mathcal{T}$ — множество всех остовных деревьев G, и пусть T' — минимальное остовное дерево G. Тогда ***вторым минимальным остовным деревом*** (second-best minimum spanning tree) будет такое дерево T, что $w(T) = \min_{T'' \in \mathcal{T} - \{T'\}} \{w(T'')\}$.

а) Покажите, что минимальное остовное дерево единственное, но вторых минимальных остовных деревьев может быть несколько.

б) Пусть T — минимальное остовное дерево G. Докажите, что существуют ребра $(u, v) \in T$ и $(x, y) \notin T$, такие что $T - \{(u, v)\} \cup \{(x, y)\}$ — второе минимальное остовное дерево G.

в) Пусть T — остовное дерево G, и пусть для любых двух вершин $u, v \in V$ $\max[u, v]$ — ребро максимального веса на единственном пути между u и v в T. Разработайте алгоритм, который за время $O(V^2)$ для данного T вычисляет $\max[u, v]$ для всех $u, v \in V$.

г) Разработайте эффективный алгоритм вычисления второго минимального остовного дерева G.

23-2. **Минимальное остовное дерево разреженного графа**

Для очень разреженного связного графа $G = (V, E)$ можно добиться дальнейшего улучшения времени работы $O(E + V \lg V)$ алгоритма Прима с использованием фибоначчиевых пирамид путем предварительной обработки G в целях уменьшения количества его вершин перед применением алгоритма Прима. В частности, для каждой вершины u мы выбираем инцидентное u ребро (u, v) с минимальным весом и помещаем это ребро в строящееся минимальное остовное дерево. Затем мы выполняем сжатие по всем выбранным ребрам (см. раздел Б.4). Вместо того чтобы проводить сжатие по одному ребру, мы сначала определяем множества вершин, которые объединяются в одну новую вершину. Затем мы создаем граф, который должен был бы получиться в результате объединения этих ребер по одному, но делаем это путем "переименования" ребер соответственно множествам, в которых оказываются концы этих ребер. При этом одинаково переименованными могут оказаться одновременно несколько ребер, и в таком случае в качестве результирующего рассматривается только одно ребро с весом, равным минимальному весу среди соответствующих исходных ребер.

Изначально мы полагаем строящееся минимальное остовное дерево T пустым, и для каждого ребра $(u, v) \in E$ выполняем присвоения $orig\,[u, v] = = (u, v)$ и $c\,[u, v] = w\,(u, w)$. Атрибут $orig$ используется для ссылки на исходное ребро графа, которое связано с данным ребром в сжатом графе. Атрибут c хранит вес ребра, и при сжатии его значение обновляется в соответствии с приведенной выше схемой выбора весов ребер. Процедура MST_REDUCE получает в качестве входных параметров G, $orig$, c и T, и возвращает сжатый граф G' и обновленные атрибуты $orig'$ и c' для графа G'. Процедура также переносит ребра из G в минимальное остовное дерево T.

```
MST_REDUCE(G, orig, c, T)
 1  for (Для) каждого v ∈ V[G]
 2       do mark[v] ← FALSE
 3          MAKE_SET(v)
 4  for (Для) каждого u ∈ V[G]
 5       do if mark[u] = FALSE
 6             then Выбираем v ∈ Adj[u] с минимальным c[u, v]
 7                  UNION(u, v)
 8                  T ← T ∪ {orig[u, v]}
 9                  mark[u] ← mark[v] ← TRUE
10  V[G'] ← {FIND_SET(v) : v ∈ V[G]}
11  E[G'] ← ∅
12  for (Для) каждого (x, y) ∈ E[G]
13       do u ← FIND_SET(x)
14          v ← FIND_SET(y)
15          if (u, v) ∉E[G']
16             then E[G'] ← E[G'] ∪ {(u, v)}
17                  orig'[u, v] ← orig[x, y]
18                  c'[u, v] ← c[x, y]
19             else if c[x, y] < c'[u, v]
20                     then orig'[u, v] ← orig[x, y]
21                          c'[u, v] ← c[x, y]
22  Строим списки смежности Adj for G'
23  return G', orig', c' и T
```

а) Пусть T — множество ребер, возвращаемое процедурой MST_REDUCE, и пусть A — минимальное остовное дерево графа G', образованное вызовом MST_PRIM(G', c', r), где r — произвольная вершина из $V\,[G']$. Докажите, что $T \cup \{orig'\,[x, y] : (x, y) \in A\}$ представляет собой минимальное остовное дерево графа G.

б) Докажите, что $|V\,[G']| \leqslant |V|/2$.

в) Покажите, как реализовать процедуру MST_REDUCE так, чтобы время ее работы составляло $O(E)$. (*Указание*: воспользуйтесь простыми структурами данных.)

г) Предположим, мы k раз применяем процедуру MST_REDUCE, используя полученные при очередном вызове G', $orig'$ и c' в качестве входных данных для следующего вызова и накапливаем ребра в T. Покажите, что общее время выполнения всех k вызовов составляет $O(k\,E)$.

д) Предположим, что после k вызовов процедуры MST_REDUCE мы применяем алгоритм Прима MST_PRIM(G', c', r), где G' и c' получены в результате последнего вызова процедуры MST_REDUCE, а r — произвольная вершина из $V[G']$. Покажите, как выбрать k, чтобы общее время работы составило $O(E \lg\lg V)$. Докажите, что ваш выбор k минимизирует общее асимптотическое время работы.

е) Для каких значений $|E|$ (выраженных через $|V|$) алгоритм Прима с предварительным сжатием эффективнее алгоритма Прима без сжатия?

23-3. **Узкое остовное дерево**

Назовем ***узким остовным деревом*** (bottleneck spanning tree) T неориентированного графа G остовное дерево G, в котором наибольший вес ребра минимален среди всех возможных остовных деревьев, и этот вес называется значением узкого остовного дерева.

а) Докажите, что минимальное остовное дерево является узким остовным деревом.

Из части *а*) следует, что поиск узкого остовного дерева не сложнее поиска минимального остовного дерева. В оставшейся части задачи мы покажем, что его можно найти за линейное время.

б) Разработайте алгоритм, который для данного графа G и целого числа b за линейное время определяет, превышает ли значение узкого остовного дерева число b или нет.

в) Воспользуйтесь вашим алгоритмом из части *б*) как подпрограммой в алгоритме решения задачи поиска узкого остовного дерева за линейное время. (*Указание*: можно воспользоваться подпрограммой сжатия множеств ребер, как в процедуре MST_REDUCE, описанной в задаче 23-2.)

23-4. **Альтернативные алгоритмы поиска минимальных остовных деревьев**

В этой задаче мы приведем псевдокоды трех различных алгоритмов. Каждый из них принимает в качестве входных данных граф и возвращает

множество ребер T. Для каждого из алгоритмов требуется доказать, что T является минимальным остовным деревом графа, или доказать, что это не так. Кроме того, опишите эффективную реализацию каждого из алгоритмов, независимо от того, вычисляет он минимальное остовное дерево или нет.

а) MAYBE_MST_A(G, w)

```
1  Сортируем ребра в невозрастающем порядке весов
2  T ← E
3  for (Для) каждого ребра e в отсортированном порядке
4       do if T − {e} — связный граф
5            then T ← T − {e}
6  return T
```

б) MAYBE_MST_B(G, w)

```
1  T ← ∅
2  for (Для) каждого ребра e в произвольном порядке
3       do if T ∪ {e} не имеет циклов
4            then T ← T ∪ {e}
5  return T
```

в) MAYBE_MST_C(G, w)

```
1  T ← ∅
2  for (Для) каждого ребра e в произвольном порядке
3       do T ← T ∪ {e}
4          if T имеет цикл c
5            then Пусть e′ — ребро максимального веса в c
6                 T ← T − {e′}
7  return T
```

Заключительные замечания

Книга Таржана (Tarjan) [292] содержит превосходный обзор задач, связанных с поиском минимальных остовных деревьев, и дополнительную информацию о них. История данной задачи изложена Грехемом (Graham) и Хеллом (Hell) [131].

Таржан указывает, что впервые алгоритм для поиска минимальных остовных деревьев был описан в 1926 году в статье Борувки (O. Borůvka). Его алгоритм состоит в выполнении $O(\lg V)$ итераций процедуры MST_REDUCE, описанной в задаче 23-2. Алгоритм Крускала описан в [195] в 1956 году, а алгоритм, известный как алгоритм Прима, — в работе Прима (Prim) [250], хотя до этого он был открыт Ярником (Jarnik) в 1930 году.

Причина, по которой жадные алгоритмы эффективно решают задачу поиска минимальных остовных деревьев, заключается в том, что множество лесов графа образует матроид (см. раздел 16.4).

Когда $|E| = \Omega(V \lg V)$, алгоритм Прима, реализованный с использованием фибоначчиевых пирамид, имеет время работы $O(E)$. Для более разреженных графов использование комбинации идей из алгоритма Прима, алгоритмов Крускала и Борувки, вместе с применением сложных структур данных дало возможность Фредману (Fredman) и Таржану [98] разработать алгоритм, время работы которого равно $O(E \lg^* V)$. Габов (Gabow), Галил (Galil), Спенсер (Spencer) и Таржан [102] усовершенствовали этот алгоритм, доведя время его работы до $O(E \lg \lg^* V)$. Чазел (Chazelle) [53] разработал алгоритм, время работы которого — $O(E\widehat{\alpha}(E, V))$, где $\widehat{\alpha}(E, V)$ — функция, обратная функции Аккермана (см. главу 21). В отличие от перечисленных ранее, алгоритм Чазела не является жадным.

С задачей поиска минимального остовного дерева связана задача ***проверки остовного дерева*** (spanning tree verification), в которой для данного графа $G = (V, E)$ и дерева $T \subseteq E$ требуется определить, является ли T минимальным остовным деревом G. Кинг (King) [177] разработал алгоритм решения данной задачи за линейное время, основанный на работах Комлёса (Komlós) [188] и Диксона (Dixon), Рауха (Rauch) и Таржана [77].

Все описанные выше алгоритмы детерминированные и относятся к модели на основе сравнений, описанной в главе 8. Каргер (Karger), Клейн (Klein) и Таржан [169] разработали рандомизированный алгоритм поиска минимальных остовных деревьев, математическое ожидание времени работы которого — $O(V + E)$. Этот алгоритм использует рекурсию наподобие алгоритма с линейным временем работы из раздела 9.3: рекурсивный вызов для вспомогательной задачи определяет подмножество ребер E', которое не может находиться ни в одном минимальном остовном дереве. Другой рекурсивный вызов, работающий с подмножеством $E - E'$, строит минимальное остовное дерево. Алгоритм также использует идеи из алгоритмов Борувки и Кинга.

Фредман и Виллард (Willard) [100] показали, как найти минимальное остовное дерево за время $O(V + E)$ с использованием детерминированного алгоритма, не основанного на сравнениях. Их алгоритм предполагает, что данные представляют собой b-битовые целые числа и что память компьютера состоит из адресуемых b-битовых слов.

ГЛАВА 24

Кратчайшие пути из одной вершины

Водителю автомобиля нужно найти как можно более короткий путь из Киева в Запорожье. Допустим, у него есть карта Украины, на которой указаны расстояния между каждой парой пересечений дорог. Как найти кратчайший маршрут?

Один из возможных способов — пронумеровать все маршруты из Киева в Запорожье, просуммировать длины участков на каждом маршруте и выбрать кратчайший из них. Однако легко понять, что даже если исключить маршруты, содержащие циклы, получится очень много вариантов, большинство которых просто не имеет смысла рассматривать. Например, очевидно, что маршрут из Киева в Запорожье через Львов — далеко не лучший выбор. Точнее говоря, такой маршрут никуда не годится, потому что Львов находится относительно Киева совсем в другой стороне.

В этой главе и в главе 25 будет показано, как эффективно решаются такие задачи. ***В задаче о кратчайшем пути*** (shortest-paths problem) задается взвешенный ориентированный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbf{R}$, отображающей ребра на их веса, значения которых выражаются действительными числами. ***Вес*** (weight) пути $p = \langle v_0, v_1, \ldots, v_k \rangle$ равен суммарному весу входящих в него ребер:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Вес кратчайшего пути (shortest-path weight) из вершины u в вершину v определяется соотношением

$$\delta(u,v) = \begin{cases} \min\left\{w(p) : u \overset{p}{\rightsquigarrow} v\right\} & \text{если имеется путь от } u \text{ к } v, \\ \infty & \text{в противном случае.} \end{cases}$$

Тогда по определению ***кратчайший путь*** (shortest path) из вершины u в вершину v — это любой путь, вес которого удовлетворяет соотношению $w(p) = \delta(u,v)$.

В примере, в котором рассматривается маршрут из Киева в Запорожье, карту дорог можно смоделировать в виде графа, вершины которого представляют перекрестки дорог, а ребра — отрезки дорог между перекрестками, причем вес каждого ребра равен расстоянию между соответствующими перекрестками. Цель — найти кратчайший путь от заданного перекрестка в Киеве (например, между улицами Клавдиевской и Корсуньской) к заданному перекрестку в Запорожье (скажем, между улицами Панфиловцев и Патриотической).

Вес каждого из ребер можно интерпретировать не как расстояние, а как другую метрику. Часто они используются для представления временны́х интервалов, стоимости, штрафов, убытков или любой другой величины, которая линейно накапливается по мере продвижения вдоль ребер графа и которую нужно свести к минимуму.

Алгоритм поиска в ширину, описанный в разделе 22.2, представляет собой алгоритм поиска кратчайшего пути по невзвешенному графу, т.е. по графу, каждому ребру которого приписывается единичный вес. Поскольку многие концепции, применяющиеся в алгоритме поиска в ширину, возникают при исследовании задачи о кратчайшем пути по взвешенным графам, рекомендуется освежить в памяти материал раздела 22.2.

Варианты

Настоящая глава посвящена ***задаче о кратчайшем пути из одной вершины*** (single-source shortest-paths problem), в которой для заданного графа $G = (V, E)$ требуется найти кратчайший путь, который начинается в определенной ***исходной вершине*** (source vertex) $s \in V$ (для краткости будем именовать ее истоком) и заканчивается в каждой из вершин $v \in V$. Предназначенный для решения этой задачи алгоритм позволяет решить многие другие задачи, в том числе те, что перечислены ниже.

- **Задача о кратчайшем пути в заданный пункт назначения** (single-destination shortest-paths problem). Требуется найти кратчайший путь в заданную ***вершину назначения*** (destination vertex) t, который начинается в каждой из вершин v. Поменяв направление каждого принадлежащего графу ребра, эту задачу можно свести к задаче о единой исходной вершине.

- **Задача о кратчайшем пути между заданной парой вершин** (single-pair shortest-paths problem). Требуется найти кратчайший путь из заданной вершины u в заданную вершину v. Если решена задача о заданной исходной вершине u, то эта задача тоже решается. Более того, для этой задачи не найдено ни одного алгоритма, который бы в асимптотическом пределе был производительнее, чем лучшие из известных алгоритмов решения задачи об одной исходной вершине в наихудшем случае.
- **Задача о кратчайшем пути между всеми парами вершин** (all-pairs shortest-paths problem). Требуется найти кратчайший путь из каждой вершины u в каждую вершину v. Эту задачу тоже можно решить с помощью алгоритма, предназначенного для решения задачи об одной исходной вершине, однако обычно она решается быстрее. Кроме того, структура этой задачи представляет интерес сама по себе. В главе 25 задача обо всех парах вершин исследуется более подробно.

Оптимальная структура задачи о кратчайшем пути

Алгоритмы поиска кратчайших путей обычно основаны на том свойстве, что кратчайший путь между двумя вершинами содержит в себе другие кратчайшие пути. (В основе описанного в главе 26 алгоритма Эдмондса-Карпа (Edmonds-Karp), предназначенного для поиска максимального потока, также лежит это свойство). Это свойство оптимальной структуры — критерий применимости и динамического программирования (глава 15), и жадного метода (глава 16). Алгоритм Дейкстры (Dijkstra), с которым мы ознакомимся в разделе 24.3, представляет собой жадный алгоритм, а алгоритм Флойда-Воршалла (Floyd-Warshall), предназначенный для поиска кратчайшего пути между всеми парами вершин (см. главу 25), — это алгоритм динамического программирования. В сформулированной ниже лемме данное свойство оптимальной структуры определяется точнее.

Лемма 24.1 (Частичные пути кратчайшего пути являются кратчайшими путями.) Пусть $p = \langle v_1, v_2, \ldots, v_k \rangle$ — кратчайший путь из вершины v_1 к вершине v_k в заданном взвешенном ориентированном графе $G = (V, E)$ с весовой функцией $w : E \to \mathbf{R}$, а $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ — частичный путь пути p, который проходит из вершины v_i к вершине v_j для произвольных i и j, удовлетворяющих неравенству $1 \leqslant i \leqslant j \leqslant k$. Тогда p_{ij} — кратчайший путь из вершины v_i к вершине v_j.

Доказательство. Если разложить путь p на составные части $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, то будет выполняться соотношение $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Теперь предположим, что существует путь p'_{ij} из вершины v_i к вершине v_j, вес которого удовлетворяет неравенству $w\left(p'_{ij}\right) < w(p_{ij})$ Тогда $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ — путь

из вершины v_1 к вершине v_k, вес которого $w(p_{1i}) + w\left(p'_{ij}\right) + w(p_{jk})$ меньше веса $w(p)$. Это противоречит предположению о том, что p — кратчайший путь из вершины v_1 к вершине v_k. ■

Ребра с отрицательным весом

В некоторых экземплярах задачи о кратчайшем пути из фиксированного истока, веса ребер могут принимать отрицательные значения. Если граф $G = (V, E)$ не содержит циклов с отрицательным весом, достижимых из истока s, то вес кратчайшего пути $\delta(s, v)$ остается вполне определенной величиной для каждой вершины $v \in V$, даже если он принимает отрицательное значение. Если же такой цикл достижим из истока s, веса кратчайших путей перестают быть вполне определенными величинами. В этой ситуации ни один путь из истока s в любую из вершин цикла не может быть кратчайшим, потому что всегда можно найти путь с меньшим весом, который проходит по предложенному "кратчайшему" пути, а потом обходит цикл с отрицательным весом. Если на некотором пути из вершины s к вершине v встречается цикл с отрицательным весом, мы определяем $\delta(s, v) = -\infty$.

На рис. 24.1 проиллюстрировано влияние наличия отрицательных весов и циклов с отрицательным весом на веса кратчайших путей. Поскольку из вершины s в вершину a ведет всего один путь (путь $\langle s, a\rangle$), то выполняется равенство $\delta(s, a) = w(s, a) = 3$. Аналогично, имеется всего один путь из вершины s в вершину b, поэтому $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$. Из вершины s в вершину c можно провести бесконечно большое количество путей: $\langle s, c\rangle$, $\langle s, c, d, c\rangle$, $\langle s, c, d, c, d, c\rangle$ и т.д. Поскольку вес цикла $\langle c, d, c\rangle$ равен $6 + (-3) = 3 > 0$, т.е. он положительный, кратчайший путь из вершины s в вершину c — это $\langle s, c\rangle$ вес которого равен $\delta(s, c) = 5$. Аналогично, кратчайший путь из вершины s в вершину d — $\langle s, c, d\rangle$, с весом $\delta(s, d) = w(s, c) + w(c, d) = 11$. Существует бесконечно большое количество путей из s в вершину e: $\langle s, e\rangle$, $\langle s, e, f, e\rangle$, $\langle s, e, f, e, f, e\rangle$ и т.д. Однако поскольку вес цикла $\langle e, f, e\rangle$ равен $3 + (-6) = -3 < 0$, т.е. он отрицательный, не существует кратчайшего пути из вершины s в вершину e. Обходя цикл с отрицательным весом $\langle e, f, e\rangle$ сколько угодно раз, можно построить путь из s в вершину e, вес которого будет выражаться как угодно большим по модулю отрицательным числом, поэтому $\delta(s, e) = -\infty$; аналогично, $\delta(s, f) = -\infty$. Поскольку вершина g достижима из вершины f, можно также найти пути из вершины s в вершину g с произвольно большими отрицательными весами, поэтому $\delta(s, g) = -\infty$. Вершины h, i и j также образуют цикл с отрицательным весом. Однако они недостижимы из вершины s, поэтому $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

В некоторых алгоритмах поиска кратчайшего пути, таких как алгоритм Дейкстры, предполагается, что вес любого из ребер входного графа неотрицательный, как это было в примере о дорожной карте. В других алгоритмах, таких как алго-

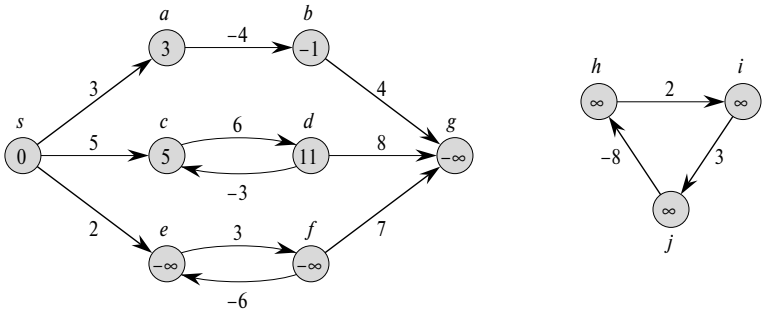


Рис. 24.1. Ребра с отрицательным весом в ориентированном графе. В каждой вершине обозначен вес кратчайшего пути в эту вершину из вершины s

ритм Беллмана-Форда (Bellman-Ford), ребра входных графов могут иметь отрицательный вес. Эти алгоритмы дают корректный ответ, если циклы с отрицательным весом недостижимы из истока. Обычно, если такой цикл с отрицательным весом существует, подобный алгоритм способен выявить его наличие и сообщить об этом.

Циклы

Может ли кратчайший путь содержать цикл? Только что мы убедились, что он не может содержать цикл с отрицательным весом. В него также не может входить цикл с положительным весом, поскольку в результате удаления этого цикла из пути получится путь, который исходит из того же истока и оканчивается в той же вершине, но обладает меньшим весом. Таким образом, если $p = \langle v_0, v_1, \ldots, v_k \rangle$ — путь, а $c = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ — цикл с положительным весом на этом пути (т.е. $v_i = v_j$ и $w(c) > 0$), то вес пути $p' = \langle v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k \rangle$ равен $w(p') = w(p) - w(c) < w(p)$, так что p не может быть кратчайшим путем из вершины v_0 в вершину v_k.

Остаются только циклы с нулевым весом. Однако из пути можно удалить цикл с нулевым весом, в результате чего получится другой путь с тем же весом. Таким образом, если существует кратчайший путь из истока s в целевую вершину v, содержащий цикл с нулевым весом, то существует также другой кратчайший путь из истока s в целевую вершину v, в котором этот цикл не содержится. Если кратчайший путь содержит циклы с нулевым весом, эти циклы можно поочередно удалять до тех пор, пока не получим кратчайший путь, в котором циклы отсутствуют. Поэтому без потери общности можно предположить, что если ведется поиск кратчайших путей, они не содержат циклов. Поскольку в любой ациклический путь в графе $G = (V, E)$ входит не более $|V|$ различных вершин, в нем

также содержится не более $|V| - 1$ ребер. Таким образом, можно ограничиться рассмотрением кратчайших путей, состоящих не более чем из $|V| - 1$ ребер.

Представление кратчайших путей

Часто требуется вычислить не только вес каждого из кратчайших путей, но и входящие в их состав вершины. Представление, которое используется для кратчайших путей, аналогично тому, что используется для описанных в разделе 22.2 деревьев поиска в ширину. В заданном графе $G = (V, E)$ для каждой вершины $v \in V$ поддерживается атрибут ***предшественник*** (predecessor) $\pi[v]$, в роли которого выступает либо другая вершина, либо значение NIL. В рассмотренных в этой главе алгоритмах поиска кратчайших путей атрибуты π присваиваются таким образом, что цепочка предшественников, которая начинается в вершине v, позволяет проследить путь, обратный кратчайшему пути из вершины s в вершину v. Таким образом, для заданной вершины v, для которой $\pi[v] \neq$ NIL, с помощью описанной в разделе 22.2 процедуры PRINT_PATH(G, s, v) можно вывести кратчайший путь из вершины s в вершину v.

Однако до тех пор, пока алгоритм поиска кратчайших путей не закончил свою работу, значения π не обязательно указывают кратчайшие пути. Как и при поиске в ширину, нас будет интересовать ***подграф предшествования*** (predecessor subgraph) $G_\pi = (V_\pi, E_\pi)$, построенный на основании значений π. Как и раньше, определим множество вершин V_π как множество, состоящее из тех вершин графа G, предшественниками которых не являются значения NIL, а также включает исток s:

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}.$$

Множество ориентированных ребер E_π — это множество ребер, индуцированных значениями π вершин из множества V_π:

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}.$$

Далее будет доказано, что значения π, полученные с помощью описанных в этой главе алгоритмов, обладают тем свойством, что после завершения этих алгоритмов G_π является “деревом кратчайших путей”. Неформально это дерево можно описать как корневое дерево, содержащее кратчайший путь из истока s к каждой вершине, достижимой из вершины s. Оно похоже на дерево поиска в ширину, знакомое нам по разделу 22.2, но содержит кратчайшие пути из истока, определенные не с помощью количества ребер, а с помощью значений их весов. Дадим более точное определение. Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbf{R}$. Предположим, что в нем не содержится циклов с отрицательным весом, достижимых из истока $s \in V$, а следовательно,

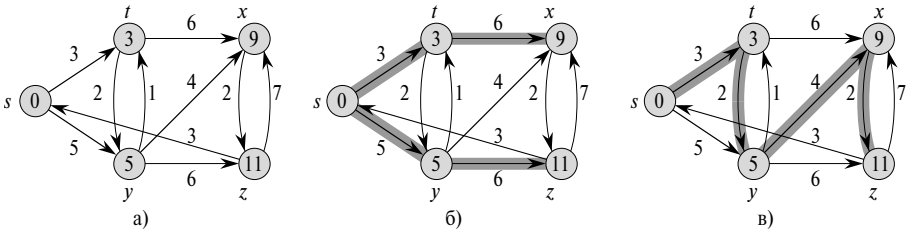


Рис. 24.2. Два разных дерева кратчайших путей, соответствующих одному и тому же взвешенному ориентированному графу

кратчайшие пути вполне определены. Тогда ***дерево кратчайших путей*** (shortest-paths tree) с корнем в вершине s — это ориентированный подграф $G' = (V', E')$, в котором множества $V' \subseteq V$ и $E' \subseteq E$ определяются такими условиями:

1. V' — множество вершин, достижимых из истока s графа G:
2. граф G' образует корневое дерево с корнем в вершине s:
3. для всех $v \in V'$ однозначно определенный простой путь из вершины s в вершину v в графе G' совпадает с кратчайшим путем из вершины s в вершину v в графе G.

Кратчайшие пути, как и деревья кратчайших путей, не обязательно единственны. Например, на рис. 24.2 изображен взвешенный ориентированный граф, на котором обозначен вес каждого из кратчайших путей из истока s (рис. 24.2*а*), а также два дерева кратчайших путей с корнем s (рис. 24.2*б* и *в*).

Ослабление

В описанных в этой главе алгоритмах используется метод ***релаксации***, или ***ослабления*** (relaxation). Для каждой вершины $v \in V$ поддерживается атрибут $d[v]$, представляющий собой верхнюю границу веса, которым обладает кратчайший путь из истока s в вершину v. Назовем атрибут $d[v]$ ***оценкой кратчайшего пути*** (shortest-path estimate). Инициализация оценок кратчайших путей и предшественников производится в приведенной ниже процедуре, время работы которой равно $\Theta(V)$:

```
INITIALIZE_SINGLE_SOURCE(G, s)
1  for (Для) каждой вершины v ∈ V[G]
2        do d[v] ← ∞
3           π[v] ← NIL
4  d[s] ← 0
```

После инициализации для всех $v \in V$ $\pi[v] = \text{NIL}$, $d[s] = 0$ и для всех $v \in V - \{s\}$ $d[v] = \infty$.

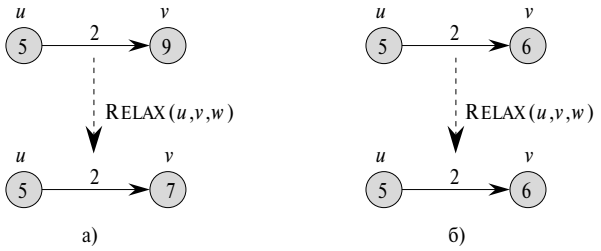


Рис. 24.3. Ослабление ребра (u, v) с весом $w(u, v) = 2$

Процесс ***ослабления***[1] (relaxation) ребра (u, v) заключается в проверке, нельзя ли улучшить найденный до сих пор кратчайший путь к вершине v, проведя его через вершину u, а также в обновлении атрибутов $d[v]$ и $\pi[v]$ при наличии такой возможности улучшения. Ослабление может уменьшить оценку кратчайшего пути $d[v]$ и обновить поле $\pi[v]$ вершины v. Приведенный ниже код выполняет ослабление ребра (u, v):

```
RELAX(u, v, w)
1  if d[v] > d[u] + w(u, v)
2     then d[v] ← d[u] + w(u, v)
3          π[v] ← u
```

На рис. 24.3 приведены два примера ослабления ребра; в одном из этих примеров оценка кратчайшего пути понижается, а в другом — не происходит никаких изменений, связанных с оценками. В каждой вершине на рисунке приводится оценка кратчайшего пути к этой вершине. В примере, приведенном в части *а* рисунка, перед ослаблением выполняется неравенство $d[v] > d[u] + w(u, v)$, поэтому в результате ослабления значение $d[v]$ уменьшается. В части *б* рисунка перед ослаблением выполняется неравенство $d[v] \leqslant d[u] + w(u, v)$, поэтому значение $d[v]$ остается неизменным.

В каждом из описанных в этой главе алгоритмов сначала вызывается процедура INITIALIZE_SINGLE_SOURCE, а затем производится ослабление ребер. Более того, ослабление — единственная операция, изменяющая оценки кратчайших путей и предшественников. Описанные в этой главе алгоритмы различаются тем, сколько раз в них производится ослабление ребер, а также порядком ребер, над которыми выполняется ослабление. В алгоритме Дейкстры и алгоритме поиска кратчайших путей в ориентированных ациклических графах каждое ребро

[1]Может показаться странным, что термин "ослабление" используется для операции, которая уточняет верхнюю границу. Этот термин определился исторически. Результат этапа ослабления можно рассматривать как ослабление ограничения $d[v] \leqslant d[u] + w(u, v)$, которое должно выполняться согласно неравенству треугольника (лемма 24.10), если $d[u] = \delta(s, u)$ и $d[v] = \delta(s, v)$. Другими словами, если справедливо неравенство $d[v] \leqslant d[u] + w(u, v)$, оно выполняется без "давления", поэтому данное условие "ослабляется".

ослабляется ровно по одному разу. В алгоритме Беллмана-Форда (Bellman-Ford) каждое ребро ослабляется по несколько раз.

Свойства кратчайших путей и ослабления

Для доказательства корректности описанных в этой главе алгоритмов будут использоваться некоторые свойства кратчайших путей и ослабления. Здесь эти свойства лишь сформулированы, а в разделе 24.5 будет представлено их формальное доказательство. Чтобы не нарушалась целостность изложения, каждое приведенное здесь свойство содержит номер соответствующей ему леммы или следствия из раздела 24.5. В последних пяти свойствах, которые относятся к оценкам кратчайших путей или подграфу предшествования, подразумевается, что они инициализированы с помощью вызова процедуры INITIALIZE_SINGLE_SOURCE(G, s), и что единственный способ, используемый для изменения оценок кратчайших путей и подграфа предшествования, — выполнение некоторой последовательности этапов ослабления.

Неравенство треугольника (лемма 24.10).
Для каждого ребра $(u, v) \in E$ выполняется неравенство $\delta(s, v) \leqslant \delta(s, u) + w(u, v)$.

Свойство верхней границы (лемма 24.11).
Для всех вершин $v \in V$ всегда выполняется неравенство $d[v] \geqslant \delta(s, v)$, а после того, как величина $d[v]$ становится равной $\delta(s, v)$, она больше не изменяется.

Свойство отсутствия пути (следствие 24.12).
Если из вершины s в вершину v нет пути, то всегда выполняется соотношение $d[v] = \delta(s, v) = \infty$.

Свойство сходимости (лемма 24.14).
Если $s \rightsquigarrow u \rightarrow v$ — кратчайший в графе G путь для некоторых вершин $u, v \in V$, и если равенство $d[u] = \delta(s, u)$ выполняется в некоторый момент времени до ослабления ребра (u, v), то в любой момент времени после этого $d[v] = \delta(s, v)$.

Свойство ослабления пути (лемма 24.15).
Если $p = \langle v_0, v_1, \ldots, v_k \rangle$ — кратчайший путь из вершины $s = v_0$ в вершину v_k, и ослабление ребер пути p производится в порядке (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k), то $d[v_k] = \delta(s, v_k)$. Это свойство выполняется независимо от других этапов ослабления, даже если они чередуются с ослаблением ребер, принадлежащих пути p.

Свойство подграфа предшествования (лемма 24.17).
Если для всех вершин $v \in V$ выполняется равенство $d[v] = \delta(s, v)$, то подграф предшествования представляет собой дерево кратчайших путей с корнем в истоке s.

Краткое содержание главы

В разделе 24.1 представлен алгоритм Беллмана-Форда, позволяющий решить задачу о кратчайшем пути из фиксированного истока в общем случае, когда вес любого ребра может быть отрицательным. Этот алгоритм отличается своей простотой. К его достоинствам также относится то, что он определяет, содержится ли в графе цикл с отрицательным весом, достижимый из истока. В разделе 24.2 приводится алгоритм с линейным временем выполнения, предназначенный для построения кратчайших путей из одной вершины в ориентированном ациклическом графе. В разделе 24.3 описывается алгоритм Дейкстры, который характеризуется меньшим временем выполнения, чем алгоритм Беллмана-Форда, но требует, чтобы вес каждого из ребер был неотрицательным. В разделе 24.4 показано, как с помощью алгоритма Беллмана-Форда можно решить частный случай задачи линейного программирования. Наконец, в разделе 24.5 доказываются сформулированные выше свойства кратчайших путей и ослабления.

Примем некоторые соглашения, необходимые для выполнения арифметических операций с бесконечно большими величинами. Будем считать, что для любого действительного числа $a \neq -\infty$ выполняется соотношение $a + \infty = \infty + a = \infty$. Кроме того, чтобы наши доказательства сохраняли силу при наличии циклов с отрицательным весом, будем считать, что для любого действительного числа $a \neq \infty$ выполняется соотношение $a + (-\infty) = (-\infty) + a = -\infty$.

Во всех описанных в этой главе алгоритмах предполагается, что ориентированный граф G хранится в виде списков смежных вершин. Кроме того, вместе с каждым ребром хранится его вес, так что при просмотре каждого списка смежности вес каждого из его ребер можно определить в течение времени $O(1)$.

24.1 Алгоритм Беллмана-Форда

Алгоритм Беллмана-Форда (Bellman-Ford algorithm) позволяет решить задачу о кратчайшем пути из одной вершины в общем случае, когда вес каждого из ребер может быть отрицательным. Для заданного взвешенного ориентированного графа $G = (V, E)$ с истоком s и весовой функцией $w : E \to \mathbf{R}$ алгоритм Беллмана-Форда возвращает логическое значение, указывающее на то, содержится ли в графе цикл с отрицательным весом, достижимый из истока. Если такой цикл существует, в алгоритме указывается, что решения не существует. Если же таких циклов нет, алгоритм выдает кратчайшие пути и их вес.

В этом алгоритме используется ослабление, в результате которого величина $d[v]$, представляющая собой оценку веса кратчайшего пути из истока s к каждой из вершин $v \in V$, уменьшается до тех пор, пока она не станет равна фактическому весу кратчайшего пути $\delta(s, v)$. Значение TRUE возвращается алгоритмом

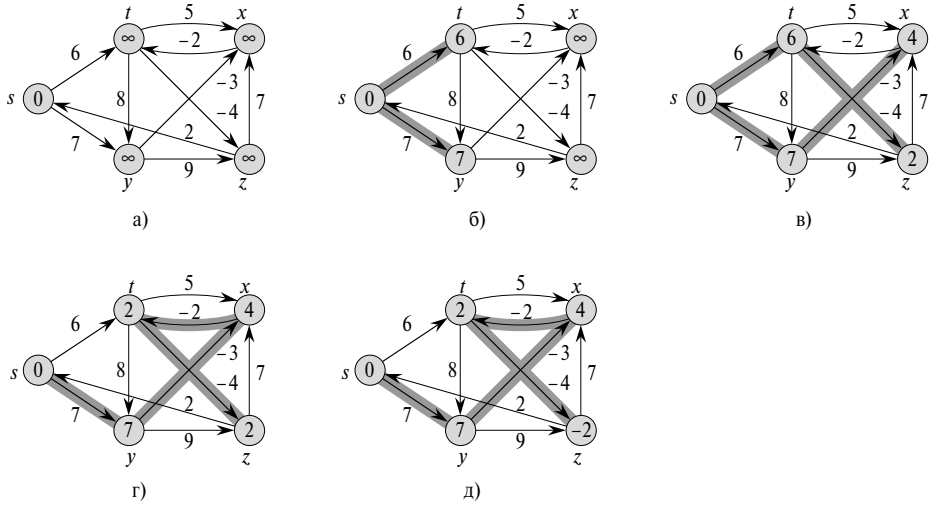


Рис. 24.4. Выполнение алгоритма Беллмана-Форда

тогда и только тогда, когда граф не содержит циклов с отрицательным весом, достижимых из истока.

BELLMAN_FORD(G, w, s)
1 INITIALIZE_SINGLE_SOURCE(G, s)
2 **for** $i \leftarrow 1$ **to** $|V[G]| - 1$
3 **do for** (для) каждого ребра $(u, v) \in E[G]$
4 **do** RELAX(u, v, w)
5 **for** (для) каждого ребра $(u, v) \in E[G]$
6 **do if** $d[v] > d[u] + w(u, v)$
7 **then return** FALSE
8 **return** TRUE

На рис. 24.4 проиллюстрирована работа алгоритма BELLMAN_FORD с графом, содержащим 5 вершин, в котором исток находится в вершине s.

Рассмотрим, как работает алгоритм. После инициализации в строке 1 всех значений d и π, алгоритм осуществляет $|V| - 1$ проходов по ребрам графа. Каждый проход соответствует одной итерации цикла **for** в строках 2–4 и состоит из однократного ослабления каждого ребра графа. После $|V| - 1$ проходов в строках 5–8 проверяется наличие цикла с отрицательным весом и возвращается соответствующее булево значение. (Причины, по которым эта проверка корректно работает, станут понятными несколько позже.) В примере, приведенном на рис. 24.4, алгоритм Беллман-Форд возвращает значение TRUE.

На рис. 24.4 в вершинах графа показаны значения атрибутов d на каждом этапе работы алгоритма, а выделенные ребра указывают на значения предшественников: если ребро (u, v) выделено, то $\pi[v] = u$. В рассматриваемом примере при каждом проходе ребра ослабляются в следующем порядке: (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y). В части *а* рисунка показана ситуация, сложившаяся непосредственно перед первым проходом по ребрам. В частях *б–д* проиллюстрирована ситуация после каждого очередного прохода по ребрам. Значения атрибутов d и π, приведенные в части *д*, являются окончательными.

Алгоритм Беллмана-Форда завершает свою работу в течение времени $O(VE)$, поскольку инициализация в строке 1 занимает время $\Theta(V)$, на каждый из $|V| - 1$ проходов по ребрам в строках 2–4 требуется время $\Theta(E)$, а на выполнение цикла **for** в строках 5–7 — время $O(E)$.

Чтобы доказать корректность алгоритма Беллмана-Форда, сначала покажем, что при отсутствии циклов с отрицательным весом он правильно вычисляет веса кратчайших путей для всех вершин, достижимых из истока.

Лемма 24.2. Пусть $G = (V, E)$ — взвешенный ориентированный граф с истоком s и весовой функцией $w : E \to \mathbf{R}$, который не содержит циклов с отрицательным весом, достижимых из вершины s. Тогда по завершении $|V| - 1$ итераций цикла **for** в строках 2–4 алгоритма BELLMAN_FORD, для всех вершин v, достижимых из вершины s, выполняется равенство $d[v] = \delta(s, v)$.

Доказательство. Докажем сформулированную лемму, воспользовавшись свойством ослабления пути. Рассмотрим произвольную вершину v, достижимую из вершины s. Пусть $p = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_0 = s$ и $v_k = v$ — кратчайший ациклический путь из вершины s в вершину v. Путь p содержит не более $|V| - 1$ ребер, так что $k \leqslant |V| - 1$. При каждой из $|V| - 1$ итераций цикла **for** в строках 2–4 ослабляются все $|E|$ ребер. Среди ребер, ослабленных во время i-й итерации ($i = 1, 2, \ldots, k$), находится ребро (v_{i-1}, v_i). Поэтому, согласно свойству ослабления путей, выполняется цепочка равенств $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$. ■

Следствие 24.3. Пусть $G = (V, E)$ — взвешенный ориентированный граф с истоком s и весовой функцией $w : E \to \mathbf{R}$. Тогда для каждой вершины $v \in V$ путь из вершины s в вершину v существует тогда и только тогда, когда после обработки графа G процедурой BELLMAN_FORD выполняется неравенство $d[v] < \infty$.

Доказательство. Доказательство этого следствия предлагается выполнить в качестве упражнения 24.1-2. ■

Теорема 24.4 (Корректность алгоритма Беллмана-Форда). Пусть алгоритм BELLMAN_FORD обрабатывает взвешенный ориентированный граф $G = (V, E)$ с истоком s и весовой функцией $w : E \to \mathbf{R}$. Если граф G не содержит циклов

с отрицательным весом, достижимых из вершины s, то этот алгоритм возвращает значение TRUE, для всех вершин $v \in V$ выполняется равенство $d[v] = \delta(s, v)$ и подграф предшествования G_π является деревом кратчайших путей с корнем в вершине s. Если же граф G содержит цикл с отрицательным весом, достижимый из вершины s, то алгоритм возвращает значение FALSE.

Доказательство. Предположим, что граф G не содержит циклов с отрицательным весом, достижимых из источника s. Сначала докажем, что по завершении работы алгоритма для всех вершин $v \in V$ выполняется равенство $d[v] = \delta(s, v)$. Если вершина v достижима из источника s, то доказательством этого утверждения служит лемма 24.2. Если же вершина v не достижима из вершины s, то это утверждение следует из свойства отсутствия пути. Таким образом, данное утверждение доказано. Из свойства подграфа предшествования и этого утверждения следует, что граф G_π — дерево кратчайших путей. А теперь с помощью обоснованного выше утверждения покажем, что алгоритм BELLMAN_FORD возвращает значение TRUE. По завершении работы алгоритма для всех ребер $(u, v) \in E$ выполняется соотношение

$$d[v] = \delta(s, v) \leqslant \delta(s, v) + w(u, v) = d[u] + w(u, v),$$

где неравенство следует из неравенства треугольника. Поэтому ни одна из проверок, выполненных в строке 6, не приведет к тому, что алгоритм BELLMAN_FORD возвратит значение FALSE. Следовательно, ему ничего не остается, как возвратить значение TRUE.

Теперь предположим, что граф G содержит цикл с отрицательным весом, достижимый из истока s; пусть это будет цикл $c = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_0 = v_k$. Тогда

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0. \tag{24.1}$$

Чтобы воспользоваться методом “от противного”, предположим, что алгоритм BELLMAN_FORD возвращает значение TRUE. Тогда для $i = 1, 2, \ldots, k$ выполняются неравенства $d[v_i] \leqslant d[v_{i-1}] + w(v_{i-1}, v_i)$. Просуммировав неравенства по всему циклу c, получим:

$$\sum_{i=1}^{k} d[v_i] \leqslant \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) = \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Поскольку $v_0 = v_k$, каждая вершина в цикле c в каждой сумме $\sum_{i=1}^{k} d[v_i]$ и $\sum_{i=1}^{k} d[v_{i-1}]$ появляется ровно по одному разу, так что

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}].$$

Кроме того, согласно следствию 24.3, атрибут $d[v_i]$ при $i = 1, 2, \ldots, k$ принимает конечные значения. Таким образом, справедливо неравенство

$$0 \leqslant \sum_{i=1}^{k} w(v_{i-1}, v_i),$$

что противоречит неравенству (24.1). Итак, мы приходим к выводу, что алгоритм BELLMAN_FORD возвращает значение TRUE, если граф G не содержит циклов с отрицательным весом, достижимых из истока, и значение FALSE — в противном случае. ■

Упражнения

24.1-1. Опишите работу алгоритма BELLMAN_FORD с ориентированным графом, показанным на рис. 24.4, в котором в качестве истока используется вершина z. В процессе каждого прохода ослабление ребер должно выполняться в том же порядке, что и на рисунке. Составьте список значений, которые принимают атрибуты d и π после каждого прохода. А теперь измените вес ребра (z, x), присвоив ему значение 4, и выполните алгоритм снова, используя в качестве истока вершину s.

24.1-2. Докажите следствие 24.3.

24.1-3. Пусть дан взвешенный ориентированный граф $G = (V, E)$, который не содержит циклов с отрицательным весом. Для каждой пары вершин u, $v \in V$ найдем минимальное количество ребер в кратчайшем пути от v к u (длина пути определяется его весом, а не количеством ребер). Пусть m — максимальное из всех полученных таким образом количеств ребер. Предложите простое изменение алгоритма BELLMAN_FORD, позволяющее ему завершаться после выполнения $m + 1$ проходов, даже если m заранее неизвестно.

24.1-4. Модифицируйте алгоритм BELLMAN_FORD таким образом, чтобы значения $-\infty$ в нем присваивались атрибутам $d[v]$ всех вершин v, для которых на одном из путей от истока к v имеется цикл с отрицательным весом.

⋆ 24.1-5. Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbf{R}$. Разработайте алгоритм со временем работы $O(VE)$, который позволял бы для каждой вершины $v \in V$ найти величину $\delta^*(v) = \min_{u \in V} \{\delta(u, v)\}$.

⋆ 24.1-6. Предположим, что взвешенный ориентированный граф $G = (V, E)$ содержит цикл с отрицательным весом. Разработайте эффективный алгоритм, позволяющий вывести список вершин одного такого цикла. Докажите, что предложенный вами алгоритм корректен.

24.2 Кратчайшие пути из одной вершины в ориентированных ациклических графах

Ослабляя ребра взвешенного ориентированного ациклического графа $G = (V, E)$ в порядке, определенном топологической сортировкой его вершин, кратчайшие пути из одной вершины можно найти в течение времени $\Theta(V + E)$. В ориентированном ациклическом графе кратчайшие пути всегда вполне определены, поскольку даже если у некоторых ребер вес отрицателен, циклов с отрицательными весами не существует.

Работа алгоритма начинается с топологической сортировки ориентированного ациклического графа (см. раздел 22.4), чтобы установить линейное упорядочение вершин. Если путь из вершины u к вершине v существует, то в топологической сортировке вершина u предшествует вершине v. По вершинам, расположенным в топологическом порядке, проход выполняется только один раз. При обработке каждой вершины производится ослабление всех ребер, исходящих из этой вершины.

```
DAG_SHORTEST_PATHS(G, w, s)
1  Топологическая сортировка вершин графа G
2  INITIALIZE_SINGLE_SOURCE(G, s)
3  for (для) каждой вершины u в порядке топологической сортировки
4        do for (Для) каждой вершины v ∈ Adj[u]
5              do RELAX(u, v, w)
```

Работа этого алгоритма проиллюстрирована на рис. 24.5. Вершины на рисунке топологически отсортированы слева направо. В роли истока выступает вершина s. В вершинах приведены значения атрибутов d, а выделенные ребра указывают значения π. В части *а* рисунка приведена ситуация перед первой итерацией цикла **for** в строках 3–5. В каждой из частей *б–ж* рисунка показаны ситуации, складывающиеся после каждой итерации цикла **for** в строках 3–5. Каждая новая черная вершина используется в соответствующей итерации в качестве u. В части *ж* приведены конечные значения.

Время выполнения этого алгоритма легко поддается анализу. Как показано в разделе 22.4, топологическую сортировку в строке 1 можно выполнить в течение времени $\Theta(V + E)$. Вызов процедуры INITIALIZE_SINGLE_SOURCE в строке 2 занимает время $\Theta(V)$. На каждую вершину приходится по одной итерации цикла **for** в строках 3–5. Для каждой вершины все исходящие из нее ребра проверяются ровно по одному разу. Таким образом, всего выполняется $|E|$ итераций внутреннего цикла **for** в строках 4–5 (мы воспользовались здесь групповым анализом). Поскольку каждая итерация внутреннего цикла **for** занимает время $\Theta(1)$, полное

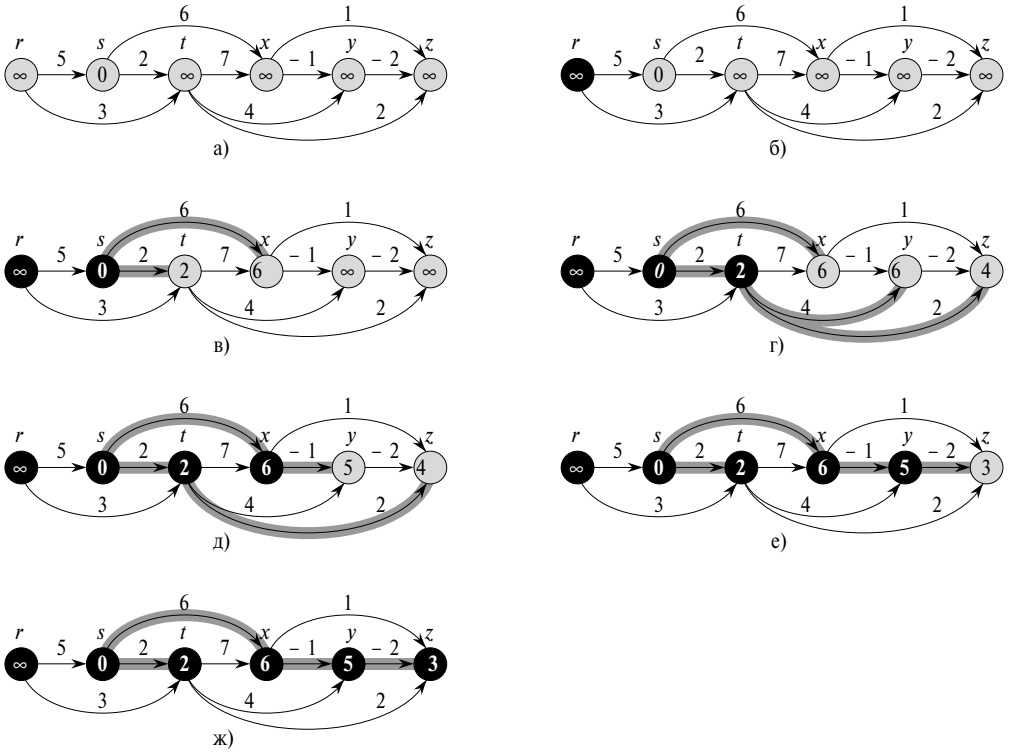


Рис. 24.5. Иллюстрация работы алгоритма, предназначенного для поиска кратчайших путей в ориентированном ациклическом графе

время работы алгоритма равно $\Theta(V + E)$, т.е. оно выражается линейной функцией от размера списка смежных вершин графа.

Из сформулированной ниже теоремы видно, что процедура DAG_SHORTEST_PATHS корректно вычисляет кратчайшие пути.

Теорема 24.5. Если взвешенный ориентированный граф $G = (V, E)$ содержит исток s и не содержит циклов, то по завершении процедуры DAG_SHORTEST_PATHS для всех вершин $v \in V$ выполняется равенство $d[v] = \delta(s, v)$ и подграф предшествования G_π представляет собой дерево кратчайших путей.

Доказательство. Сначала покажем, что по завершении процедуры для всех вершин $v \in V$ выполняется равенство $d[v] = \delta(s, v)$. Если вершина v недостижима из истока s, то, согласно свойству отсутствия пути, выполняется соотношение $d[v] = \delta(s, v) = \infty$. Теперь предположим, что вершина v достижима из истока s и, следовательно, существует кратчайший путь $p = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_0 = s$, а $v_k = v$. Поскольку вершины обрабатываются в порядке топологической

сортировки, ребра пути p ослабляются в порядке (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k). Из свойства ослабления путей следует, что по завершении процедуры для всех $i = 0, 1, , k$ выполняется равенство $d[v_i] = \delta(s, v_i)$. И наконец, согласно свойству подграфа предшествования, G_π — дерево кратчайших путей. ■

Интересное применение этого алгоритма возникает при определении критических путей во время анализа ***диаграммы PERT*** (PERT chart)[2]. Ребра представляют предназначенные для выполнения задания, а вес каждого из ребер — промежутки времени, необходимые для выполнения того или иного задания. Если ребро (u, v) входит в вершину v, а ребро (v, x) покидает ее, это означает, что задание (u, v) должно быть выполнено перед заданием (v, x). Путь по такому ориентированному ациклическому графу представляет последовательность заданий, которые необходимо выполнить в определенном порядке. ***Критический путь*** (critical path) — *самый длинный* путь по ориентированному ациклическому графу, соответствующий самому длительному времени, необходимому для выполнения упорядоченной последовательности заданий. Вес критического пути равен нижней границе полного времени выполнения всех заданий. Критический путь можно найти одним из таких двух способов:

- заменить знаки всех весов ребер и выполнить алгоритм DAG_SHORTEST_PATHS;
- воспользоваться модифицированным алгоритмом DAG_SHORTEST_PATHS, заменив в строке 2 процедуры INITIALIZE_SINGLE_SOURCE значение ∞ значением $-\infty$, а в процедуре RELAX — знак “$>$” знаком “$<$”.

Упражнения

24.2-1. Выполните процедуру DAG_SHORTEST_PATHS для графа, показанного на рис. 24.5, с использованием вершины r в качестве истока.

24.2-2. Предположим, что строка 3 в процедуре DAG_SHORTEST_PATHS изменена следующим образом:

> 3 **for** (для) первых $|V| - 1$ вершин, расположенных в порядке топологической сортировки

Покажите, что процедура останется корректной.

24.2-3. Представленная выше формулировка диаграммы PERT несколько неестественна. Логичнее было бы, если бы вершины представляли задания,

[2] Аббревиатура PERT расшифровывается как “program evaluation and review technique” — система планирования и руководства разработками.

а ребра — ограничения, накладываемые на порядок их выполнения, т.е. если бы наличие ребра (u, v) указывало на то, что задание u необходимо выполнить перед заданием v. Тогда бы вес присваивался не ребрам, а вершинам. Модифицируйте процедуру DAG_SHORTEST_PATHS таким образом, чтобы она в течение линейного времени позволяла найти самый длинный путь по ориентированному ациклическому графу со взвешенными вершинами.

24.2-4. Разработайте эффективный алгоритм, позволяющий определить общее количество путей, содержащихся в ориентированном ациклическом графе. Проанализируйте время работы этого алгоритма.

24.3 Алгоритм Дейкстры

Алгоритм Дейкстры решает задачу о кратчайшем пути из одной вершины во взвешенном ориентированном графе $G = (V, E)$ в том случае, когда веса ребер неотрицательны. Поэтому в настоящем разделе предполагается, что для всех ребер $(u, v) \in E$ выполняется неравенство $w(u, v) \geqslant 0$. Через некоторое время станет понятно, что при хорошей реализации алгоритм Дейкстры производительнее, чем алгоритм Беллмана-Форда.

В алгоритме Дейкстры поддерживается множество вершин S, для которых уже вычислены окончательные веса кратчайших путей к ним из истока s. В этом алгоритме поочередно выбирается вершина $u \in V - S$, которой на данном этапе соответствует минимальная оценка кратчайшего пути. После добавления этой вершины u в множество S производится ослабление всех исходящих из нее ребер. В приведенной ниже реализации используется неубывающая очередь с приоритетами Q, состоящая из вершин, в роли ключей для которых выступают значения d.

```
DIJKSTRA(G, w, s)
1  INITIALIZE_SINGLE_SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5        do u ← EXTRACT_MIN(Q)
6           S ← S ∪ {u}
7           for (для) каждой вершины v ∈ Adj[u]
8                 do RELAX(u, v, w)
```

Процесс ослабления ребер в алгоритме Дейкстры проиллюстрирован на рис. 24.6. Исток s расположен на рисунке слева от остальных вершин. В каждой вершине приведена оценка кратчайшего пути к ней, а выделенные ребра указывают предшественников. Черным цветом обозначены вершины, добавленные

в множество S, а белым — содержащиеся в неубывающей очереди с приоритетами $Q = V - S$. В части *а* рисунка проиллюстрирована ситуация, сложившаяся непосредственно перед выполнением первой итерации цикла **while** в строках 4–8. Выделенная серым цветом вершина имеет минимальное значение d и выбирается в строке 5 в качестве вершины u для следующей итерации. В частях *б–е* изображены ситуации после выполнения очередной итерации цикла **while**. В каждой из этих частей выделенная серым цветом вершина выбирается в качестве вершины u в строке 5. В части *е* приведены конечные значения величин d и π.

Опишем работу рассматриваемого алгоритма. В строке 1 производится обычная инициализация величин d и π, а в строке 2 инициализируется пустое множество вершин S. В этом алгоритме поддерживается инвариант, согласно которому в начале каждой итерации цикла **while** в строках 4–8 выполняется равенство $Q = V - S$. В строке 3 неубывающая очередь с приоритетами Q инициализируется таким образом, чтобы она содержала все вершины множества V; поскольку в этот момент $S = \emptyset$, после выполнения строки 3 сформулированный выше инвариант выполняется. При каждой итерации цикла **while** в строках 4–8 вершина u извлекается из множества $Q = V - S$ и добавляется в множество S, в результате чего инвариант продолжает соблюдаться. (Во время первой итерации этого цикла $u = s$.) Таким образом, вершина u имеет минимальную оценку кратчайшего пути среди всех вершин множества $V - S$. Затем строках 7–8 ослабляются все ребра (u, v), исходящие из вершины u. Если текущий кратчайший путь к вершине v может быть улучшен в результате прохождения через вершину u, выполняется ослабление и соответствующее обновление оценки величины $d[v]$ и предшественника $\pi[v]$. Обратите внимание, что после выполнения строки 3 вершины никогда не добавляются в множество Q и что каждая вершина извлекается из этого множества и добавляется в множество S ровно по одному разу, поэтому количество итераций цикла **while** в строках 4–8 равно $|V|$.

Поскольку в алгоритме Дейкстры из множества $V - S$ для помещения в множество S всегда выбирается самая "легкая" или "близкая" вершина, говорят, что этот алгоритм придерживается жадной стратегии. Жадные стратегии подробно описаны в главе 16, однако для понимания принципа работы алгоритма Дейкстры читать эту главу не обязательно. Жадные стратегии не всегда приводят к оптимальным результатам, однако, как видно из приведенной ниже теоремы и следствия из нее, алгоритм Дейкстры действительно находит кратчайшие пути. Основное — показать, что после каждого добавления вершины u в множество S выполняется равенство $d[u] = \delta(s, u)$.

Теорема 24.6 (Корректность алгоритма Дейкстры). По завершении обработки алгоритмом Дейкстры взвешенного ориентированного графа $G = (V, E)$ с неотрицательной весовой функцией w и истоком s для всех вершин $u \in V$ выполняется равенство $d[u] = \delta(s, u)$.

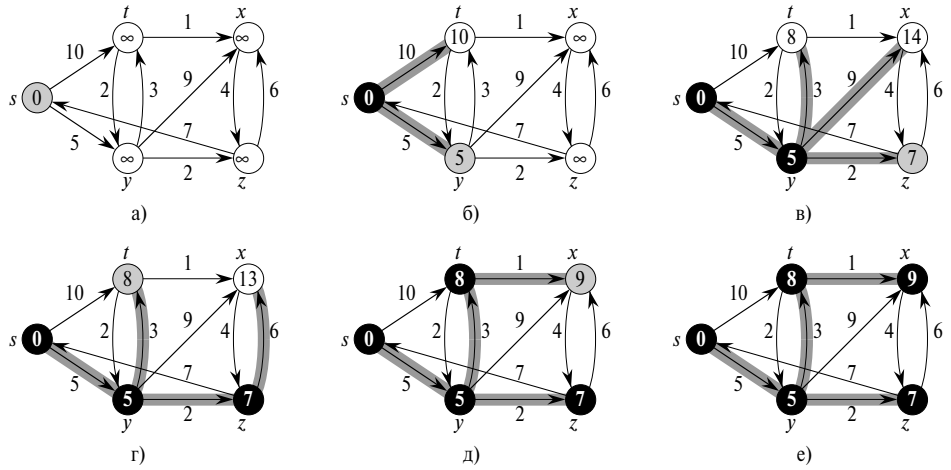


Рис. 24.6. Выполнение алгоритма Дейкстры

Доказательство. Воспользуемся сформулированным ниже инвариантом цикла.

> В начале каждой итерации цикла **while** в строках 4–8 для каждой вершины $v \in S$ выполняется равенство $d[v] = \delta(s, v)$.

Достаточно показать, что для каждой вершины $u \in V$ равенство $d[u] = \delta(s, u)$ выполняется в момент ее добавления в множество S. После того как будет показана справедливость равенства $d[u] = \delta(s, u)$, на основе свойства верхней границы мы покажем, что это равенство продолжает выполняться и в дальнейшем.

Инициализация. Изначально выполняется равенство $S = \emptyset$, поэтому справедливость инварианта очевидна.

Сохранение. Нужно показать, что при каждой итерации равенство $d[u] = \delta(s, u)$ выполняется для каждой вершины, добавленной в множество S. Воспользуемся методом "от противного". Чтобы получить противоречие, предположим, что u — первая добавленная в множество S вершина, для которой $d[u] \neq \delta(s, u)$. Сосредоточим внимание на ситуации, сложившейся в начале той итерации цикла **while**, в которой вершина u добавляется в множество S. Проанализировав кратчайший путь из вершины s в вершину u, можно будет получить противоречие, заключающееся в том, что на тот момент справедливо равенство $d[u] = \delta(s, u)$. Должно выполняться условие $u \neq s$, поскольку s — первая вершина, добавленная в множество S, и в момент ее добавления выполняется равенство $d[s] = \delta(s, s)$. Из условия $u \neq s$ следует также, что непосредственно перед добавлением вершины u в множество S оно не является пустым. Из вершины s в вершину u должен вести какой-нибудь путь, так как в противном случае, в соответствии со свойством отсутствия пути, выполняется соотношение $d[u] = \delta(s, u) = \infty$, нарушающее

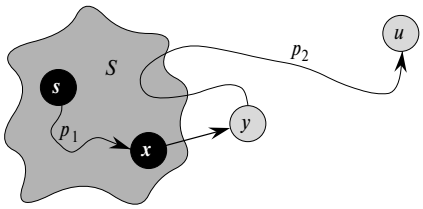


Рис. 24.7. Иллюстрация к доказательству теоремы 24.6

справедливость предположения, что $d[u] \neq \delta(s,u)$. Поскольку хоть один путь существует, то должен существовать и кратчайший путь p из вершины s в вершину u. Перед добавлением вершины u в множество S путь p соединяет вершину из множества S (вершину s) с вершиной из множества $V-S$ (вершину u). Рассмотрим первую вершину y на пути p, принадлежащую множеству $V-S$, а также предположим, что ей предшествует вершина $x \in S$. Тогда, как видно из рис. 24.7, путь p можно разложить на составляющие $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$. (Любой из путей p_1 и p_2 может не содержать ни одного ребра.)

В примере, приведенном на рисунке, множество S непосредственно перед добавлением в него вершины u не является пустым. Кратчайший путь p из истока s в вершину u раскладывается на составляющие $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$, где y — первая вершина на пути, не принадлежащая множеству S, а вершина $x \in S$ расположена непосредственно перед вершиной y. Вершины x и y не совпадают, однако возможен случай, когда $s = x$ или $y = u$. Путь p_2 может возвращаться в множество S (но может и не возвращаться).

Мы утверждаем, что когда вершина u добавляется в множество S, выполняется равенство $d[y] = \delta(s,y)$. Чтобы доказать это утверждение, заметим, что $x \in S$. Далее, поскольку вершина u выбрана как первая вершина, после добавления которой в множество S выполняется соотношение $d[u] \neq \neq \delta(s,u)$, то после добавления вершины x в множество S справедливо равенство $d[x] = \delta(s,x)$. В это время (т.е. при добавлении вершины u в множество S) происходит ослабление ребра (x,y), поэтому сформулированное выше утверждение следует из свойства сходимости.

Теперь можно получить противоречие, позволяющее доказать, что $d[u] = \delta(s,u)$. Поскольку на кратчайшем пути из вершины s в вершину u вершина y находится перед вершиной u, и вес каждого из ребер выражается неотрицательным значением (это особенно важно для ребер, из которых состоит путь p_2), выполняется неравенство $\delta(s,y) \leqslant \delta(s,u)$, поэтому

$$d[y] = \delta(s,y) \leqslant \delta(s,u) \leqslant d[u], \tag{24.2}$$

где последнее неравенство следует из свойства верхней границы. Однако поскольку и вершина u, и вершина y во время выбора вершины u в строке 5 находились в множестве $V - S$, выполняется неравенство $d[u] \leqslant d[y]$. Таким образом, оба неравенства в соотношении (24.2) фактически являются равенствами, т.е.

$$d[y] = \delta(s, y) = \delta(s, u) = d[u].$$

Следовательно, $d[u] = \delta(s, u)$, что противоречит нашему выбору вершины u. Приходим к выводу, что во время добавления вершины u в множество S выполняется равенство $d[u] = \delta(s, u)$, а следовательно, оно выполняется и в дальнейшем.

Завершение. По завершении работы алгоритма выполняется равенство $Q = \emptyset$. Из этого равенства и инварианта $Q = V - S$ следует, что $S = V$. Таким образом, для всех вершин $u \in V$ выполняется равенство $d[u] = \delta(s, u)$. ■

Следствие 24.7. Если выполнить алгоритм Дейкстры для взвешенного ориентированного графа $G = (V, E)$ с неотрицательной весовой функцией w и истоком s, то по завершении работы алгоритма подграф предшествования G_π является деревом кратчайших путей с корнем в вершине s.

Доказательство. Сформулированное выше утверждение непосредственно следует из теоремы 24.6 и свойства подграфа предшествования. ■

Анализ

Насколько быстро работает алгоритм Дейкстры? В нем поддерживается неубывающая очередь с приоритетами Q и тремя операциями, характерными для очередей с приоритетами: INSERT (явно вызывается в строке 3), EXTRACT_MIN (строка 5) и DECREASE_KEY (неявно присутствует в процедуре RELAX, которая вызывается в строке 8). Процедура INSERT, как и процедура EXTRACT_MIN, вызывается по одному разу для каждой вершины. Поскольку каждая вершина $v \in V$ добавляется в множество S ровно по одному разу, каждое ребро в списке смежных вершин $Adj[v]$ обрабатывается в цикле **for**, заданном в строках 7–8, ровно по одному разу на протяжении работы алгоритма. Так как полное количество ребер во всех списках смежных вершин равно $|E|$, всего выполняется $|E|$ итераций этого цикла **for**, а следовательно, не более $|E|$ операций DECREASE_KEY. (Еще раз заметим, что здесь используется групповой анализ.)

Время выполнения алгоритма Дейкстры зависит от реализации неубывающей очереди с приоритетами. Сначала рассмотрим случай, когда неубывающая очередь с приоритетами поддерживается за счет того, что все вершины пронумерованы от 1 до $|V|$. Атрибут $d[v]$ просто помещается в элемент массива с индексом v. Каждая операция INSERT и DECREASE_KEY занимает время $O(1)$, а каждая

операция EXTRACT_MIN — время $O(V)$ (поскольку в ней производится поиск по всему массиву); в результате полное время работы алгоритма равно $O\left(V^2 + E\right) = O\left(V^2\right)$.

Если граф достаточно разреженный, в частности, если количество вершин и ребер в нем связаны соотношением $E = o\left(V^2/\lg V\right)$, с практической точки зрения целесообразно реализовать неубывающую очередь с приоритетами в виде бинарной неубывающей пирамиды. (Как было сказано в разделе 6.5, важная деталь реализации заключается в том, что вершины и соответствующие им элементы пирамиды должны поддерживать идентификаторы друг друга.) Далее, каждая операция EXTRACT_MIN занимает время $O(\lg V)$. Как и раньше, таких операций $|V|$. Время, необходимое для построения неубывающей пирамиды, равно $O(V)$. Каждая операция DECREASE_KEY занимает время $O(\lg V)$, и всего выполняется не более $|E|$ таких операций. Поэтому полное время выполнения алгоритма равно $O((V + E)\lg V)$, что равно $O(E \lg V)$, если все вершины достижимы из истока. Это время работы оказывается лучшим по сравнению со временем работы прямой реализации $O\left(V^2\right)$, если $E = o\left(V^2/\lg V\right)$.

Фактически время работы алгоритма может достигать значения $O(V \lg V + E)$, если неубывающая очередь с приоритетами реализуется с помощью пирамиды Фибоначчи (см. главу 20). Амортизированная стоимость каждой из $|V|$ операций EXTRACT_MIN равна $O(\lg V)$, и каждый вызов процедуры DECREASE_KEY (всего не более $|E|$), занимает лишь $O(1)$ амортизированного времени. Исторически сложилось так, что развитие пирамид Фибоначчи было стимулировано наблюдением, согласно которому в алгоритме Дейкстры процедура DECREASE_KEY обычно вызывается намного чаще, чем процедура EXTRACT_MIN, поэтому любой метод, уменьшающий амортизированное время каждой операции DECREASE_KEY до величины $o(\lg V)$, не увеличивая при этом амортизированное время операции EXTRACT_MIN, позволяет получить реализацию, которая в асимптотическом пределе работает быстрее, чем реализация с помощью бинарных пирамид.

Алгоритм Дейкстры имеет некоторую схожесть как с поиском в ширину (см. раздел 22.2), так и с алгоритмом Прима, предназначенным для построения минимальных остовных деревьев (см. раздел 23.2). Поиск в ширину он напоминает в том отношении, что множество S соответствует множеству черных вершин, используемых при поиске в ширину; точно так же, как вершинам множества S сопоставляются конечные веса кратчайших путей, так и черным вершинам при поиске в ширину сопоставляются правильные расстояния. На алгоритм Прима алгоритм Дейкстры похож в том отношении, что в обоих этих алгоритмах с помощью неубывающей очереди с приоритетами находится “самая легкая” вершина за пределами заданного множества (в алгоритме Дейкстры это множество S, а в алгоритме Прима — “выращиваемое” дерево), затем эта вершина добавляется в множество, после чего соответствующим образом корректируются и упорядочиваются веса оставшихся за пределами множества вершин.

Упражнения

24.3-1. Выполните алгоритм Дейкстры над ориентированным графом, изображенным на рис. 24.2. Рассмотрите два варианта задачи, в одном из которых истоком является вершина s, а в другом — вершина z. Используя в качестве образца рис. 24.6, укажите, чему будут равны значения d и π, и какие вершины будут входить в множество S после каждой итерации цикла **while**.

24.3-2. Приведите простой пример ориентированного графа с отрицательными весами ребер, для которого алгоритм Дейкстры дает неправильные ответы. Почему доказательство теоремы 24.6 не годится, если веса ребер могут быть отрицательными?

24.3-3. Предположим, что строка 4 алгоритма Дейкстры изменена следующим образом:

4 **while** $|Q| > 1$

В результате этого изменения цикл **while** выполняется $|V| - 1$ раз, а не $|V|$ раз. Корректен ли предложенный алгоритм?

24.3-4. Пусть дан ориентированный граф $G = (V, E)$, с каждым ребром $(u, v) \in E$ которого связано значение $r(u, v)$, являющееся действительным числом в интервале $0 \leqslant r(u, v) \leqslant 1$, и представляющее надежность соединительного кабеля между вершиной u и вершиной v. Величина $r(u, v)$ интерпретируется как вероятность того, что в канале, соединяющем вершины u и v, не произойдет сбой; при этом предполагается, что все вероятности независимы. Сформулируйте эффективный алгоритм, позволяющий найти наиболее надежный путь, соединяющий две заданные вершины.

24.3-5. Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \{1, 2, \ldots, W\}$, где W — некоторое положительное целое число. Ни для каких двух вершин кратчайшие пути, ведущие из истока s в эти вершины, не имеют одинаковые веса. Предположим, что определен невзвешенный ориентированный граф $G' = (V \cup V', E')$, в котором каждое ребро $(u, v) \in E$ заменяется $w(u, v)$ последовательными ребрами единичного веса. Сколько вершин содержит граф G'? Теперь предположим, что в графе G' выполняется поиск в ширину. Покажите, что порядок, в котором вершины множества V окрашиваются в черный цвет при поиске в ширину в графе G', совпадает с порядком извлечения вершин множества V из очереди с приоритетами в строке 5 при выполнении алгоритма DIJKSTRA над графом G.

24.3-6. Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \{0, 1, \ldots, W\}$, где W — некоторое целое неотрицательное число. Модифицируйте алгоритм Дейкстры таким образом, чтобы он

находил кратчайшие пути из заданной вершины s в течение времени $O(WV + E)$.

24.3-7. Модифицируйте алгоритм из упражнения 24.3-6 таким образом, чтобы он выполнялся за время $O((V + E) \lg W)$. (*Указание*: сколько различных оценок кратчайших путей для вершин из множества $V - S$ может встретиться одновременно?)

24.3-8. Предположим, что имеется взвешенный ориентированный граф $G = (V, E)$, в котором веса ребер, исходящих из некоторого истока s, могут быть отрицательными, веса всех других ребер неотрицательные, и циклы с отрицательными весами отсутствуют. Докажите, что в таком графе алгоритм Дейкстры корректно находит кратчайшие пути из истока s.

24.4 Разностные ограничения и кратчайшие пути

В главе 29 изучается общая задача линейного программирования, в котором нужно оптимизировать линейную функцию, удовлетворяющую системе линейных неравенств. В этом разделе исследуется частный случай задачи линейного программирования, который сводится к поиску кратчайших путей из одной вершины. Полученную в результате задачу о кратчайших путях из одной вершины можно решить с помощью алгоритма Беллмана-Форда, решив таким образом задачу линейного программирования.

Линейное программирование

В общей ***задаче линейного программирования*** (linear-programming problem) задается матрица A размером $m \times n$, m-компонентный вектор b и n-компонентный вектор c. Нужно найти состоящий из n элементов вектор x, максимизирующий ***целевую функцию*** (objective function) $\sum_{i=1}^{n} c_i x_i$, на которую накладывается m ограничений $Ax \leqslant b$.

Несмотря на то, что время работы симплекс-алгоритма, который рассматривается в главе 29, не всегда выражается полиномиальной функцией от размера входных данных, существуют другие алгоритмы линейного программирования с полиномиальным временем работы. Имеется ряд причин, по которым важно понимать, как устроены задачи линейного программирования. Во-первых, если известно, что данная задача приводится к задаче линейного программирования с полиномиальным размером, то отсюда непосредственно следует, что для такой задачи существует алгоритм с полиномиальным временем работы. Во-вторых, имеется большое количество частных случаев задач линейного программирования, для которых существуют более производительные алгоритмы. Например,

в настоящем разделе показано, что задача о кратчайших путях из заданного истока — именно такой частный случай. К другим задачам, которые можно привести к задачам линейного программирования, относятся задача о кратчайшем пути между парой заданных вершин (упражнение 24.4-4) и задача о максимальном потоке (упражнение 26.1-8).

Иногда не имеет значения, какой должна получиться целевая функция; достаточно найти произвольное ***допустимое решение*** (feasible solution), т.е. любой вектор x, удовлетворяющий неравенству $Ax \leqslant b$, или определить, что допустимых решений не существует. Сосредоточим внимание на таких ***задачах существования*** (feasibility problem).

Системы разностных ограничений

В ***системе разностных ограничений*** (system of difference constraints) каждая строка в матрице линейного программирования A содержит одно значение 1 и одно значение -1, а все прочие элементы в этой строке равны 0. Другими словами, ограничения, заданные системой неравенств $Ax \leqslant b$, представляют собой систему m ***разностных ограничений*** (difference constraints), содержащих n неизвестных. Каждое ограничение в этой системе — обычное линейное неравенство вида

$$x_j - x_i \leqslant b_k,$$

где $1 \leqslant i, j \leqslant n$ и $1 \leqslant k \leqslant m$.

Например, рассмотрим задачу поиска 5-компонентного вектора $x = (x_i)$, удовлетворяющего системе неравенств

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leqslant \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}.$$

Эта задача эквивалентна поиску неизвестных величин x_i при $i = 1, 2, \ldots, 5$, удовлетворяющих восьми приведенным ниже ограничениям:

$$x_1 - x_2 \leqslant 0, \tag{24.3}$$
$$x_1 - x_5 \leqslant -1, \tag{24.4}$$
$$x_2 - x_5 \leqslant 1, \tag{24.5}$$
$$x_3 - x_1 \leqslant 5, \tag{24.6}$$
$$x_4 - x_1 \leqslant 4, \tag{24.7}$$
$$x_4 - x_3 \leqslant -1, \tag{24.8}$$
$$x_5 - x_3 \leqslant -3, \tag{24.9}$$
$$x_5 - x_4 \leqslant -3. \tag{24.10}$$

Одно из решений этой задачи имеет вид $x = (-5, -3, 0, -1, -4)$, что можно проверить прямой подстановкой. На самом деле эта задача имеет множество решений. Например, еще одно решение — $x' = (0, 2, 5, 4, 1)$. Эти два решения взаимосвязаны: разность между любой парой соответствующих компонентов векторов x' и x равна 5. Этот факт — не простое совпадение.

Лемма 24.8. Пусть $x = (x_1, x_2, \ldots, x_n)$ — решение системы разностных ограничений $Ax \leqslant b$, а d — произвольная константа. Тогда $x + d = (x_1 + d, x_2 + d, \ldots, x_n + d)$ — тоже решение системы $Ax \leqslant b$.

Доказательство. Для каждой пары переменных x_i и x_j можно записать соотношение $(x_i + d) - (x_j + d) = x_i - x_j$. Таким образом, если вектор x удовлетворяет системе неравенств $Ax \leqslant b$, то ей удовлетворяет и вектор $x + d$. ■

Системы разностных ограничений возникают в самых разнообразных приложениях. Например, неизвестные величины x_i могут обозначать моменты времени, в которые происходят события. Каждое ограничение можно рассматривать как требование, при котором между двумя событиями должно пройти некоторое время, не меньшее (или не большее) некоторой заданной величины. Возможно, эти события — задания, которые необходимо выполнить в процессе сборки некоторого изделия. Например, если в момент времени x_1 применяется клей, время фиксации которого — 2 часа, а деталь будет устанавливаться в момент времени x_2, то необходимо наложить ограничение $x_2 \geqslant x_1 + 2$ или, что эквивалентно, $x_1 - x_2 \leqslant -2$. При другой технологии может потребоваться, чтобы деталь устанавливалась после применения клея, но не позже, чем когда он “схватится” наполовину. В этом случае получаем пару ограничений $x_2 \geqslant x_1$ и $x_2 \leqslant x_1 + 1$, или, что эквивалентно, $x_1 - x_2 \leqslant 0$ и $x_2 - x_1 \leqslant 1$.

Графы ограничений

Системы разностных ограничений удобно рассматривать с точки зрения теории графов. Идея заключается в том, что в системе разностных ограничений $Ax \leqslant b$ матрицу задачи линейного программирования A размерами $m \times n$ можно рассматривать как транспонированную матрицу инцидентности (см. упражнение 22.1-7) графа с n вершинами и m ребрами. Каждая вершина v_i графа при $i = 1, 2, \ldots, n$ соответствует одной из n неизвестных переменных x_i. Каждое ориентированное ребро графа соответствует одному из m независимых неравенств, в которое входят две неизвестных величины.

Если выражаться формальным языком, то заданной системе разностных ограничений $Ax \leqslant b$ сопоставляется ***граф ограничений*** (constraint graph), представляющий собой взвешенный ориентированный граф $G = (V, E)$, в котором

$$V = \{v_0, v_1, \ldots, v_n\}$$

и

$$E = \{(v_i, v_j) : x_j - x_i \leqslant b_k\} \cup \{(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_n)\}.$$

Вскоре станет понятно, что дополнительная вершина v_0 вводится для того, чтобы из нее гарантированно была достижима любая другая вершина. Таким образом, множество V состоит из вершин v_i, каждая из которых соответствует неизвестной x_i, и дополнительной вершины v_0. В множестве ребер E на каждое разностное ограничение приходится по одному ребру; кроме того, в это множество входят ребра (v_0, v_i), каждое из которых соответствует неизвестной величине x_i. Если накладывается разностное ограничение $x_j - x_i \leqslant b_k$, это означает, что вес ребра (v_i, v_j) равен $w(v_i, v_j) = b_k$. Вес каждого ребра, исходящего из вершины v_0, равен 0. На рис. 24.8 приведен граф ограничений для системы разносных ограничений (24.3)–(24.10). В каждой вершине v_i указано значение $\delta(v_0, v_i)$. Допустимое решение этой системы имеет вид $x = (-5, -3, 0, -1, -4)$.

Из приведенной ниже теоремы видно, что решение системы разностных ограничений сводится к определению веса кратчайшего пути в соответствующем графе ограничений.

Теорема 24.9. Пусть $G = (V, E)$ — граф ограничений, соответствующий заданной системе разностных ограничений $Ax \leqslant b$. Если граф G не содержит циклов с отрицательным весом, то вектор

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n)) \tag{24.11}$$

является допустимым решением системы. Если граф G содержит циклы с отрицательным весом, то допустимых решений не существует.

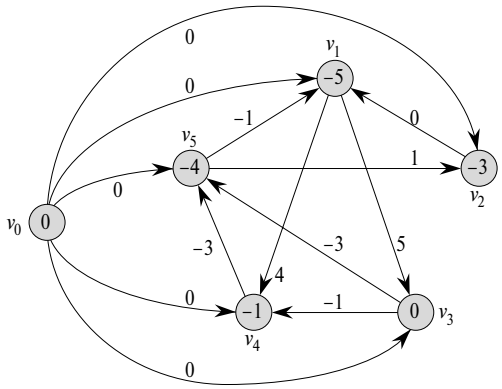


Рис. 24.8. Граф ограничений, соответствующих системе разностных ограничений (24.3)–(24.10)

Доказательство. Сначала покажем, что если граф ограничений не содержит циклов с отрицательным весом, то уравнение (24.11) определяет допустимое решение. Рассмотрим произвольное ребро $(v_i, v_j) \in E$. Согласно неравенству треугольника, выполняется неравенство $\delta(v_0, v_j) \leqslant \delta(v_0, v_i) + w(v_i, v_j)$ или, что то же самое, $\delta(v_0, v_j) - \delta(v_0, v_i) \leqslant w(v_i, v_j)$. Таким образом, полагая $x_i = \delta(v_0, v_i)$ и $x_j = \delta(v_0, v_j)$, мы удовлетворяем разностное ограничение $x_j - x_i \leqslant w(v_i, v_j)$, соответствующее ребру (v_i, v_j).

Теперь покажем, что если граф ограничений содержит цикл с отрицательным весом, то система разностных ограничений не имеет допустимых решений. Не теряя общности, предположим, что цикл с отрицательным весом — это цикл $c = \langle v_1, v_2, \ldots, v_k \rangle$, где $v_1 = v_k$. (Вершина v_0 не может принадлежать циклу c, потому что в нее не входит ни одно ребро.) Цикл c соответствует приведенной ниже системе ограничений:

$$\begin{aligned} x_2 - x_1 &\leqslant w(v_1, v_2), \\ x_3 - x_2 &\leqslant w(v_2, v_3), \\ &\vdots \\ x_{k-1} - x_{k-2} &\leqslant w(v_{k-2}, v_{k-1}), \\ x_k - x_{k-1} &\leqslant w(v_{k-1}, v_k). \end{aligned}$$

Предположим, существует решение x, удовлетворяющее всем этим k неравенствам. Это решение должно также удовлетворять неравенству, полученному в результате суммирования всех перечисленных k неравенств. Если просуммировать все левые части, то каждая неизвестная x_i один раз войдет со знаком "+" и один раз — со знаком "−" (напомним, что из равенства $v_1 = v_k$ следует равенство

$x_1 = x_k$). Это означает, что сумма всех левых частей равна 0. Сумма правых частей равна $w(c)$, откуда получается неравенство $0 \leqslant w(c)$. Однако поскольку c — цикл с отрицательным весом, то выполняется неравенство $w(c) < 0$, что приводит нас к противоречию $0 \leqslant w(c) < 0$. ■

Решение систем разностных ограничений

Теорема 24.9 говорит о том, что с помощью алгоритма Беллмана-Форда можно решить систему линейных ограничений. Поскольку в графе ограничений существуют ребра, ведущие из вершины v_0 во все другие вершины, любой цикл с отрицательным весом в графе ограничений достижим из вершины v_0. Если алгоритм Беллмана-Форда возвращает значение TRUE, веса кратчайших путей образуют допустимое решение этой системы. В примере, проиллюстрированном на рис. 24.8, веса кратчайших путей образуют допустимое решение $x = (-5, -3, 0, -1, -4)$, и, согласно лемме 24.8, $x = (d-5, d-3, d, d-1, d-4)$, где d — произвольная константа, — тоже является допустимым решением. Если алгоритм Беллмана-Форда возвращает значение FALSE, то система разностных ограничений не имеет решений.

Система разностных ограничений с m ограничениями и n неизвестными порождает граф, состоящий из $n+1$ вершин и $n+m$ ребер. Таким образом, с помощью алгоритма Беллмана-Форда такую систему можно решить в течение времени $O((n+1)(n+m)) = O(n^2 + nm)$. В упражнении 24.4-5 предлагается модифицировать этот алгоритм таким образом, чтобы он выполнялся в течение времени $O(nm)$, даже если m намного меньше, чем n.

Упражнения

24.4-1. Найдите допустимое решение приведенной ниже системы разностных ограничений или определите, что решений не существует:

$$\begin{aligned}
x_1 - x_2 &\leqslant 1,\\
x_1 - x_4 &\leqslant -4,\\
x_2 - x_3 &\leqslant 2,\\
x_2 - x_5 &\leqslant 7,\\
x_2 - x_6 &\leqslant 5,\\
x_3 - x_6 &\leqslant 10,\\
x_4 - x_2 &\leqslant 2,\\
x_5 - x_1 &\leqslant -1,\\
x_5 - x_4 &\leqslant 3,\\
x_6 - x_3 &\leqslant -8.
\end{aligned}$$

24.4-2. Найдите допустимое решение приведенной ниже системы разностных ограничений или определите, что решений не существует:

$$\begin{aligned}
x_1 - x_2 &\leqslant 4,\\
x_1 - x_5 &\leqslant 5,\\
x_2 - x_4 &\leqslant -6,\\
x_3 - x_2 &\leqslant 1,\\
x_4 - x_1 &\leqslant 3,\\
x_4 - x_3 &\leqslant 5,\\
x_4 - x_5 &\leqslant 10,\\
x_5 - x_3 &\leqslant -4,\\
x_5 - x_4 &\leqslant -8.
\end{aligned}$$

24.4-3. Может ли вес кратчайшего пути, исходящего из новой вершины v_0 в графе ограничений, быть положительным? Объясните свой ответ.

24.4-4. Сформулируйте задачу о кратчайшем пути между парой заданных вершин в виде задачи линейного программирования.

24.4-5. Покажите, как можно модифицировать алгоритм Беллмана-Форда, чтобы он позволял решить систему разностных ограничений с m неравенствами и n неизвестными в течение времени $O(nm)$.

24.4-6. Предположим, что в дополнение к системе разностных ограничений накладываются ***ограничения-равенства*** (equality constraints), имеющие вид $x_i = x_j + b_k$. Покажите, как адаптировать алгоритм Беллмана-Форда, чтобы с его помощью можно было решать системы ограничений такого вида.

24.4-7. Покажите, как можно решить систему разностных ограничений с помощью алгоритма, подобного алгоритму Беллмана-Форда, который обрабатывает граф ограничений, не содержащий дополнительную вершину v_0.

⋆ 24.4-8. Пусть $Ax \leqslant b$ — система m разностных ограничений с n неизвестными. Покажите, что при обработке графа ограничений, соответствующего этой системе, алгоритм Беллмана-Форда максимизирует сумму $\sum_{i=1}^{n} x_i$, где вектор x удовлетворяет системе $Ax \leqslant b$, а все компоненты x_i — неравенству $x_i \leqslant 0$.

⋆ 24.4-9. Покажите, что в процессе обработки графа ограничений, соответствующего системе разностных ограничений $Ax \leqslant b$, алгоритм Беллмана-Форда минимизирует величину $(\max\{x_i\} - \min\{x_i\})$, где x_i — компоненты вектора x, удовлетворяющего системе $Ax \leqslant b$. Объясните, как можно использовать этот факт, если алгоритм применяется для составления расписания строительных работ.

24.4-10. Предположим, что каждая строка матрицы A задачи линейного программирования $Ax \leqslant b$ соответствует либо разностному ограничению, либо ограничению на одну переменную вида $x_i \leqslant b_k$ или $-x_i \leqslant b_k$. Покажите, как адаптировать алгоритм Беллмана-Форда для решения систем ограничений этого вида.

24.4-11. Разработайте эффективный алгоритм решения системы разностных ограничений $Ax \leqslant b$, в которой все элементы вектора b являются действительными числами, а все неизвестные x_i должны быть целыми числами.

⋆ 24.4-12. Разработайте эффективный алгоритм решения системы разностных ограничений $Ax \leqslant b$, в которой все элементы вектора b являются действительными числами, а определенное подмножество некоторых (но не обязательно всех) неизвестных x_i должны быть целыми числами.

24.5 Доказательства свойств кратчайших путей

Корректность аргументов, которые используются в этой главе, основана на неравенстве треугольника, свойстве верхней границы, свойстве отсутствия пути, свойстве сходимости, свойстве ослабления пути и свойстве подграфа предшествования. В начале главы эти свойства сформулированы без доказательств, которые представлены в настоящем разделе.

Неравенство треугольника

При изучении поиска в ширину (раздел 22.2) в лемме 22.1 доказано простое свойство, которым обладают кратчайшие расстояния в невзвешенных графах. Неравенство треугольника обобщает это свойство для взвешенных графов.

Лемма 24.10 (Неравенство треугольника). Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbf{R}$ и истоком s. Тогда для всех ребер $(u, v) \in E$ выполняется неравенство

$$\delta(s, v) \leqslant \delta(s, u) + w(u, v).$$

Доказательство. Предположим, что существует кратчайший путь p из истока s в вершину v. Тогда вес этого пути p не превышает веса любого другого пути из вершины s в вершину v. В частности, вес пути p не превышает веса пути, состоящего из кратчайшего пути из исходной вершины s в вершину u и ребра (u, v).

В упражнении 24.5-3 предлагается рассмотреть случай, в котором не существует кратчайшего пути из вершины s в вершину v. ■

Влияние ослабления на оценки кратчайшего пути

Приведенная ниже группа лемм описывает, какому воздействию подвергаются оценки кратчайшего пути, когда выполняется последовательность шагов ослабления ребер взвешенного ориентированного графа, инициализированного процедурой INITIALIZE_SINGLE_SOURCE.

Лемма 24.11 (Свойство верхней границы). Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbf{R}$. Пусть $s \in V$ — исток, а граф инициализирован процедурой INITIALIZE_SINGLE_SOURCE(G, s). Тогда для всех вершин $v \in V$ выполняется неравенство $d[v] \geqslant \delta(s, v)$, и этот инвариант поддерживается в ходе всей последовательности этапов ослабления ребер графа G. Более того, как только атрибут $d[v]$ достигает своей нижней границы $\delta(s, v)$, он в дальнейшем не изменяется.

Доказательство. Докажем инвариант $d[v] \geqslant \delta(s, v)$ для всех вершин $v \in V$, воспользовавшись индукцией по числу шагов ослабления.

В качестве базиса индукции воспользуемся неравенством $d[v] \geqslant \delta(s, v)$, которое, очевидно, истинно непосредственно после инициализации, поскольку $d[s] = 0 \geqslant \delta(s, s)$ (заметим, что величина $\delta(s, s) = -\infty$, если вершина s находится в цикле с отрицательным весом; в противном случае $\delta(s, s) = 0$), и из $d[v] = \infty$ следует, что $d[v] \geqslant \delta(s, v)$ для всех вершин $v \in V - \{s\}$.

В качестве шага индукции рассмотрим ослабление ребра (u, v). Согласно гипотезе индукции, для всех вершин $x \in V$ перед ослаблением выполняется неравенство $d[x] \geqslant \delta(s, x)$. Единственное значение d, которое может измениться — это значение $d[v]$. Если оно изменяется, мы имеем

$$d[v] = d[u] + w(u, v) \geqslant \delta(s, u) + w(u, v) \geqslant \delta(s, v),$$

где первое неравенство следует из гипотезы индукции, а второе — из неравенства треугольника, так что инвариант сохраняется.

Чтобы увидеть, что значение $d[v]$ не будет изменяться после того, как становится справедливым соотношение $d[v] = \delta(s, v)$, заметим, что по достижении своей нижней границы это значение больше не сможет уменьшаться, поскольку, как уже было показано, $d[v] \geqslant \delta(s, v)$. Оно также не может возрастать, поскольку ослабление не увеличивает значений d. ■

Следствие 24.12 (Свойство отсутствия пути). Предположим, что во взвешенном ориентированном графе $G = (V, E)$ с весовой функцией $w : E \to \mathbf{R}$ отсутствуют пути, соединяющие исток $s \in V$ с данной вершиной $v \in V$. Тогда после инициализации графа процедурой INITIALIZE_SINGLE_SOURCE(G, s) мы имеем $d[v] = \delta(s, v) = \infty$, и это равенство сохраняется в качестве инварианта в ходе выполнения произвольной последовательности шагов ослабления ребер графа G.

Доказательство. Согласно свойству верхней границы, всегда выполняется соотношение $\infty = \delta(s, v) \leqslant d[v]$, так что $d[v] = \infty = \delta(s, v)$. ■

Лемма 24.13. Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbf{R}$, и пусть $(u, v) \in E$. Тогда непосредственно после ослабления ребра (u, v) при помощи процедуры RELAX(u, v, w) выполняется неравенство $d[v] \leqslant d[u] + w(u, v)$.

Доказательство. Если непосредственно перед ослаблением ребра (u, v) выполняется неравенство $d[v] > d[u] + w(u, v)$, то после этой операции оно преобразуется в равенство $d[v] = d[u] + w(u, v)$. Если же непосредственно перед ослаблением выполняется неравенство $d[v] \leqslant d[u] + w(u, v)$, то в процессе ослабления ни значение $d[u]$, ни значение $d[v]$ не изменяются, так что после данной операции $d[v] \leqslant d[u] + w(u, v)$. ■

Лемма 24.14 (Свойство сходимости). Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbf{R}$, $s \in V$ — исток, а $s \rightsquigarrow u \to v$ — кратчайший путь в графе G для некоторых его вершин $u, v \in V$. Предположим, что граф G инициализирован процедурой INITIALIZE_SINGLE_SOURCE(G, s), после чего выполнена последовательность этапов ослабления, включающая вызов процедуры RELAX(u, v, w). Если в некоторый момент времени до вызова выполняется равенство $d[u] = \delta(s, u)$, то в любой момент времени после вызова справедливо равенство $d[v] = \delta(s, v)$.

Доказательство. Согласно свойству верхней границы, если в некоторый момент до ослабления ребра (u, v) выполняется равенство $d[u] = \delta(s, u)$, то оно остается справедливым и впоследствии. В частности, после ослабления ребра (u, v) имеем

$$d[v] \leqslant d[u] + w(u, v) = \delta(s, u) + w(u, v) = \delta(s, v),$$

где неравенство следует из леммы 24.13, а последнее равенство — из леммы 24.1. В соответствии со свойством верхней границы $d[v] \geqslant \delta(s, v)$, откуда можно сделать вывод о том, что $d[v] = \delta(s, u)$, и это равенство впоследствии сохраняется. ■

Лемма 24.15 (Свойство ослабления пути). Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbf{R}$, а $s \in V$ — исток. Рассмотрим произвольный кратчайший путь $p = \langle v_0, v_1, \ldots, v_k \rangle$ из истока $s = v_0$ в вершину v_k. Если граф G инициализирован процедурой INITIALIZE_SINGLE_SOURCE(G, s), а затем выполнена последовательность ослаблений ребер (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k) в указанном порядке, то после этих ослаблений и в любой момент времени впоследствии выполняется равенство $d[v_k] = \delta(s, v_k)$. Это свойство

справедливо независимо от того, производятся ли ослабления на других ребрах, включая ослабления, которые чередуются с ослаблениями ребер пути p.

Доказательство. По индукции покажем, что после ослабления на i-м ребре пути p выполняется равенство $d[v_i] = \delta(s, v_i)$. В качестве базиса примем $i = 0$; перед тем, как будет ослаблено хоть одно ребро, входящее в путь p, после процедуры инициализации очевидно, что $d[v_0] = d[s] = 0 = \delta(s, s)$. Согласно свойству верхней границы, значение $d[s]$ после инициализации больше не изменяется.

На очередном шаге индукции предполагается, что выполняется равенство $d[v_{i-1}] = \delta(s, v_{i-1})$, и рассматривается ослабление ребра (v_{i-1}, v_i). Согласно свойству сходимости, после этого ослабления выполняется равенство $d[v_i] = \delta(s, v_i)$, которое впоследствии сохраняется. ■

Ослабление и деревья кратчайших путей

Теперь покажем, что после того, как в результате последовательности ослаблений оценки кратчайших путей сойдутся к весам кратчайших путей, подграф предшествования G_π, образованный полученными значениями π, будет деревом кратчайших путей для графа G. Начнем с приведенной ниже леммы, в которой показано, что подграф предшествования всегда образует корневое дерево с корнем в истоке.

Лемма 24.16. Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbf{R}$, а $s \in V$ — исток. Предполагается также, что граф G не содержит циклов с отрицательным весом, достижимых из истока s. Тогда после инициализации графа с помощью процедуры INITIALIZE_SINGLE_SOURCE(G, s) подграф предшествования G_π образует корневое дерево с корнем s, а любая последовательность шагов ослабления ребер графа G поддерживает это свойство в качестве инварианта.

Доказательство. Вначале единственной вершиной графа G_π является исток, и лемма тривиальным образом выполняется. Рассмотрим подграф предшествования G_π, возникающий в результате последовательности этапов ослабления. Сначала докажем, что этот граф — ациклический. Чтобы получить противоречие, предположим, что после некоторого шага ослабления в графе G_π создается цикл. Пусть это цикл $c = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_k = v_0$. Тогда при $i = 1, 2, \ldots, k$ выполняется равенство $\pi[v_i] = v_{i-1}$, и без потери общности можно считать, что цикл был создан в результате ослабления ребра (v_{k-1}, v_k) графа G_π.

Утверждается, что все вершины цикла c достижимы из истока s. Почему? Для каждой вершины цикла c существует предшественник (т.е. значение соответствующего атрибута отлично от значения NIL), поэтому каждой из этих вершин

сопоставляется конечная оценка кратчайшего пути, когда ее атрибуту π присваивается значение, отличное от значения NIL. Согласно свойству верхней границы, каждой вершине цикла c соответствует вес кратчайшего пути, из чего следует, что она достижима из истока s.

Рассмотрим оценки кратчайших путей, которые относятся к циклу c, непосредственно перед вызовом процедуры RELAX(v_{k-1}, v_k, w) и покажем, что c — цикл с отрицательным весом, а это противоречит предположению о том, что в графе G не содержится циклов с отрицательным весом, достижимых из истока. Непосредственно перед вызовом для $i = 1, 2, \ldots, k-1$ выполняется равенство $\pi[v_i] = v_{i-1}$. Таким образом, при $i = 1, 2, \ldots, k-1$ последним обновлением величины $d[v_i]$ является присвоение $d[v_i] \leftarrow d[v_{i-1}] + w(v_{i-1}, v_i)$. Если после этого значение $d[v_{i-1}]$ изменяется, то оно может только уменьшаться. Поэтому непосредственно перед вызовом процедуры RELAX(v_{k-1}, v_k, w) выполняется неравенство

$$d[v_i] \geqslant d[v_{i-1}] + w(v_{i-1}, v_i) \quad \text{для всех } i = 1, 2, \ldots, k-1. \tag{24.12}$$

Поскольку величина $\pi[v_k]$ в результате вызова изменяется, непосредственно перед этим выполняется строгое неравенство

$$d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k).$$

Суммируя это строгое неравенство с $k-1$ неравенствами (24.12), получим сумму оценок кратчайшего пути вдоль цикла c:

$$\begin{aligned}\sum_{i=1}^{k} d[v_i] &> \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) = \\ &= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).\end{aligned}$$

Однако

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}],$$

поскольку каждая вершина цикла c входит в каждую сумму ровно по одному разу. Из этого равенства следует

$$0 > \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Таким образом, суммарный вес цикла c — величина отрицательная, что и приводит к желаемому противоречию.

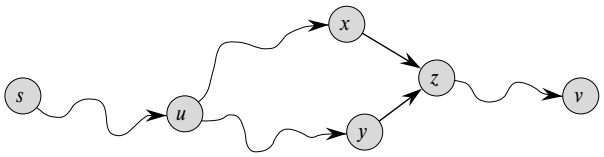


Рис. 24.9. Иллюстрация того, что путь в графе G_π из истока s в вершину v — единственный

Итак, доказано, что G_π — ориентированный ациклический граф. Чтобы показать, что он образует корневое дерево с корнем s, осталось доказать (см. упражнение Б.5-2), что для каждой вершины $v \in V$ в графе G_π имеется единственный путь из истока s в вершину v.

Сначала необходимо показать, что из истока s существует путь в каждую вершину множества V_π. В это множество входят вершины, значение атрибута π которых отлично от значения NIL, а также вершина s. Идея заключается в том, чтобы доказать наличие такого пути из истока s в каждую вершину множества V_π по индукции. Детали предлагается рассмотреть в упражнении 24.5-6.

Чтобы завершить доказательство леммы, теперь нужно показать, что для любой вершины $v \in V_\pi$ в графе G_π существует не более одного пути из вершины s в вершину v. Предположим обратное. Другими словами, предположим, что из истока s существует два простых пути в некоторую вершину v: путь p_1, который можно разложить как $s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$, и путь p_2, который можно разложить как $s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$, где $x \neq y$ (рис. 24.9). Однако в таком случае выполняются равенства $\pi[z] = x$ и $\pi[z] = y$, откуда следует противоречие $x = y$. Приходим к выводу, что в графе G_π существует единственный путь из истока s в вершину v, и, следовательно, этот граф образует корневое дерево с корнем s. ■

Теперь можно показать, что если после некоторой последовательности этапов ослабления всем вершинам присвоены истинные веса кратчайших путей, то подграф предшествования G_π — это дерево кратчайших путей.

Лемма 24.17 (Свойство подграфа предшествования). Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \rightarrow \mathbf{R}$, а $s \in V$ — исток. Предположим также, что граф G не содержит циклов с отрицательным весом, достижимых из вершины s. Вызовем процедуру INITIALIZE_SINGLE_SOURCE(G, s), после чего выполним произвольную последовательность шагов ослабления ребер графа G, в результате которой для всех вершин $v \in V$ выполняется равенство $d[v] = \delta(s, v)$. Тогда подграф предшествования G_π — дерево кратчайших путей с корнем s.

Доказательство. Необходимо доказать, что для графа G_π выполняются все три свойства, сформулированные в разделе “Представление кратчайших путей” для

деревьев кратчайших путей. Чтобы доказать первое свойство, необходимо показать, что V_π — множество вершин, достижимых из истока s. По определению вес кратчайшего пути $\delta(s, v)$ выражается конечной величиной тогда и только тогда, когда вершина v достижима из истока s, поэтому из истока s достижимы только те вершины, которым соответствуют конечные значения d. Однако величине $d[v]$, соответствующей вершине $v \in V - \{s\}$, конечное значение присваивается тогда и только тогда, когда $\pi[v] \neq$ NIL. Таким образом, в множество V_π входят только те вершины, которые достижимы из истока s.

Второе свойство следует непосредственно из леммы 24.16.

Поэтому остается доказать справедливость последнего свойства для дерева кратчайших путей: для каждой вершины $v \in V_\pi$ единственный простой путь $s \overset{p}{\rightsquigarrow} v$ в графе G_π — это кратчайший путь в графе G из истока s в вершину v. Пусть $p = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_0 = s$ и $v_k = v$. При $i = 1, 2, \ldots, k$ выполняются соотношения $d[v_i] = \delta(s, v_i)$ и $d[v_i] \geqslant d[v_{i-1}] + w(v_{i-1}, v_i)$, из чего следует, что $w(v_{i-1}, v_i) \leqslant \delta(s, v_i) - \delta(s, v_{i-1})$. В результате суммирования весов вдоль пути p получаем

$$\begin{aligned} w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \leqslant \\ &\leqslant \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) = \\ &= \delta(s, v_k) - \delta(s, v_0) = \\ &= \delta(s, v_k), \end{aligned}$$

где предпоследнее равенство следует из "телескопичности" суммы разностей, а последнее — из того, что $\delta(s, v_0) = \delta(s, s) = 0$.

Таким образом, справедливо неравенство $w(p) \leqslant \delta(s, v_k)$. Поскольку $\delta(s, v_k)$ — нижняя граница веса, которым обладает любой путь из истока s в вершину v_k, приходим к выводу, что $w(p) = \delta(s, v_k)$, и поэтому p — кратчайший путь из истока s в вершину $v = v_k$. ■

Упражнения

24.5-1. Для ориентированного графа, изображенного на рис. 24.2, приведите пример двух деревьев кратчайших путей, отличных от показанных.

24.5-2. Приведите пример взвешенного ориентированного графа $G = (V, E)$ с весовой функцией $w : E \to \mathbf{R}$ и истоком s, для которого выполняется сформулированное ниже свойство: для каждого ребра $(u, v) \in E$ существует дерево кратчайших путей с корнем s, содержащее ребро (u, v),

и другое дерево кратчайших путей с корнем s, в котором это ребро отсутствует.

24.5-3. Усовершенствуйте доказательство леммы 24.10, чтобы она охватывала случаи, когда веса кратчайших путей равны ∞ и $-\infty$.

24.5-4. Пусть $G = (V, E)$ — взвешенный ориентированный граф с исходной вершиной s, инициализированный процедурой INITIALIZE_SINGLE_SOURCE(G, s). Докажите, что если в результате выполнения последовательности этапов ослабления атрибуту $\pi[s]$ присваивается значение, отличное от NIL, то граф G содержит цикл с отрицательным весом.

24.5-5. Пусть $G = (V, E)$ — взвешенный ориентированный граф, не содержащий ребер с отрицательным весом, и пусть $s \in V$ — исток. Предположим, что в качестве $\pi[v]$ может выступать предшественник вершины v на *любом* кратчайшем пути к вершине v из истока s, если вершина $v \in V - \{s\}$ достижима из вершины s; в противном случае $\pi[v]$ принимает значение NIL. Приведите пример такого графа G и значений, которые следует присвоить атрибутам $\pi[v]$, чтобы в графе G_π образовался цикл. (Согласно лемме 24.16, такое присвоение не может быть получено в результате последовательности этапов ослабления.)

24.5-6. Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbf{R}$, не содержащий циклов с отрицательным весом. Пусть $s \in V$ — исток, а граф G инициализируется процедурой INITIALIZE_SINGLE_SOURCE(G, s). Докажите, что для каждой вершины $v \in V_\pi$ в графе G_π существует путь из вершины s в вершину v и что это свойство поддерживается как инвариант в ходе произвольной последовательности ослаблений.

24.5-7. Пусть $G = (V, E)$ — взвешенный ориентированный граф, не содержащий циклов с отрицательным весом. Пусть также $s \in V$ — исток, а граф G инициализируется процедурой INITIALIZE_SINGLE_SOURCE(G, s). Докажите, что существует такая последовательность $|V| - 1$ этапов ослабления, что для всех вершин $v \in V$ выполняется равенство $d[v] = \delta(s, v)$.

24.5-8. Пусть G — произвольный взвешенный ориентированный граф, который содержит цикл с отрицательным весом, достижимый из истока s. Покажите, что всегда можно построить такую бесконечную последовательность этапов ослабления ребер графа G, что каждое ослабление будет приводить к изменению оценки кратчайшего пути.

Задачи

24-1. **Улучшение Йена алгоритма Беллмана-Форда**

Предположим, что в каждом проходе алгоритма Беллмана-Форда ослабления ребер упорядочены следующим образом. Перед первым проходом вершины входного графа $G = (V, E)$ выстраиваются в произвольном порядке $v_1, v_2, \ldots, v_{|V|}$. Затем множество ребер E разбивается на множества $E_f \cup E_b$, где $E_f = \{(v_i, v_j) \in E : i < j\}$ и $E_b = \{(v_i, v_j) \in E : i > j\}$. (Предполагается, что граф G не содержит петель, так что каждое ребро принадлежит либо множеству E_f, либо множеству E_b.) Определим графы $G_f = (V, E_f)$ и $G_b = (V, E_b)$.

а) Докажите, что G_f — ациклический граф с топологической сортировкой $\langle v_1, v_2, \ldots, v_{|V|} \rangle$, а G_b — ациклический граф с топологической сортировкой $\langle v_{|V|}, v_{|V|-1}, \ldots, v_1 \rangle$.

Предположим, что каждый проход Беллмана-Форда реализуется следующим образом. Сначала вершины обходятся в порядке $v_1, v_2, \ldots, v_{|V|}$, и при этом ослабляется каждое ребро графа E_f, исходящее из посещаемой вершины. Затем вершины обходятся в порядке $v_{|V|}, v_{|V|-1}, \ldots, v_1$, и при этом ослабляется каждое ребро графа E_b, исходящее из этой вершины.

б) Докажите, что при такой схеме, если граф G не содержит циклов с отрицательным весом, достижимых из истока s, то после $\lceil |V|/2 \rceil$ проходов по ребрам $d[v] = \delta(s, v)$ для всех вершин $v \in V$.

в) Улучшает ли эта схема асимптотическое поведение времени работы алгоритма Беллмана-Форда?

24-2. **Вложенные ящики**

Говорят, что d-мерный ящик с размерами $(x_1, x_2, \ldots, x_d)$ ***вкладывается*** (nests) в другой ящик с размерами $(y_1, y_2, \ldots, y_d)$, если существует такая перестановка π индексов $\{1, 2, \ldots, d\}$, что выполняются неравенства $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \ldots, x_{\pi(d)} < y_d$.

а) Докажите, что вложение обладает свойством транзитивности.

б) Разработайте эффективный метод определения, вкладывается ли один d-мерный ящик в другой.

в) Предположим, что задано множество, состоящее из n d-мерных ящиков $\{B_1, B_2, \ldots, B_n\}$. Разработайте эффективный алгоритм, позволяющий найти самую длинную последовательность ящиков $\langle B_{i_1}, B_{i_2}, \ldots, B_{i_k} \rangle$ такую, что для $j = 1, 2, \ldots, k-1$ ящик B_{i_j} вкладывается в ящик $B_{i_{j+1}}$. Выразите время работы этого алгоритма через n и d.

24-3. Арбитражные операции

Арбитражные операции (arbitrage) — это использование различий в текущем курсе валют для преобразования единицы валюты в большее количество единиц этой же валюты. Например, предположим, что за 1 американский доллар можно купить 46.4 индийских рупий, за одну индийскую рупию можно купить 2.5 японских иен, а за одну японскую иену — 0.0091 американских долларов. В этом случае в результате ряда операций обмена торговец может начать с одного американского доллара и купить $46.4 \times 2.5 \times 0.0091 = 1.0556$ американских долларов, получив таким образом доход в размере 5.56%.

Предположим, что даны n валют $c_1, c_2, \ldots, c_n$ и таблица R размером $n \times n$, содержащая обменные курсы, т.е. за одну единицу валюты c_i можно купить $R[i, j]$ единиц валюты c_j.

а) Разработайте эффективный алгоритм, позволяющий определить, существует ли последовательность валют $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$, такая что

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1.$$

Проанализируйте время работы этого алгоритма.

б) Разработайте эффективный алгоритм поиска такой последовательности, если она существует. Проанализируйте время работы этого алгоритма.

24-4. Алгоритм Габова для масштабирования кратчайших путей из одной вершины

Алгоритм ***масштабирования*** (scaling) решает задачу, сначала рассматривая только старший бит каждой соответствующей входной величины (такие как вес ребра). Затем начальное решение улучшается за счет рассмотрения двух старших битов. После этого последовательно рассматривается все большее число (старших) битов, в результате чего каждый раз решение улучшается до тех пор, пока не будут рассмотрены все биты, и не будет найдено правильное решение.

В этой задаче исследуется алгоритм поиска кратчайших путей из одной вершины путем масштабирования весов ребер. Нам дан ориентированный граф $G = (V, E)$, веса ребер в котором выражаются неотрицательными целыми величинами w. Введем величину $W = \max_{(u,v) \in E} \{w(u, v)\}$. Наша цель — разработать алгоритм, время работы которого составляет $O(E \lg W)$. Предполагается, что все вершины достижимы из истока.

В алгоритме последовательно по одному обрабатываются биты, образующие бинарное представление весов ребер, начиная с самого старшего к самому младшему. В частности, пусть $k = \lceil \lg(W + 1) \rceil$ — количество битов в бинарном представлении величины W, и пусть $w_i(u, v) =$

$= \lfloor w(u,v)/2^{k-i} \rfloor$, где $i = 1, 2, \ldots, k$. Другими словами, $w_i(u,v)$ — "масштабированная" версия $w(u,v)$, полученная из i старших битов этой величины. (Таким образом, для всех ребер $(u,v) \in E$ $w_k(u,v) = w(u,v)$.) Например, если $k = 5$ и $w(u,v) = 25$ (бинарное представление которого имеет вид $\langle 11001 \rangle$), то $w_3(u,v) = \langle 110 \rangle = 6$. Другой пример с $k = 5$: если $w(u,v) = \langle 00100 \rangle = 4$, то $w_3(u,v) = \langle 001 \rangle = 1$. Определим величину $\delta_i(u,v)$ — вес кратчайшего пути из вершины u в вершину v, вычисленный с помощью весовой функции w_i. Таким образом, для всех вершин $u, v \in V$ $\delta_k(u,v) = \delta(u,v)$. Для заданного истока s в алгоритме масштабирования для всех вершин $v \in V$ сначала вычисляются веса кратчайших путей $\delta_1(s,v)$, затем для всех вершин вычисляются величины $\delta_2(s,v)$ и т.д., пока для всех вершин $v \in V$ не будут вычислены величины $\delta_k(s,v)$. Предполагается, что $|E| \geqslant |V| - 1$. Как вы увидите, вычисление величины δ_i на основании δ_{i-1} требует $O(E)$ времени, так что время работы всего алгоритма — $O(kE) = O(E \lg W)$.

а) Предположим, что для всех вершин $v \in V$ выполняется неравенство $\delta(s,v) \leqslant |E|$. Покажите, что величину $\delta(s,v)$ для всех вершин можно вычислить в течение времени $O(E)$.

б) Покажите, что величину $\delta_1(s,v)$ для всех вершин $v \in V$ можно вычислить в течение времени $O(E)$.

Теперь сосредоточим внимание на вычислении величины δ_i, зная величину δ_{i-1}.

в) Докажите, что при $i = 2, 3, \ldots, k$ выполняется либо равенство $w_i(u,v) = 2w_{i-1}(u,v)$, либо равенство $w_i(u,v) = 2w_{i-1}(u,v) + 1$. Затем докажите, что для всех вершин $v \in V$ справедливы неравенства

$$2\delta_{i-1}(s,v) \leqslant \delta_i(s,v) \leqslant 2\delta_{i-1}(s,v) + |V| - 1.$$

г) Определим для $i = 2, 3, \ldots, k$ и всех ребер $(u,v) \in E$ величину

$$\widehat{w}_i(u,v) = w_i(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v).$$

Докажите, что для $i = 2, 3, \ldots, k$ и всех $u, v \in V$ значение $\widehat{w}_i(u,v)$ представляет собой неотрицательное целое число.

д) Теперь определим величину $\widehat{\delta}_i(s,v)$ как вес кратчайшего пути из истока s в вершину v, полученного с помощью весовой функции $\widehat{w}_i$. Докажите, что для $i = 2, 3, \ldots, k$ и всех вершин $v \in V$ выполняется равенство

$$\delta_i(s,v) = \widehat{\delta}_i(s,v) + 2\delta_{i-1}(s,v)$$

и что $\widehat{\delta}_i(s,v) \leqslant |E|$.

е) Покажите, как вычислить величину $\delta_i(s, v)$ на основе величины $\delta_{i-1}(s, v)$ для всех вершин $v \in V$ в течение времени $O(E)$ и обоснуйте вывод о том, что вычислить $\delta(s, v)$ для всех вершин $v \in V$ можно за время $O(E \lg W)$.

24-5. **Алгоритм Карпа для поиска цикла с минимальным средним весом**

Пусть $G = (V, E)$ — ориентированный граф с весовой функцией $w : E \to \mathbf{R}$, и пусть $n = |V|$. Определим ***средний вес*** (mean weight) цикла $c = \langle e_1, e_2, \ldots, e_k \rangle$, состоящего из ребер множества E, как

$$\mu(c) = \frac{1}{k} \sum_{i=1}^{k} w(e_i).$$

Пусть $\mu^* = \min_c \mu(c)$, где c охватывает все ориентированные циклы графа G. Цикл c, для которого выполняется равенство $\mu(c) = \mu^*$, называется ***циклом с минимальным средним весом*** (minimum mean-weight cycle). В этой задаче исследуется эффективный алгоритм вычисления величины μ^*.

Без потери общности предположим, что каждая вершина $v \in V$ достижима из истока $s \in V$. Пусть $\delta(s, v)$ — вес кратчайшего пути из истока s в вершину v, а $\delta_k(s, v)$ — вес кратчайшего пути из истока s в вершину v, содержащего *ровно* k ребер. Если такого пути не существует, то $\delta_k(s, v) = \infty$.

а) Покажите, что если $\mu^* = 0$, то граф G не содержит циклов с отрицательным весом, и $\delta(s, v) = \min_{0 \leqslant k \leqslant n-1} \delta_k(s, v)$ для всех вершин $v \in V$.

б) Покажите, что если $\mu^* = 0$, то

$$\max_{0 \leqslant k \leqslant n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \geqslant 0$$

для всех вершин $v \in V$. (*Указание*: воспользуйтесь обоими свойствами из части *а*.)

в) Пусть c — цикл с нулевым весом, а u и v — две произвольные вершины в этом цикле. Предположим, что $\mu^* = 0$, и что вес пути из вершины u в вершину v вдоль цикла равен x. Докажите, что $\delta(s, v) = \delta(s, u) + x$. (*Указание*: вес пути из вершины v в вершину u вдоль цикла равен $-x$.)

г) Покажите, что если $\mu^* = 0$, то в каждом цикле с минимальным средним весом существует вершина v такая, что

$$\max_{0 \leqslant k \leqslant n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0.$$

(*Указание*: покажите, что кратчайший путь в каждую вершину, принадлежащую циклу с минимальным средним весом, можно расширить вдоль цикла к следующей вершине цикла.)

д) Покажите, что если $\mu^* = 0$, то

$$\min_{v \in V} \max_{0 \leqslant k \leqslant n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0.$$

е) Покажите, что если к весу каждого ребра графа G добавить константу t, то величина μ^* увеличится на t. Покажите с помощью этого факта справедливость соотношения

$$\mu^* = \min_{v \in V} \max_{0 \leqslant k \leqslant n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k}.$$

ж) Разработайте алгоритм, позволяющий вычислить величину μ^* за время $O(VE)$.

24-6. **Битонические кратчайшие пути**

Последовательность называется ***битонической*** (bitonic), если она монотонно возрастает, а затем монотонно убывает, или если путем циклического сдвига ее можно привести к такому виду. Например, битоническими являются последовательности $\langle 1, 4, 6, 8, 3, -2 \rangle$, $\langle 9, 2, -4, -10, -5 \rangle$ и $\langle 1, 2, 3, 4 \rangle$, но не $\langle 1, 3, 12, 4, 2, 10 \rangle$. (См. также главу 27 и задачу 15-1.)

Предположим, что задан ориентированный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbf{R}$, и нужно найти кратчайшие пути из одной вершины s. Имеется также дополнительная информация: для каждой вершины $v \in V$ веса ребер вдоль произвольного кратчайшего пути из истока s в вершину v образуют битоническую последовательность.

Разработайте наиболее эффективный алгоритм, позволяющий решить эту задачу, и проанализируйте время его работы.

Заключительные замечания

Алгоритм Дейкстры (Dijkstra) [75] появился в 1959 году, но в нем не содержалось никаких упоминаний по поводу очереди с приоритетами. Алгоритм Беллмана-Форда основан на отдельных алгоритмах Беллмана (Bellman) [35] и Форда

(Ford) [93]. Беллман описывает, какое отношение имеют кратчайшие пути к разностным ограничениям. Лоулер (Lawler) [196] описывает алгоритм с линейным временем работы для поиска кратчайших путей в ориентированном ациклическом графе, который он рассматривает как часть "народного творчества".

Если вес каждого из ребер выражается сравнительно малыми неотрицательными целыми числами, задача о кратчайших путях из одной вершины решается с помощью более эффективных алгоритмов. Последовательность значений, возвращаемых в результате вызовов процедуры EXTRACT_MIN в алгоритме Дейкстры, монотонно возрастает со временем. Как говорилось в заключительных замечаниях к главе 6, в этом случае существует несколько структур данных, позволяющих эффективнее реализовать различные операции над очередями с приоритетами, чем бинарная пирамида или пирамида Фибоначчи. Ахуя (Ahuja), Мельхорн (Mehlhorn), Орлин (Orlin) и Таржан (Tarjan) [8] предложили для графов с неотрицательными весами ребер алгоритм со временем работы $O\left(E + V\sqrt{\lg W}\right)$, где W — максимальный вес ребра графа. Наилучшие границы достигнуты Торупом (Thorup) [299], который предложил алгоритм со временем работы $O\left(E \lg\lg V\right)$, и Раманом (Raman) [256], чей алгоритм имеет время работы $O(E + V \min\{(\lg V)^{1/3+\varepsilon}, (\lg W)^{1/4+\varepsilon}\})$. Оба алгоритма используют объем памяти, зависящий от размера слова машины, на которой выполняется алгоритм. Хотя объем использующейся памяти может оказаться неограниченным в зависимости от размера входных данных, с помощью рандомизированного хеширования его можно снизить до линейно зависящего от размера входных данных.

Для неориентированных графов с целочисленными весами Торуп [298] привел алгоритм со временем работы $O\left(V + E\right)$, предназначенный для поиска кратчайших путей из одной вершины. В отличие от алгоритмов, упомянутых в предыдущем абзаце, этот алгоритм — не реализация алгоритма Дейкстры, поскольку последовательность значений, возвращаемых вызовами процедуры EXTRACT_MIN, не является монотонно неубывающей.

Для графов, содержащих ребра с отрицательными весами, алгоритм, предложенный Габовым (Gabow) и Таржаном [104], имеет время работы $O(\sqrt{V}E \lg(VW))$, а предложенный Гольдбергом (Goldberg) [118] выполняется в течение времени $O(\sqrt{V}E \lg W)$, где $W = \max_{(u,v)\in E}\{|w\left(u,v\right)|\}$.

Черкасский (Cherkassky), Гольдберг (Goldberg) и Радзик (Radzik) [57] провели большое количество экспериментов по сравнению различных алгоритмов, предназначенных для поиска кратчайших путей.

ГЛАВА 25

Кратчайшие пути между всеми парами вершин

В этой главе рассматривается задача о поиске кратчайших путей между всеми парами вершин графа. Эта задача может возникнуть, например, при составлении таблицы расстояний между всеми парами городов, нанесенных на атлас дорог. Как и в главе 24, в этой задаче задается взвешенный ориентированный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbf{R}$, отображающей ребра на их веса, выраженные действительными числами. Для каждой пары вершин $u, v \in V$ требуется найти кратчайший (обладающий наименьшим весом) путь из вершины u в вершину v, вес которого определяется как сумма весов входящих в него ребер. Обычно выходные данные представляются в табличной форме: на пересечении строки с индексом u и столбца с индексом v расположен вес кратчайшего пути из вершины u в вершину v.

Задачу о поиске кратчайших путей между всеми парами вершин можно решить, $|V|$ раз запустив алгоритм поиска кратчайших путей из единого истока, каждый раз выбирая в качестве истока новую вершину графа. Если веса всех ребер неотрицательные, можно воспользоваться алгоритмом Дейкстры. Если используется реализация неубывающей очереди с приоритетами в виде линейного массива, то время работы такого алгоритма равно $O\left(V^3 + VE\right) = O\left(V^3\right)$. Если же неубывающая очередь с приоритетами реализована в виде бинарной неубывающей пирамиды, то время работы будет равно $O\left(V\,E \lg V\right)$, что предпочтительнее для разреженных графов. Можно также реализовать неубывающую очередь с приоритетами как пирамиду Фибоначчи; в этом случае время работы алгоритма равно $O\left(V^2 \lg V + VE\right)$.

Если допускается наличие ребер с отрицательным весом, алгоритм Дейкстры неприменим. Вместо него для каждой вершины следует выполнить алгоритм Беллмана-Форда, который работает медленнее. Полученное в результате время работы будет равно $O\left(V^2E\right)$, которое на плотных графах можно записать как $O\left(V^4\right)$. В этой главе станет понятно, как лучше поступить. Будет также исследована связь задачи о кратчайших расстояниях между всеми парами вершин с умножением матриц, и изучена алгебраическая структура этой задачи.

В отличие от алгоритмов поиска кратчайшего пути из фиксированного истока, в которых предполагается, что представление графа имеет вид списка смежных вершин, в большей части представленных в этой главе алгоритмов используется представление в виде матрицы смежности. (В алгоритме Джонсона (Johnson) для разреженных графов используются списки смежных вершин.) Для удобства предполагается, что вершины пронумерованы как $1, 2, \ldots, |V|$, поэтому в роли входных данных выступает матрица W размером $n \times n$, представляющая веса ребер ориентированного графа $G = (V, E)$ с n вершинами. Другими словами, $W = (w_{ij})$, где

$$w_{ij} = \begin{cases} 0 & \text{если } i = j, \\ \text{вес ориентированного ребра } (i,j) & \text{если } i \neq j \text{ и } (i,j) \in E, \\ \infty & \text{если } i \neq j \text{ и } (i,j) \notin E. \end{cases} \qquad (25.1)$$

Наличие ребер с отрицательным весом допускается, но пока что предполагается, что входной граф не содержит циклов с отрицательным весом.

Выходные данные представленных в этой главе алгоритмов, предназначенных для поиска кратчайших путей между всеми парами вершин, имеют вид матрицы $D = (d_{ij})$ размером $n \times n$, где элемент d_{ij} содержит вес кратчайшего пути из вершины i в вершину j. Другими словами, если обозначить через $\delta(i, j)$ кратчайший путь из вершины i в вершину j (как это было сделано в главе 24), то по завершении алгоритма $d_{ij} = \delta(i, j)$.

Чтобы решить задачу о поиске кратчайших путей между всеми парами вершин со входной матрицей смежности, необходимо вычислить не только вес каждого из кратчайших путей, но также и ***матрицу предшествования*** (predecessor matrix) $\Pi = (\pi_{ij})$, где величина π_{ij} имеет значение NIL, если $i = j$ или путь из вершины i в вершину j отсутствует; в противном случае π_{ij} — предшественник вершины j на некотором кратчайшем пути из вершины i. Точно так же, как описанный в главе 24 подграф предшествования G_π является деревом кратчайших путей для заданного истока, подграф, индуцированный i-й строкой матрицы Π, должен быть деревом кратчайших путей с корнем i. Определим для каждой вершины $i \in V$ ***подграф предшествования*** (predecessor subgraph) графа G для вершины i как граф $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, где

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$$

и

$$E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} - \{i\}\}\,.$$

Если $G_{\pi,i}$ — дерево кратчайших путей, то приведенная ниже процедура, представляющая собой модифицированную версию описанной в главе 22 процедуры PRINT_PATH, выводит кратчайший путь из вершины i в вершину j:

```
PRINT_ALL_PAIRS_SHORTEST_PATH(Π, i, j)
1  if i = j
2     then print i
3     else if π_ij = NIL
4             then print "Не существует пути из" i "в" j
5             else PRINT_ALL_PAIRS_SHORTEST_PATH(Π, i, π_ij)
6                  print j
```

Чтобы подчеркнуть важные особенности представленных в этой главе алгоритмов поиска кратчайших путей между всеми парами вершин, создание матриц предшествования и их свойств не будет рассматриваться здесь так же подробно, как это было в главе 24 в случае подграфа предшествования. Основные моменты предлагается рассмотреть в некоторых упражнениях.

Краткое содержание главы

В разделе 25.1 представлен алгоритм динамического программирования, основанный на операции умножения матриц, который позволяет решить задачу о поиске кратчайших путей между всеми парами вершин. С помощью метода "многократного возведения в квадрат" можно сделать так, чтобы время работы этого алгоритма было равно $\Theta\left(V^3 \lg V\right)$. В разделе 25.2 приведен другой алгоритм динамического программирования — алгоритм Флойда-Варшалла (Floyd-Warshall). Время работы этого алгоритма равно $\Theta\left(V^3\right)$. В этом же разделе исследуется задача поиска транзитивного замыкания ориентированного графа, связанная с задачей о поиске кратчайших путей между всеми парами вершин. Наконец, в разделе 25.3 представлен алгоритм Джонсона. В отличие от других алгоритмов, описанных в этой главе, в алгоритме Джонсона применяется представление графа в виде списка смежных вершин. Этот алгоритм позволяет решить задачу о поиске кратчайших путей между всеми парами вершин за время $O\left(V^2 \lg V + VE\right)$, что делает его пригодным для больших разрешенных графов.

Перед тем как продолжить, нужно принять некоторые соглашения для представлений в виде матрицы смежности. Во-первых, в общем случае предполагается, что входной граф $G = (V, E)$ содержит n вершин, так что $n = |V|$. Во-вторых, будет использоваться соглашение об обозначении матриц прописными буквами, например, W, L или D, а их отдельных элементов — строчными буквами с нижними индексами, например, w_{ij}, l_{ij} или d_{ij}. Возле некоторых матриц

будут взятые в скобки верхние индексы, указывающие на количество итераций, например, $L^{(m)} = \left(l_{ij}^{(m)}\right)$ или $D^{(m)} = \left(d_{ij}^{(m)}\right)$. Наконец, для заданной матрицы A размером $n \times n$ предполагается, что значение n хранятся в атрибуте $rows\,[A]$.

25.1 Задача о кратчайших путях и умножение матриц

В этом разделе представлен алгоритм динамического программирования, предназначенный для решения задачи о поиске кратчайших путей между всеми парами вершин в ориентированном графе $G = (V, E)$. В каждом основном цикле динамического программирования будет вызываться операция, очень напоминающая матричное умножение, поэтому такой алгоритм будет напоминать многократное умножение матриц. Начнем с того, что разработаем для решения задачи о кратчайших путях между всеми парами вершин алгоритм со временем работы $\Theta\left(V^4\right)$, после чего улучшим этот показатель до величины $\Theta\left(V^3 \lg V\right)$.

Перед тем как продолжить, кратко напомним описанные в главе 15 этапы разработки алгоритма динамического программирования.

1. Описание структуры оптимального решения.
2. Рекурсивное определение значения оптимального решения.
3. Вычисление значения, соответствующего оптимальному решению, пользуясь методом восходящего анализа.

(Четвертый этап, т.е. составление оптимального решения на основе полученной информации, рассматривается в упражнениях.)

Структура кратчайшего пути

Начнем с того, что охарактеризуем структуру оптимального решения. Для задачи о кратчайших путях между всеми парами вершин графа $G = (V, E)$ доказано (лемма 24.1), что все частичные пути кратчайшего пути — тоже кратчайшие пути. Предположим, что граф представлен матрицей смежности $W = (w_{ij})$. Рассмотрим кратчайший путь p из вершины i в вершину j и предположим, что этот путь содержит не более m ребер. Если циклы с отрицательным весом отсутствуют, то m конечно. Если $i = j$, то вес пути p равен 0, а ребра в нем отсутствуют. Если же вершины i и j различаются, то путь p раскладывается на $i \overset{p'}{\rightsquigarrow} k \rightarrow j$, где путь p' содержит не более $m - 1$ ребер. Согласно лемме 24.1, p' — кратчайший путь из вершины i в вершину k, поэтому выполняется равенство $\delta\,(i, j) = \delta\,(i, k) + w_{kj}$.

Рекурсивное решение задачи о кратчайших путях между всеми парами вершин

Пусть теперь $l_{ij}^{(m)}$ — минимальный вес любого пути из вершины i в вершину j, содержащий не более m ребер. Если $m = 0$, то кратчайший путь из вершины i в вершину j существует тогда и только тогда, когда $i = j$. Таким образом,

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{если } i = j, \\ \infty & \text{если } i \neq j. \end{cases}$$

Для $m \geqslant 1$ величина $l_{ij}^{(m)}$ вычисляется как минимум двух величин. Первая — $l_{ij}^{(m-1)}$ (вес кратчайшего пути из вершины i в вершину j, состоящего не более чем из $m-1$ ребер), а вторая — минимальный вес произвольного пути из вершины i в вершину j, который состоит не более чем из m ребер. Этот минимальный вес получается в результате рассмотрения всех возможных предшественников k вершины j. Таким образом, мы можем рекурсивно определить

$$l_{ij}^{(m)} = \min\left(l_{ij}^{(m-1)}, \min_{1 \leqslant k \leqslant n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) = \min_{1 \leqslant k \leqslant n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}. \quad (25.2)$$

Последнее равенство следует из того, что $w_{jj} = 0$ для всех j.

Чему равен фактический вес каждого из кратчайших путей $\delta(i, j)$? Если граф не содержит циклов с отрицательным весом, то для каждой пары вершин i и j, для которых справедливо неравенство $\delta(i, j) < \infty$, существует кратчайший путь из вершины i в вершину j, который является простым и, следовательно, содержит не более $n-1$ ребер. Путь из вершины i в вершину j, содержащий более $n-1$ ребер, не может иметь меньший вес, чем кратчайший путь из вершины i в вершину j. Поэтому фактический вес каждого из кратчайших путей определяется равенствами

$$\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots. \quad (25.3)$$

Вычисление весов кратчайших путей в восходящем порядке

Используя в качестве входной матрицу $W = (w_{ij})$, вычислим ряд матриц $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, где для $m = 1, 2, \ldots, n-1$ имеем $L^{(m)} = \left(l_{ij}^{(m)}\right)$. Конечная матрица $L^{(n-1)}$ содержит фактический вес каждого из кратчайших путей. Заметим, что для всех вершин $i, j \in V$ выполняется равенство $l_{ij}^{(1)} = w_{ij}$, так что $L^{(1)} = W$.

Сердцем алгоритма является приведенная ниже процедура, которая на основе заданных матриц $L^{(m-1)}$ и W вычисляет и возвращает матрицу $L^{(m)}$. Другими словами, она расширяет вычисленные на текущий момент кратчайшие пути, добавляя в них еще по одному ребру.

```
EXTEND_SHORTEST_PATHS(L, W)
1  n ← rows[L]
2  Пусть L' = (l'_ij) — матрица размера n × n
3  for i ← 1 to n
4        do for j ← 1 to n
5              do l'_ij ← ∞
6                 for k ← 1 to n
7                       do l'_ij ← min(l'_ij, l_ik + w_kj)
8  return L'
```

В этой процедуре вычисляется матрица $L' = \left(l'_{ij}\right)$, которая и возвращается процедурой по завершении. Вычисления осуществляются на основе уравнения (25.2) для всех пар индексов i и j; при этом в качестве $L^{(m-1)}$ используется матрица L, а в качестве $L^{(m)}$ — матрица L'. (В псевдокоде верхние индексы не используются, чтобы входные и выходные матрицы процедуры не зависели от m.) Из-за наличия трех вложенных циклов **for** время работы алгоритма равно $\Theta\left(n^3\right)$.

Теперь становится понятной связь с умножением матриц. Предположим, требуется вычислить матричное произведение $C = A \cdot B$, где A и B — матрицы размером $n \times n$. Тогда для $i, j = 1, 2, \ldots, n$ мы вычисляем

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}. \tag{25.4}$$

Заметим, что если в уравнении (25.2) выполнить подстановки

$$\begin{aligned} l^{(m-1)} &\rightarrow a, \\ w &\rightarrow b, \\ l^{(m)} &\rightarrow c, \\ \min &\rightarrow +, \\ + &\rightarrow \cdot, \end{aligned}$$

то мы получим уравнение (25.4). Таким образом, если в процедуре EXTEND_SHORTEST_PATHS произвести соответствующие изменения и заменить значение ∞ (исходное значение для операции вычисления минимума) значением 0 (исходное значение для вычисления суммы), получим процедуру для непосредственного перемножения матриц со временем выполнения $\Theta\left(n^3\right)$:

```
MATRIX_MULTIPLY(A, B)
1  n ← rows[A]
2  Пусть C — матрица n × n
3  for i ← 1 to n
4       do for j ← 1 to n
5            do c_ij ← 0
6               for k ← 1 to n
7                    do c_ij ← c_ij + a_ik · b_kj
8  return C
```

Возвращаясь к задаче о кратчайших путях между всеми парами вершин, вычислим веса кратчайших путей путем поэтапного расширения путей ребро за ребром. Обозначая через $A \cdot B$ матричное "произведение", которое возвращается процедурой EXTEND_SHORTEST_PATHS(A, B), вычислим последовательность $n-1$ матриц:

$$\begin{aligned} L^{(1)} &= L^{(0)} \cdot W &&= W, \\ L^{(2)} &= L^{(1)} \cdot W &&= W^2, \\ L^{(3)} &= L^{(2)} \cdot W &&= W^3, \\ &\vdots \\ L^{(n-1)} &= L^{(n-2)} \cdot W &&= W^{n-1}. \end{aligned}$$

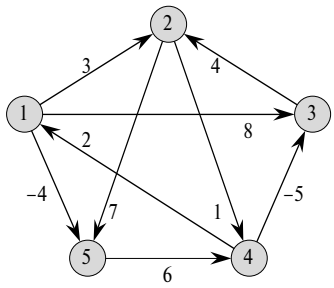
Как было показано ранее, матрица $L^{(n-1)} = W^{n-1}$ содержит веса кратчайших путей. В приведенной ниже процедуре эта последовательность вычисляется в течение времени $\Theta\left(n^4\right)$:

```
SLOW_ALL_PAIRS_SHORTEST_PATHS(W)
1  n ← rows[W]
2  L^(1) ← W
3  for m ← 2 to n − 1
4       do L^(m) ← EXTEND_SHORTEST_PATHS(L^(m−1), W)
5  return L^(n−1)
```

На рис. 25.1 приведен граф и матрицы $L^{(m)}$, вычисленные процедурой SLOW_ALL_PAIRS_SHORTEST_PATHS. Можно легко убедиться в том, что величина $L^{(5)} = L^{(4)} \cdot W$ равна $L^{(4)}$, а следовательно, для всех $m \geqslant 4$ выполняется равенство $L^{(m)} = L^{(4)}$.

Улучшение времени работы

Следует сказать, что наша цель состоит не в том, чтобы вычислить *все* матрицы $L^{(m)}$: нам нужна только матрица $L^{(n-1)}$. Напомним, что если циклы с отрицательным весом отсутствуют, из уравнения (25.3) вытекает равенство $L^{(m)} = L^{(n-1)}$



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Рис. 25.1. Ориентированный граф и последовательность матриц $L^{(m)}$, вычисленных в процедуре SLOW_ALL_PAIRS_SHORTEST_PATHS

для всех целых $m \geqslant n-1$. Матричное умножение, определенное процедурой EXTEND_SHORTEST_PATHS, как и обычное матричное умножение, является ассоциативным (упражнение 25.1-4). Таким образом, матрицу $L^{(n-1)}$ можно получить путем вычисления $\lceil \lg(n-1) \rceil$ матричных умножений в последовательности:

$$\begin{aligned} L^{(1)} &= W\,, \\ L^{(2)} &= W^2 = W \cdot W\,, \\ L^{(4)} &= W^4 = W^2 \cdot W^2 \\ L^{(8)} &= W^8 = W^4 \cdot W^4\,, \\ &\vdots \\ L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}\,. \end{aligned}$$

В силу неравенства $2^{\lceil \lg(n-1) \rceil} \geqslant n-1$ последнее произведение $L^{\left(2^{\lceil \lg(n-1) \rceil}\right)}$ равно $L^{(n-1)}$.

В приведенной ниже процедуре данная последовательность матриц вычисляется методом ***многократного возведения в квадрат*** (repeated squaring):

```
FASTER_ALL_PAIRS_SHORTEST_PATHS(W)
1  n ← rows[W]
2  L^(1) ← W
3  m ← 1
4  while m < n − 1
5        do L^(2m) ← EXTEND_SHORTEST_PATHS(L^(m), L^(m))
6           m ← 2m
7  return L^(m)
```

В каждой итерации цикла **while** в строках 4–6, начиная с $m = 1$, вычисляется матрица $L^{(2m)} = \left(L^{(m)}\right)^2$. В конце каждой итерации значение m удваивается. В последней итерации матрица $L^{(n-1)}$ вычисляется путем фактического вычисления матрицы $L^{(2m)}$ для некоторого значения $n - 1 \leqslant 2m \leqslant 2n - 2$. Согласно уравнению (25.3), $L^{(2m)} = L^{(n-1)}$. Далее выполняется проверка в строке 4, значение m удваивается, после чего выполняется условие $m \geqslant n - 1$, так что условие цикла оказывается невыполненным, и процедура возвращает последнюю вычисленную матрицу.

Время работы процедуры FASTER_ALL_PAIRS_SHORTEST_PATHS равно $\Theta\left(n^3 \lg n\right)$, поскольку вычисление каждого из $\lceil \lg (n-1) \rceil$ матричных произведений требует $\Theta\left(n^3\right)$ времени. Заметим, что этот код довольно компактен. Он не содержит сложных структур данных, поэтому константа, скрытая в Θ-обозначении, невелика.

Упражнения

25.1-1. Выполните процедуру SLOW_ALL_PAIRS_SHORTEST_PATHS над взвешенным ориентированным графом, показанным на рис. 25.2, и запишите промежуточные матрицы, которые получаются в каждой итерации цикла. Затем проделайте то же самое для процедуры FASTER_ALL_PAIRS_SHORTEST_PATHS.

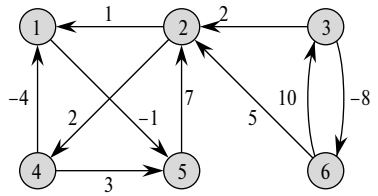


Рис. 25.2. Взвешенный ориентированный граф, который используется в упражнениях 25.1-1, 25.2-1 и 25.3-1

25.1-2. Почему требуется, чтобы при всех $1 \leqslant i \leqslant n$ выполнялось равенство $w_{ii} = 0$?

25.1-3. Чему в операции обычного матричного умножения соответствует матрица

$$L^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$

использующаяся в алгоритмах поиска кратчайших путей?

25.1-4. Покажите, что матричное умножение, определенное в процедуре EXTEND_ SHORTEST_PATHS, обладает свойством ассоциативности.

25.1-5. Покажите, как выразить задачу о кратчайшем пути из единого истока в виде произведения матриц и вектора. Опишите, как вычисление этого произведения соответствует алгоритму, аналогичного алгоритму Беллмана-Форда (см. раздел 24.1).

25.1-6. Предположим, что в алгоритмах, о которых идет речь в этом разделе, также нужно найти вершины, принадлежащие кратчайшим путям. Покажите, как в течение времени $O\left(n^3\right)$ вычислить матрицу предшествования Π на основе известной матрицы L, содержащей веса кратчайших путей.

25.1-7. Вершины, принадлежащие кратчайшим путям, можно вычислить одновременно с весом каждого из кратчайших путей. Пусть $\pi_{ij}^{(m)}$ — предшественник вершины j на произвольном пути с минимальным весом, который соединяет вершины i и j и содержит не более m ребер. Модифицируйте процедуры EXTEND_SHORTEST_PATHS и SLOW_ALL_PAIRS_SHORTEST_PATHS таким образом, чтобы они позволяли вычислять матрицы $\Pi^{(1)}, \Pi^{(2)}, \ldots \Pi^{(n-1)}$ так же, как вычисляются матрицы $L^{(1)}, L^{(2)}, \ldots,$ $L^{(n-1)}$.

25.1-8. В процедуре FASTER_ALL_PAIRS_SHORTEST_PATHS в том виде, в котором она представлена, требуется хранить $\lceil \lg (n-1) \rceil$ матриц, содержащих по n^2 элементов, для чего требуется общий объем памяти, равный $\Theta\left(n^2 \lg n\right)$. Модифицируйте данную процедуру таким образом, чтобы ей требовалось всего $\Theta\left(n^2\right)$ памяти для хранения двух матриц размером $n \times n$.

25.1-9. Модифицируйте процедуру FASTER_ALL_PAIRS_SHORTEST_PATHS таким образом, чтобы она была способна выявлять наличие циклов с отрицательным весом.

25.1-10. Разработайте эффективный алгоритм, позволяющий найти в графе количество ребер цикла с отрицательным весом, имеющего минимальную длину.

25.2 Алгоритм Флойда-Варшалла

В этом разделе задача о поиске кратчайших путей между всеми парами вершин в ориентированном графе $G = (V, E)$ будет решаться с помощью различных модификаций динамического программирования. Время работы полученного в результате алгоритма, известного как ***алгоритм Флойда-Варшалла*** (Floyd-Warshall algorithm), равно $\Theta\left(V^3\right)$. Как и ранее, наличие ребер с отрицательным весом допускается, но предполагается, что циклы с отрицательным весом отсутствуют. Как и в разделе 25.1, будет пройден весь процесс разработки алгоритма в стиле динамического программирования. После исследования полученного в результате алгоритма будет представлен аналогичный метод, позволяющий найти транзитивное замыкание ориентированного графа.

Структура кратчайшего пути

В алгоритме Флойда-Варшалла используется характеристика структуры кратчайшего пути, отличная от той, которая применялась алгоритмах, основанных на перемножении матриц для всех пар вершин. В этом алгоритме рассматриваются "промежуточные" вершины кратчайшего пути. ***Промежуточной*** (intermediate) вершиной простого пути $p = \langle v_1, v_2, \ldots, v_l \rangle$ называется произвольная вершина, отличная от v_1 и v_l, т.е. это любая вершина из множества $\{v_2, v_3, \ldots, v_{l-1}\}$.

Алгоритм Флойда-Варшалла основан на следующем наблюдении. Предположим, что граф G состоит из вершин $V = \{1, 2, \ldots, n\}$. Рассмотрим подмножество вершин $\{1, 2, \ldots, k\}$ для некоторого k. Для произвольной пары вершин $i, j \in V$ рассмотрим все пути из вершины i в вершину j, все промежуточные вершины которых выбраны из множества $\{1, 2, \ldots, k\}$. Пусть среди этих путей p — путь с минимальным весом (этот путь простой). В алгоритме Флойда-Варшалла используется взаимосвязь между путем p и кратчайшими путями из вершины i в вершину j, все промежуточные вершины которых принадлежат множеству $\{1, 2, \ldots, k-1\}$. Эта взаимосвязь зависит от того, является ли вершина k промежуточной на пути p.

- Если k — не промежуточная вершина пути p, то все промежуточные вершины этого пути принадлежат множеству $\{1, 2, \ldots, k-1\}$. Таким образом, кратчайший путь из вершины i в вершину j со всеми промежуточными вершинами из множества $\{1, 2, \ldots, k-1\}$ одновременно является кратчайшим путем из вершины i в вершину j со всеми промежуточными вершинами из множества $\{1, 2, \ldots, k\}$.

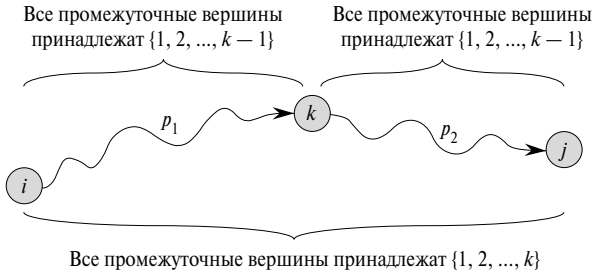


Рис. 25.3. Схема, иллюстрирующая структуру кратчайших путей

- Если k — промежуточная вершина пути p, то этот путь, как видно из рис. 25.3, можно разбить следующим образом: $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$. Согласно лемме 24.1, p_1 — кратчайший путь из вершины i в вершину k, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k\}$. Поскольку k не является промежуточной вершиной пути p_1, понятно, что p_1 — кратчайший путь из вершины i в вершину k, все промежуточные вершины которого принадлежит множеству $\{1, 2, \ldots, k-1\}$. Аналогично, p_2 — кратчайший путь из вершины k в вершину j, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k-1\}$.

Рекурсивное решение задачи о кратчайших путях между всеми парами вершин

Определим на основе сделанных выше наблюдений рекурсивную формулировку оценок кратчайших путей, отличную от той, которая использовалась в разделе 25.1. Пусть $d_{ij}^{(k)}$ — вес кратчайшего пути из вершины i в вершину j, для которого все промежуточные вершины принадлежат множеству $\{1, 2, \ldots, k\}$. Если $k = 0$, то путь из вершины i в вершину j, в котором отсутствуют промежуточные вершины с номером, большим нуля, не содержит промежуточных вершин вообще. Такой путь содержит не более одного ребра, поэтому $d_{ij}^{(0)} = w_{ij}$. Рекурсивное определение, которое соответствует приведенному выше описанию, дается соотношением

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{если } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{если } k \geqslant 1. \end{cases} \tag{25.5}$$

Поскольку все промежуточные вершины произвольного пути принадлежат множеству $\{1, 2, \ldots, n\}$, матрица $D_{ij}^{(n)} = \left(d_{ij}^{(n)}\right)$ дает конечный ответ: $d_{ij}^{(n)} = \delta(i, j)$ для всех пар вершин $i, j \in V$.

Вычисление весов кратчайших путей в восходящем порядке

Исходя из рекуррентного соотношения (25.5), можно составить приведенную ниже процедуру, предназначенную для вычисления величин $d_{ij}^{(k)}$ в порядке возрастания k. В качестве входных данных выступает матрица W размерами $n \times n$, определенная в уравнении (25.1). Процедура возвращает матрицу $D^{(n)}$, содержащую веса кратчайших путей.

```
FLOYD_WARSHALL(W)
1  n ← rows[W]
2  D^(0) ← W
3  for k ← 1 to n
4       do for i ← 1 to n
5            do for j ← 1 to n
6                 do d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7  return D^(n)
```

На рис. 25.4 приведены матрицы $D^{(k)}$, вычисленные алгоритмом Флойда-Варшалла для графа, изображенного на рис. 25.1.

Время работы алгоритма Флойда-Варшалла определяется трижды вложенными друг в друга циклами **for**, определенными в строках 3–6. Поскольку для каждого выполнения строки 6 требуется время $O(1)$, алгоритм завершает работу в течение времени $\Theta(n^3)$. Код этого алгоритма такой же компактный, как и код алгоритма, представленного в разделе 25.1. Он не содержит сложных структур данных, поэтому константа, скрытая в Θ-обозначениях, мала. Таким образом, алгоритм Флойда-Варшалла имеет практическую ценность даже для входных графов среднего размера.

Построение кратчайшего пути

Существует множество различных методов, позволяющих строить кратчайшие пути в алгоритме Флойда-Варшалла. Один из них — вычисление матрицы D, содержащей веса кратчайших путей, с последующим конструированием на ее основе матрицы предшествования Π. Этот метод можно реализовать таким образом, чтобы время его выполнения было равно $O(n^3)$ (упражнение 25.1-6). Если задана матрица предшествования Π, то вывести вершины на указанном кратчайшем пути можно с помощью процедуры PRINT_ALL_PAIRS_SHORTEST_PATH.

Матрицу предшествования Π можно так же вычислить “на лету”, как в алгоритме Флойда-Варшалла вычисляются матрицы $D^{(k)}$. Точнее говоря, вычисляется последовательность матриц $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, где $\Pi = \Pi^{(n)}$, а элемент $\pi_{ij}^{(k)}$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Рис. 25.4. Последовательность матриц $D^{(k)}$ и $\Pi^{(k)}$, вычисленная алгоритмом Флойда-Варшалла для графа, приведенного на рис. 25.1

определяется как предшественник вершины j на кратчайшем пути из вершины i, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k\}$.

Можно дать рекурсивное определение величины $\pi_{ij}^{(k)}$. Если $k = 0$, то кратчайший путь из вершины i в вершину j не содержит промежуточных вершин. Таким образом,

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{если } i = j \text{ или } w_{ij} = \infty, \\ i & \text{если } i \neq j \text{ и } w_{ij} < \infty. \end{cases} \tag{25.6}$$

Если при $k \geqslant 1$ получаем путь $i \rightsquigarrow k \rightsquigarrow j$, где $k \neq j$, то выбранный нами предшественник вершины j совпадает с выбранным предшественником этой же вершины на кратчайшем пути из вершины k, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k-1\}$. В противном случае выбирается тот же предшественник вершины j, который выбран на кратчайшем пути из вершины i, у которого все промежуточные вершины принадлежат множеству $\{1, 2, \ldots, k-1\}$. Выражаясь формально, при $k \geqslant 1$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{если } d_{ij}^{(k-1)} \leqslant d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{если } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases} \tag{25.7}$$

Вопрос о том, как включить вычисление матрицы $\Pi^{(k)}$ в процедуру FLOYD_WARSHALL, предлагается рассмотреть в упражнении 25.2-3 самостоятельно. На рис. 25.4 показана последовательность матриц $\Pi^{(k)}$, полученных в результате обработки алгоритмом графа, изображенного на рис. 25.1. В упомянутом упражнении также предлагается выполнить более сложную задачу, — доказать, что подграф предшествования $G_{\pi,i}$ является деревом кратчайших путей с корнем i. Еще один способ реконструкции кратчайших путей представлен в упражнении 25.2-7.

Транзитивное замыкание ориентированного графа

Может возникнуть необходимость установить, существуют ли в заданном ориентированном графе $G = (V, E)$, множество вершин которого $V = \{1, 2, \ldots, n\}$, пути из вершины i в вершину j для всех возможных пар вершин $i, j \in V$. ***Транзитивное замыкание*** (transitive closure) графа G определяется как граф $G^* = (V, E^*)$, где $E^* = \{(i, j) :$ в графе G имеется путь из вершины i в вершину $j\}$.

Один из способов найти транзитивное замыкание графа в течение времени $\Theta\left(n^3\right)$ — присвоить каждому ребру из множества E вес 1 и выполнить алгоритм Флойда-Варшалла. Если путь из вершины i в вершину j существует, то мы получим $d_{ij} < n$; в противном случае $d_{ij} = \infty$.

Имеется и другой, подобный путь вычисления транзитивного замыкания графа G в течение времени $\Theta\left(n^3\right)$, на практике позволяющий сэкономить время и память. Этот метод включает в себя подстановку логических операций $\vee$ (логическое

ИЛИ) и $\wedge$ (логическое И) вместо использующихся в алгоритме Флойда-Варшалла арифметических операций min и +. Определим значение $t_{ij}^{(k)}$ при $i, j, k = 1, 2, \ldots, n$ равным 1, если в графе G существует путь из вершины i в вершину j, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k\}$; в противном случае эта величина равна 0. Конструируя транзитивное замыкание $G^* = (V, E^*)$, будем помещать ребро (i, j) в множество E^* тогда и только тогда, когда $t_{ij}^{(k)} = 1$. Рекурсивное определение величины $t_{ij}^{(k)}$, построенное по аналогии с рекуррентным соотношением (25.5), имеет вид

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{если } i \neq j \text{ и } (i,j) \notin E, \\ 1 & \text{если } i = j \text{ или } (i,j) \in E, \end{cases}$$

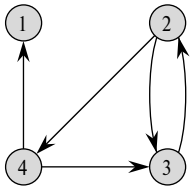
а при $k \geqslant 1$ выполняется соотношение

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right). \tag{25.8}$$

Как и в алгоритме Флойда-Варшалла, матрицы $T^{(k)} = \left(t_{ij}^{(k)} \right)$ вычисляются в порядке возрастания k:

```
TRANSITIVE_CLOSURE(G)
 1  n ← |V[G]|
 2  for i ← 1 to n
 3       do for j ← 1 to n
 4            do if i = j или (i, j) ∈ E[G]
 5                 then t_ij^(0) ← 1
 6                 else t_ij^(0) ← 0
 7  for k ← 1 to n
 8       do for i ← 1 to n
 9            do for j ← 1 to n
10                 do t_ij^(k) ← t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
11  return T^(n)
```

На рис. 25.5 приведены матрицы $T^{(k)}$, вычисленные процедурой TRANSITIVE_CLOSURE для приведенного графа. Время работы процедуры TRANSITIVE_CLOSURE, как и время работы алгоритма Флойда-Варшалла, равно $\Theta\left(n^3\right)$. Однако на некоторых компьютерах логические операции с однобитовыми величинами выполняются быстрее, чем арифметические операции со словами, представляющими целочисленные данные. Кроме того, поскольку в прямом алгоритме транзитивного замыкания используются только булевы, а не целые величины, ему требуется меньший объем памяти, чем алгоритму Флойда-Варшалла. Объем экономящейся памяти зависит от размера слова компьютера.



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Рис. 25.5. Ориентированный граф и матрицы $T^{(k)}$, вычисленные алгоритмом транзитивного замыкания

Упражнения

25.2-1. Примените алгоритм Флойда-Варшалла ко взвешенному ориентированному графу, изображенному на рис. 25.2. Приведите матрицы $D^{(k)}$, полученные на каждой итерации внешнего цикла.

25.2-2. Покажите, как найти транзитивное замыкание с использованием методики из раздела 25.1.

25.2-3. Модифицируйте процедуру FLOYD_WARSHALL таким образом, чтобы в ней вычислялись матрицы $\Pi^{(k)}$ в соответствии с уравнениями (25.6) и (25.7). Дайте строгое доказательство того, что для всех $i \in V$ граф предшествования $G_{\pi,i}$ — дерево кратчайших путей с корнем i. (*Указание*: чтобы показать, что граф $G_{\pi,i}$ — ациклический, сначала покажите, что из равенства $\pi_{ij}^{(k)} = l$ в соответствии с определением $\pi_{ij}^{(k)}$ следует, что $d_{ij}^{(k)} \geqslant d_{il}^{(k)} + w_{lj}$. Затем адаптируйте доказательство леммы 24.16.)

25.2-4. Как было показано, для работы алгоритма Флойда-Варшалла требуется объем памяти, равный $\Theta\left(n^3\right)$, поскольку мы вычисляем величины $d_{uj}^{(k)}$ для $i, j, k = 1, 2, \ldots, n$. Покажите, что приведенная ниже процедура, в которой все верхние индексы просто опускаются, корректна и для ее работы требуется объем памяти $\Theta\left(n^2\right)$:

```
FLOYD_WARSHALL'(W)
1  n ← rows[W]
2  D ← W
3  for k ← 1 to n
4       do for i ← 1 to n
5            do for j ← 1 to n
6                 do d_ij ← min(d_ij, d_ik + d_kj)
7  return D
```

25.2-5. Предположим, что мы модифицируем способ обработки равенства в уравнении (25.7):

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{если } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{если } d_{ij}^{(k-1)} \geqslant d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}.$$

Корректно ли такое альтернативное определение матрицы предшествования П?

25.2-6. Как с помощью выходных данных алгоритма Флойда-Варшалла установить наличие цикла с отрицательным весом?

25.2-7. В одном из способов, позволяющем восстановить в алгоритме Флойда-Варшалла кратчайшие пути, используются величины $\phi_{ij}^{(k)}$ при $i, j, k = 1, 2, \ldots, n$, где $\phi_{ij}^{(k)}$ — промежуточная вершина с наибольшим номером, принадлежащая кратчайшему пути из вершины i в вершину j, все промежуточные вершины которого принадлежат множеству $\{1, 2, \ldots, k\}$. Дайте рекурсивное определение величин $\phi_{ij}^{(k)}$, модифицируйте процедуру FLOYD_WARSHALL таким образом, чтобы в ней вычислялись эти величины, и перепишите процедуру PRINT_ALL_PAIRS_SHORTEST_PATH так, чтобы в качестве ее входных данных выступала матрица $\Phi = \left(\phi_{ij}^{(n)}\right)$. Чем матрица Φ похожа на таблицу s, которая используется в задаче о перемножении цепочки матриц, описанной в разделе 15.2?

25.2-8. Сформулируйте алгоритм, позволяющий вычислить транзитивное замыкание ориентированного графа $G = (V, E)$ в течение времени $O(V E)$.

25.2-9. Предположим, что транзитивное замыкание ориентированного ациклического графа можно вычислить в течение времени $f(|V|, |E|)$, где f — монотонно возрастающая функция величин $|V|$ и $|E|$. Покажите, что время поиска транзитивного замыкания $G^* = (V, E^*)$ ориентированного графа общего вида $G = (V, E)$ равно $f(|V|, |E|) + O(V + E^*)$.

25.3 Алгоритм Джонсона для разреженных графов

Алгоритм Джонсона позволяет найти кратчайшие пути между всеми парами вершин в течение времени $O\left(V^2 \lg V + V\,E\right)$. Для разреженных графов в асимптотическом пределе он ведет себя лучше, чем алгоритм многократного возведения матриц в квадрат и алгоритм Флойда-Варшалла. Этот алгоритм либо возвращает матрицу, содержащую веса кратчайших путей для всех пар вершин, либо выводит сообщение о том, что входной граф содержит цикл с отрицательным весом. В алгоритме Джонсона используются подпрограммы, в которых реализованы алгоритмы Дейкстры и Беллмана-Форда, описанные в главе 24.

В алгоритме Джонсона используется метод ***изменения веса*** (reweighting), который работает следующим образом. Если веса всех ребер w в графе $G = (V, E)$ неотрицательные, можно найти кратчайшие пути между всеми парами вершин, по одному разу запустив алгоритм Дейкстры для каждой вершины. Если неубывающая очередь с приоритетами реализована в виде пирамиды Фибоначчи, то время работы этого алгоритма будет равно $O\left(V^2 \lg V + V\,E\right)$. Если в графе G содержатся ребра с отрицательным весом, но отсутствуют циклы с отрицательным весом, можно просто вычислить новое множество ребер с неотрицательными весами, позволяющее воспользоваться тем же методом. Новое множество, состоящее из весов ребер $\widehat{w}$, должно удовлетворять двум важным свойствам.

1. Для всех пар вершин $u, v \in V$ путь p является кратчайшим путем из вершины u в вершину v с использованием весовой функции w тогда и только тогда, когда p — также кратчайший путь из вершины u в вершину v с весовой функцией $\widehat{w}$.
2. Для всех ребер (u, v) новый вес $\widehat{w}\,(u, v)$ — неотрицательный.

Как вскоре станет понятно, предварительную обработку графа G с целью определить новую весовую функцию $\widehat{w}$ можно выполнить в течение времени $O\,(V\,E)$.

Сохранение кратчайших путей

Как видно из приведенной ниже леммы, легко организовать изменение весов, удовлетворяющее первому из сформулированных выше свойств. Значения весов кратчайших путей, полученные с помощью весовой функции w, обозначены как δ, а веса кратчайших путей, полученных с помощью весовой функции $\widehat{w}$, — как $\widehat{\delta}$.

Лемма 25.1 (Изменение весов сохраняет кратчайшие пути). Пусть дан взвешенный ориентированный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbf{R}$,

и пусть $h : E \to \mathbf{R}$ — произвольная функция, отображающая вершины на действительные числа. Для каждого ребра $(u, v) \in E$ определим

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v). \tag{25.9}$$

Пусть $p = \langle v_0, v_1, \ldots, v_k \rangle$ — произвольный путь из вершины v_0 в вершину v_k. p является кратчайшим путем с весовой функцией w тогда и только тогда, когда он является кратчайшим путем с весовой функцией $\widehat{w}$, т.е. равенство $w(p) = \delta(v_0, v_k)$ равносильно равенству $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$. Кроме того, граф G содержит цикл с отрицательным весом с использованием весовой функции w тогда и только тогда, когда он содержит цикл с отрицательным весом с использованием весовой функции $\widehat{w}$.

Доказательство. Начнем с того, что покажем справедливость равенства

$$\widehat{w}(p) = w(p) + h(v_0) - h(v_k). \tag{25.10}$$

Запишем цепочку соотношений

$$\begin{aligned}
\widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) = \\
&= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) = \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) = \\
&= w(p) + h(v_0) - h(v_k),
\end{aligned}$$

где предпоследнее равенство легко получается из "телескопичности" суммы разностей.

Таким образом, вес любого пути p из вершины v_0 в вершину v_k равен $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$. Если один путь из вершины v_0 в вершину v_k короче другого с использованием весовой функции w, то он будет короче и с использованием весовой функции $\widehat{w}$. Таким образом, равенство $w(p) = \delta(v_0, v_k)$ выполняется тогда и только тогда, когда $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$.

Наконец, покажем, что граф G содержит цикл с отрицательным весом с использованием весовой функции w тогда и только тогда, когда он содержит такой цикл с использованием весовой функции $\widehat{w}$. Рассмотрим произвольный цикл $c = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_0 = v_k$. В соответствии с уравнением (25.10), выполняется соотношение

$$\widehat{w}(c) = w(c) + h(v_0) - h(v_k) = w(c),$$

а следовательно, вес цикла c будет отрицательным с использованием весовой функции w тогда и только тогда, когда он отрицательный с использованием весовой функции $\widehat{w}$. ■

Генерация неотрицательных весов путем их изменения

Следующая наша цель заключается в том, чтобы обеспечить выполнение второго свойства: нужно, чтобы величина $\widehat{w}(u,v)$ была неотрицательной для всех ребер $(u,v) \in E$. Для данного взвешенного ориентированного графа $G = (V,E)$ с весовой функцией $w : E \to \mathbf{R}$ мы создадим новый граф $G' = (V',E')$, где $V' = V \cup \{s\}$ для некоторой новой вершины $s \notin V$ и $E' = E \cup \{(s,v) : v \in V\}$. Расширим весовую функцию w таким образом, чтобы для всех вершин $v \in V$ выполнялось равенство $w(s,v) = 0$. Заметим, что поскольку в вершину s не входит ни одно ребро, эту вершину не содержит ни один кратчайший путь графа G' отличный от того, который исходит из s. Кроме того, граф G' не содержит циклов с отрицательным весом тогда и только тогда, когда таких циклов не содержит граф G. На рис. 25.6*а* показан граф G', соответствующий графу G на рис. 25.1.

Теперь предположим, что графы G и G' не содержат циклов с отрицательным весом. Определим для всех вершин $v \in V'$ величину $h(v) = \delta(s,v)$. Согласно неравенству треугольника (лемма 24.10), для всех ребер $(u,v) \in E'$ выполняется соотношение $h(v) \leqslant h(u) + w(u,v)$. Таким образом, если мы определим новые веса $\widehat{w}$ в соответствии с уравнением (25.9), то получим $\widehat{w}(u,v) = w(u,v) + h(u) - h(v) \geqslant 0$, так что второе свойство удовлетворяется. На рис. 25.6*б* показан граф G', полученный в результате переопределения весов графа, изображенного на рис. 25.6*а*.

Вычисление кратчайших путей между всеми парами вершин

В алгоритме Джонсона, предназначенном для вычисления кратчайших путей между всеми парами вершин, используется алгоритм Беллмана-Форда (раздел 24.1) и алгоритм Дейкстры (раздел 24.3), реализованные в виде подпрограмм. Предполагается, что ребра хранятся в виде списков смежных вершин. Этот алгоритм возвращает обычную матрицу $D = d_{ij}$ размером $|V| \times |V|$, где $d_{ij} = \delta(i,j)$, или выдает сообщение о том, что входной граф содержит цикл с отрицательным весом. Предполагается, что вершины пронумерованы от 1 до $|V|$, что типично для алгоритмов, предназначенных для поиска кратчайших путей между всеми парами вершин.

```
JOHNSON(G)
 1  Строится граф G', где V[G'] = V[G] ∪ {s},
                E[G'] = E[G] ∪ {(s, v) : v ∈ V[G]}
                и w(s, v) = 0 для всех v ∈ V[G]
 2  if BELLMAN_FORD(G', w, s) = FALSE
 3     then print "Входной граф содержит цикл с отрицательным весом"
 4     else for (для) каждой вершины v ∈ V[G']
 5             do присвоить величине h(v) значение δ(s, v),
                          вычисленное алгоритмом Беллмана-Форда
 6          for (для) каждого ребра (u, v) ∈ E[G']
 7             do ŵ(u, v) ← w(u, v) + h(u) − h(v)
 8          for (для) каждой вершины u ∈ V[G]
 9             do вычисление с помощью алгоритма
                   DIJKSTRA(G, ŵ, u) величин δ̂(u, v) для
                   всех вершин v ∈ V[G]
10                 for (для) каждой вершины v ∈ V[G]
11                    do d_uv ← δ̂(u, v) + h(v) − h(u)
12          return D
```

В этом коде просто выполняются описанные ранее действия. В строке 1 строится граф G'. В строке 2 выполняется алгоритм Беллмана-Форда с входным графом G', весовой функцией w и истоком s. Если граф G', а следовательно, и граф G содержит цикл с отрицательным весом, об этом выводится сообщение в строке 3. В строках 4–11 предполагается, что граф G' не содержит циклов с отрицательным весом. В строках 4–5 для всех вершин $v \in V'$ величине $h(v)$ присваивается вес кратчайшего пути $\delta(s, v)$, вычисленный алгоритмом Беллмана-Форда. В строках 6–7 для каждого ребра вычисляется новый вес $\widehat{w}$. Для каждой пары вершин $u, v \in V$ цикл **for** в строках 8–11 вычисляет вес кратчайшего пути $\widehat{w}(u, v)$ путем однократного вызова алгоритма Дейкстры для каждой вершины из множества V. В строке 11 в элемент матрицы d_{uv} заносится корректный вес кратчайшего пути $\delta(u, v)$, вычисленный с помощью уравнения (25.10). Наконец, в строке 12 возвращается вычисленная матрица D. Работа алгоритма Джонсона проиллюстрирована на рис. 25.6.

Если неубывающая очередь с приоритетами реализована в алгоритме Дейкстры в виде пирамиды Фибоначчи, то время работы алгоритма Джонсона равно $O\left(V^2 \lg V + V E\right)$. Более простая реализация неубывающей очереди с приоритетами приводит к тому, что время работы становится равным $O(V E \lg V)$, но для разреженных графов эта величина в асимптотическом пределе ведет себя лучше, чем время работы алгоритма Флойда-Варшалла.

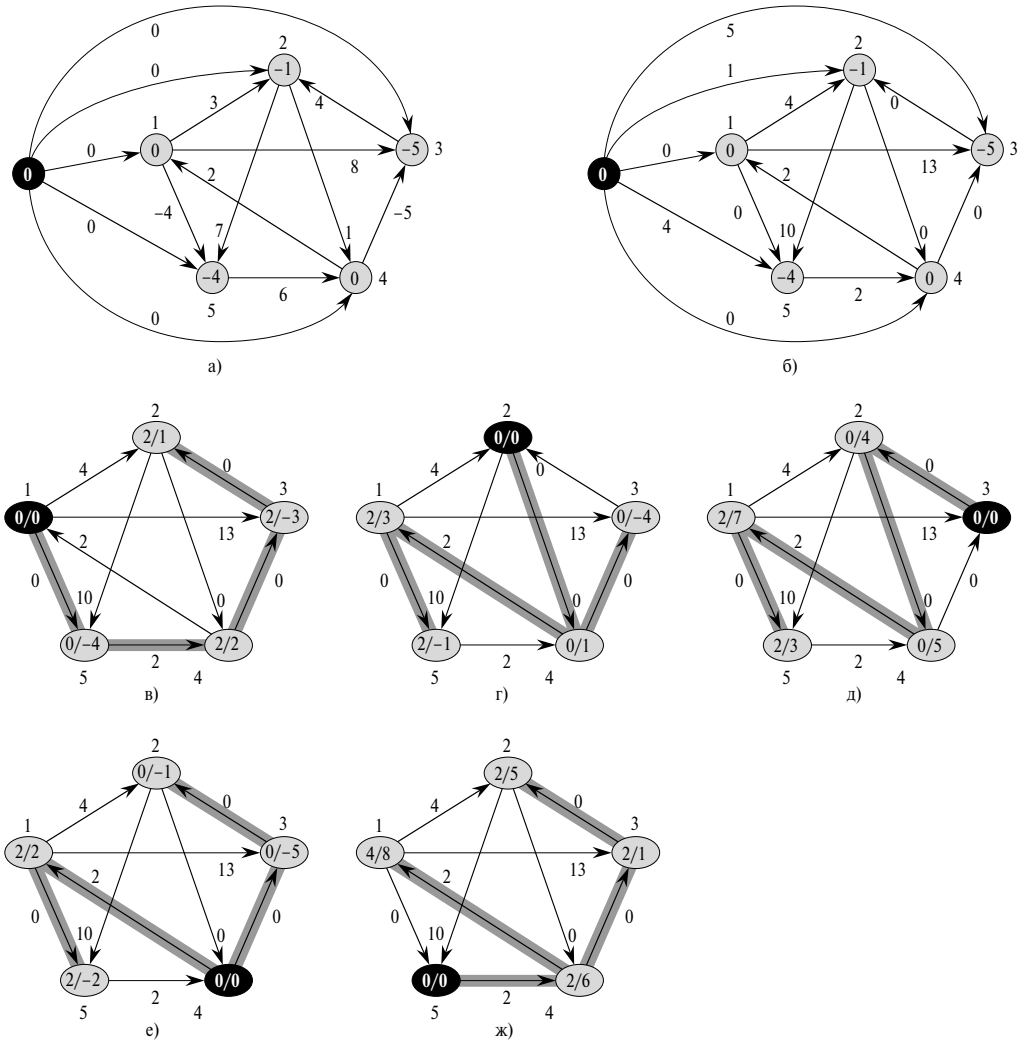


Рис. 25.6. Работа алгоритма Джонсона, предназначенного для поиска кратчайших путей между всеми парами вершин, с графом, изображенным на рис. 25.1

Упражнения

25.3-1. Вычислите с помощью алгоритма Джонсона кратчайшие пути между всеми парами вершин в графе, изображенном на рис. 25.2. Приведите значения h и $\widehat{w}$, вычисленные этим алгоритмом.

25.3-2. Зачем в множество V добавляется новая вершина s, в результате чего создается множество V'?

25.3-3. Предположим, что для всех ребер $(u, v) \in E$ выполняется неравенство $w(u, v) \geqslant 0$. Как между собой взаимосвязаны весовые функции w и $\widehat{w}$?

25.3-4. Профессор утверждает, что изменить веса ребер можно проще, чем это делается в алгоритме Джонсона. Для этого надо найти $w^* = \min_{(u,v)\in E}\{w(u, v)\}$ и для всех ребер $(u, v) \in E$ принять $\widehat{w}(u, v) = w(u, v) - w^*$. Где кроется ошибка в методе, предложенном профессором?

25.3-5. Предположим, что алгоритм Джонсона выполняется для ориентированного графа G с весовой функцией w. Покажите, что если граф G содержит цикл c с нулевым весом, то для всех ребер (u, v) этого цикла $\widehat{w}(u, v) = 0$.

25.3-6. Профессор утверждает, что нет необходимости создавать новую вершину-исток в строке 1 алгоритма JOHNSON. Профессор считает, что вместо этого можно использовать $G' = G$, а в роли вершины s использовать любую вершину из множества $V[G]$. Приведите пример взвешенного ориентированного графа G, для которого воплощение идеи профессора в алгоритме JOHNSON даст неправильный ответ. Затем покажите, что если граф G строго связан (каждая его вершина достижима из любой другой вершины), то результаты работы алгоритма JOHNSON с учетом модификации профессора будут верны.

Задачи

25-1. **Транзитивное замыкание динамического графа**

Предположим, что нужно поддерживать транзитивное замыкание ориентированного графа $G = (V, E)$ по мере добавления ребер в множество E. Другими словами, после добавления каждого ребра нужно обновить транзитивное замыкание добавленных до этого времени ребер. Предположим, что граф G изначально не содержит ребер, и что транзитивное замыкание должно быть представлено в виде булевой матрицы.

а) Покажите, каким образом после добавления нового ребра в граф G транзитивное замыкание $G^* = (V, E^*)$ графа $G = (V, E)$ можно обновить в течение времени $O\left(V^2\right)$.

б) Приведите пример графа G и ребра e такой, что обновление транзитивного замыкания после добавления ребра e в граф G будет выполняться в течение времени $\Omega\left(V^2\right)$ независимо от используемого алгоритма.

в) Разработайте эффективный алгоритм обновления транзитивного замыкания по мере добавления ребер в граф. Для любой последова-

тельности n добавлений общее время работы этого алгоритма должно быть равно $\sum_{i=1}^{n} t_i = O\left(V^3\right)$, где t_i — время, необходимое для обновления транзитивного замыкания при добавлении i-го ребра. Докажите, что в вашем алгоритме достигается указанная граница времени работы.

25-2. **Кратчайшие пути в ε-плотном графе**

Граф $G = (V, E)$ называется ***ε-плотным*** (ε-dense), если $|E| = \Theta\left(V^{1+\varepsilon}\right)$ для некоторой константы $0 < \varepsilon \leqslant 1$. Если в алгоритмах, предназначенных для поиска кратчайших путей в ε-плотных графах, воспользоваться d-арными неубывающими пирамидами (см. задачу 6-2), то время их выполнения может быть сопоставимо времени выполнения алгоритмов, основанных на применении пирамиды Фибоначчи. При этом удается обойтись без сложных структур данных.

а) Чему равно асимптотическое время работы алгоритмов INSERT, EXTRACT_MIN и DECREASE_KEY в зависимости от кратности d и количества элементов d-арной неубывающей пирамиды n? Чему равно время работы этих алгоритмов, если выбрать $d = \Theta\left(n^{\alpha}\right)$, где $0 < \alpha \leqslant 1$ — некоторая константа? Сравните время работы этих алгоритмов с амортизированными стоимостями этих операций для пирамиды Фибоначчи.

б) Покажите, как в течение времени $O(E)$ вычислить кратчайшие пути из единого истока в ε-плотном ориентированном графе $G = (V, E)$, в котором отсутствуют ребра с отрицательным весом. (*Указание*: выберите величину d как функцию ε.)

в) Покажите, как в течение времени $O(VE)$ решить задачу поиска кратчайших путей между всеми парами вершин в ε-плотном ориентированном графе $G = (V, E)$, в котором отсутствуют ребра с отрицательным весом.

г) Покажите, как в течение времени $O(VE)$ решить задачу поиска кратчайших путей между всеми парами вершин в ε-плотном ориентированном графе $G = (V, E)$, в котором допускается наличие ребер с отрицательным весом, но отсутствуют циклы с отрицательным весом.

Заключительные замечания

Неплохое обсуждение задачи о поиске кратчайших путей между всеми парами вершин содержится в книге Лоулера (Lawler) [196], хотя в ней и не анализируются решения для разреженных графов. Алгоритм перемножения матриц он считает

результатом народного творчества. Алгоритм Флойда-Варшалла был предложен Флойдом (Floyd) [89] и основывался на теореме Варшалла (Warshall) [308], в которой описывается, как найти транзитивное замыкание булевых матриц. Алгоритм Джонсона (Johnson) взят из статьи [168].

Несколько исследователей предложили улучшенные алгоритмы, предназначенные для поиска кратчайших путей с помощью умножения матриц. Фридман (Fredman) [95] показал, что задачу о поиске кратчайшего пути между всеми парами вершин можно решить, выполнив $O\left(V^{5/2}\right)$ операций по сравнению суммарных весов ребер; в результате получится алгоритм, время работы которого равно $O\left(V^3 \left(\lg\lg V / \lg V\right)^{1/3}\right)$, что несколько лучше соответствующей величины для алгоритма Флойда-Варшалла. Еще одно направление исследований показывает, что к задаче о поиске кратчайших путей между всеми парами вершин можно применить алгоритмы быстрого умножения матриц (см. заключительные замечания к главе 28). Пусть $O\left(n^{\omega}\right)$ — время работы самого производительного алгоритма, предназначенного для перемножения матриц размером $n \times n$; в настоящее время $\omega < 2.376$ [70]. Галил (Galil) и Маргалит (Margalit) [105, 106], а также Зайдель (Seidel) [270] разработали алгоритмы, решающие задачу о поиске кратчайших путей между всеми парами вершин в неориентированных невзвешенных графах в течение времени $O\left(V^{\omega} p\left(V\right)\right)$, где $p\left(n\right)$ — функция, полилогарифмически ограниченная по n. В плотных графах время работы этих алгоритмов меньше величины $O\left(V\,E\right)$, необходимой для $|V|$ поисков в ширину. Некоторые исследователи расширили эти результаты и разработали алгоритмы, предназначенные для поиска кратчайших путей между всеми парами вершин в неориентированных графах с целочисленными весами ребер в диапазоне $\{1, 2, \ldots, W\}$. Среди таких алгоритмов быстрее всего в асимптотическом пределе ведет себя алгоритм Шошана (Shoshan) и Цвика (Zwick) [278], время работы которого равно $O\left(W\,V^{\omega} p\left(V\,W\right)\right)$.

Каргер (Karger), Коллер (Koller) и Филлипс (Phillips) [170], а также независимо Мак-Геч (McGeoch) [215] дали временну́ю границу, зависящую от E^*, — подмножества ребер из множества E, входящих в некоторый кратчайший путь. Для заданного графа с неотрицательными весами ребер эти алгоритмы завершают свою работу за время $O\left(V\,E^* + V^2 \lg V\right)$. Это лучший показатель, чем $|V|$-кратное выполнение алгоритма Дейкстры, если $|E^*| = o\left(E\right)$.

Ахо (Aho), Хопкрофт (Hopcroft) и Ульман (Ullman) [5] дали определение алгебраической структуры, известной как “замкнутое полукольцо”. Эта структура служит общим каркасом для решения задач о поиске путей в ориентированных графах. Алгоритм Флойда-Варшалла и описанный в разделе 25.2 алгоритм транзитивного замыкания — примеры алгоритмов поиска путей между всеми парами вершин, основанных на замкнутых полукольцах. Маггс (Maggs) и Плоткин (Plotkin) [208] показали, как искать минимальные остовные деревья с помощью замкнутых полуколец.

ГЛАВА 26

Задача о максимальном потоке

Так же, как дорожную карту можно смоделировать ориентированным графом, чтобы найти кратчайший путь из одной точки в другую, ориентированный граф можно интерпретировать как некоторую транспортную сеть и использовать его для решения задач о потоках вещества в системе трубопроводов. Представим, что некоторый продукт передается по системе от источника, где данный продукт производится, к стоку, где он потребляется. Источник производит продукт с некоторой постоянной скоростью, а сток с той же скоростью потребляет продукт. Интуитивно потоком продукта в любой точке системы является скорость движения продукта. С помощью транспортных сетей можно моделировать течение жидкостей по трубопроводам, движение деталей на сборочных линиях, передачу тока по электрическим сетям, информации — по информационным сетям и т.д.

Каждое ориентированное ребро сети можно рассматривать как канал, по которому движется продукт. Каждый канал имеет заданную пропускную способность, которая характеризует максимальную скорость перемещения продукта по каналу, например, 200 литров жидкости в минуту для трубопровода или 20 ампер для провода электрической цепи. Вершины являются точками пересечения каналов; через вершины, отличные от источника и стока, продукт проходит, не накапливаясь. Иными словами, скорость поступления продукта в вершину должна быть равна скорости его удаления из вершины. Это свойство называется свойством сохранения потока; в случае передачи тока по электрическим цепям ему соответствует закон Кирхгофа.

В задаче о максимальном потоке мы хотим найти максимальную скорость пересылки продукта от источника к стоку, при которой не будут нарушаться ограничения пропускной способности. Это одна из простейших задач, возникающих

в транспортных сетях, и, как будет показано в данной главе, существуют эффективные алгоритмы ее решения. Более того, основные методы, используемые в алгоритмах решения задач о максимальном потоке, можно применять для решения других задач, связанных с транспортными сетями.

В данной главе предлагается два общих метода решения задачи о максимальном потоке. В разделе 26.1 формализуются понятия транспортных сетей и потоков, а также дается формальное определение задачи о максимальном потоке. В разделе 26.2 описывается классический метод Форда-Фалкерсона (Ford and Fulkerson) для поиска максимального потока. В качестве приложения данного метода в разделе 26.3 осуществляется поиск максимального паросочетания в неориентированном двудольном графе. В разделе 26.4 описывается метод "проталкивания предпотока" ("push-relabel"), который лежит в основе многих наиболее быстрых алгоритмов для задач в транспортных сетях. В разделе 26.5 рассматривается еще один алгоритм, время работы которого составляет $O\left(V^3\right)$. Хотя он и не является самым быстрым известным алгоритмом, зато позволяет проиллюстрировать некоторые методы, используемые в асимптотически более быстрых алгоритмах, и на практике является достаточно эффективным.

26.1 Транспортные сети

В данном разделе мы дадим определение транспортных сетей в терминах теории графов, обсудим их свойства, дадим точное определение задачи о максимальном потоке, а также введем некие полезные обозначения.

Транспортные сети и потоки

Транспортная сеть (flow network) $G = (V, E)$ представляет собой ориентированный граф, в котором каждое ребро $(u, v) \in E$ имеет неотрицательную ***пропускную способность*** (capacity) $c(u, v) > 0$. Если $(u, v) \notin E$, предполагается, что $c(u, v) = 0$. В транспортной сети выделяются две вершины: ***источник*** (source) s и ***сток*** (sink) t. Для удобства предполагается, что каждая вершина лежит на неком пути из источника к стоку, т.е. для любой вершины $v \in V$ существует путь $s \rightsquigarrow v \rightsquigarrow t$. Таким образом, граф является связным и $|E| > |V| - 1$. На рис. 26.1*а* показан пример транспортной сети $G = (V, E)$ для задачи грузовых перевозок компании Lucky Puck. Источник s — это фабрика в Ванкувере, а сток t — склад в Виннипеге. Шайбы доставляются через промежуточные города, но за день из города u в город v можно отправить только $c(u, v)$ ящиков. На рисунке приведена пропускная способность каждого ребра сети. На рис. 26.1*б* показаны потоки в транспортной сети. Поток f в сети G имеет значение $|f| = 19$. Показаны только положительные потоки. Если $f(u, v) > 0$, то ребро (u, v) снабжено меткой $f(u, v)/c(u, v)$ (косая черта используется только для того, чтобы отделить

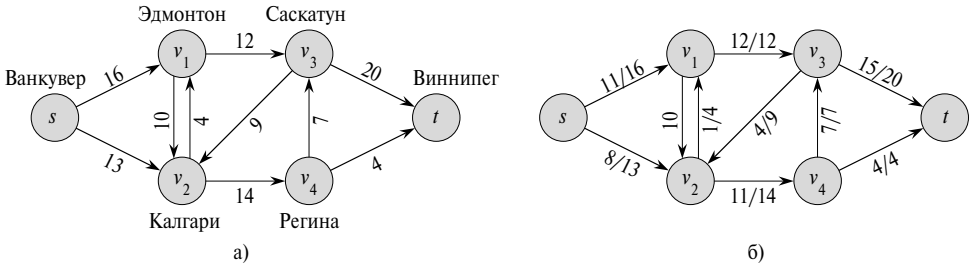


Рис. 26.1. Пример транспортной сети и ее потоки

поток от пропускной способности и не обозначает деление). Если $f(u,v) \leqslant 0$, то у ребра (u,v) показана только его пропускная способность.

Теперь дадим формальное определение потоков. Пусть $G = (V,E)$ — транспортная сеть с функцией пропускной способности c. Пусть s — источник, а t — сток. ***Потоком*** (flow) в G является действительная функция $f : V \times V \to \mathbf{R}$, удовлетворяющая следующим трем условиям.

Ограничение пропускной способности (capacity constraint): $f(u,v) \leqslant c(u,v)$ для всех $u, v \in V$.

Антисимметричноть (skew symmetry): $f(u,v) = -f(v,u)$ для всех $u, v \in V$.

Сохранение потока (flow conservation): для всех $u \in V - \{s,t\}$

$$\sum_{v \in V} f(u,v) = 0.$$

Количество $f(u,v)$, которое может быть положительным, нулевым или отрицательным, называется ***потоком*** (flow) из вершины u в вершину v. ***Величина*** (value) потока f определяется как

$$|f| = \sum_{v \in V} f(s,v), \tag{26.1}$$

т.е. как суммарный поток, выходящий из источника (в данном случае $|\cdot|$ обозначает величину потока, а не абсолютную величину или мощность). В ***задаче о максимальном потоке*** (maximum flow problem) дана некоторая транспортная сеть G с источником s и стоком t, и необходимо найти поток максимальной величины.

Перед тем как рассматривать пример задачи о потоке в сети, кратко проанализируем три свойства потока. Ограничение пропускной способности предполагает, чтобы поток из одной вершины в другую не превышал заданную пропускную способность ребра. Антисимметричность введена для удобства обозначений и заключается в том, что поток из вершины u в вершину v противоположен потоку в обратном направлении. Свойство сохранения потока утверждает, что суммарный поток, выходящий из вершины, не являющейся источником или стоком, равен

нулю. Используя антисимметричность, можно записать свойство сохранения потока как

$$\sum_{u \in V} f(u, v) = 0$$

для всех $v \in V - \{s, t\}$, т.е. суммарный поток, входящий в вершину, отличную от источника и стока, равен 0.

Если в E не присутствуют ни (u, v), ни (v, u), между вершинами u и v нет потока, и $f(u, v) = f(v, u) = 0$. (В упражнении 26.1-1 предлагается формально доказать это свойство.)

Последнее наблюдение касается свойств, связанных с положительными потоками. ***Суммарный положительный поток*** (total positive flow), входящий в вершину v, задается выражением

$$\sum_{\substack{u \in V \\ f(u,v) > 0}} f(u, v). \tag{26.2}$$

Суммарный положительный поток, выходящий из некоторой вершины, определяется симметрично. ***Суммарный чистый поток*** (total net flow) в некоторой вершине равен разности суммарного положительного потока, выходящего из данной вершины, и суммарного положительного потока, входящего в нее. Одна из интерпретаций свойства сохранения потока состоит в том, что для отличной от источника и стока вершины входящий в нее суммарный положительный поток должен быть равен выходящему суммарному положительному потоку. Свойство, что суммарный чистый поток в транзитной вершине должен быть равен 0, часто нестрого формулируют как “входящий поток равен выходящему потоку”.

Пример потока

С помощью транспортной сети можно моделировать задачу о грузовых перевозках, представленную на рис. 26.1*a*. У компании Lucky Puck в Ванкувере есть фабрика (источник s), производящая хоккейные шайбы, а в Виннипеге — склад (сток t), где эти шайбы хранятся. Компания арендует места на грузовиках других фирм для доставки шайб с фабрики на склад. Поскольку грузовики ездят по определенным маршрутам (ребрам) между городами (вершинами) и имеют ограниченную грузоподъемность, компания Lucky Puck может перевозить не более $c(u, v)$ ящиков в день между каждой парой городов u и v, как показано на рис. 26.1*a*. Компания Lucky Puck не может повлиять на маршруты и пропускную способность, т.е. она не может менять транспортную сеть, представленную на рис. 26.1*a*. Ее задача — определить, какое наибольшее количество p ящиков в день можно отгружать, и затем производить именно такое количество, поскольку не имеет смысла производить шайб больше, чем можно отправить на склад. Для

компании не важно, сколько времени займет доставка конкретной шайбы с фабрики на склад, она заботится только о том, чтобы p ящиков в день отправлялось с фабрики и p ящиков в день прибывало на склад.

На первый взгляд, вполне логично моделировать потоком в данной сети "поток" отгрузок, поскольку число ящиков, отгружаемых ежедневно из одного города в другой, подчиняется ограничению пропускной способности. Кроме того, должно соблюдаться условие сохранения потока, поскольку в стационарном состоянии скорость ввоза шайб в некоторый промежуточный город должна быть равна скорости их вывоза. В противном случае ящики станут накапливаться в промежуточных городах.

Однако между отгрузками и потоками существует одно отличие. Компания Lucky Puck может отправлять шайбы из Эдмонтона в Калгари и одновременно из Калгари в Эдмонтон. Предположим, что 8 ящиков в день отгружается из Эдмонтона (v_1 на рис. 26.1) в Калгари (v_2) и 3 ящика в день из Калгари в Эдмонтон. Как бы ни хотелось непосредственно представлять отгрузки потоками, мы не можем этого сделать. Согласно свойству антисимметричности, должно выполняться условие $f(v_1, v_2) = -f(v_2, v_1)$, но это, очевидно, не так, если считать, что $f(v_1, v_2) = 8$, а $f(v_2, v_1) = 3$.

Компания Lucky Puck понимает, что бессмысленно отправлять 8 ящиков в день из Эдмонтона в Калгари и 3 ящика из Калгари в Эдмонтон, если того же результата можно добиться, отгружая 5 ящиков из Эдмонтона в Калгари и 0 ящиков из Калгари в Эдмонтон (что потребует к тому же меньших затрат). Этот последний сценарий мы и представим потоком: $f(v_1, v_2) = 5$, а $f(v_2, v_1) = -5$. По сути, 3 из 8 ящиков в день, отправляемые из v_1 в v_2, ***взаимно уничтожаются*** (canceled) с 3 ящиками в день, отправляемыми из v_2 в v_1.

В общем случае взаимное уничтожение позволяет нам представить отгрузки между двумя городами потоком, который положителен не более чем вдоль одного из двух ребер, соединяющих соответствующие вершины. Таким образом, любую ситуацию, когда ящики перевозятся между двумя городами в обоих направлениях, можно привести с помощью взаимного уничтожения к эквивалентной ситуации, в которой ящики перевозятся только в одном направлении — направлении положительного потока.

По определенному таким образом потоку f нельзя восстановить, какими в действительности являются поставки. Если известно, что $f(u, v) = 5$, то это может быть как в случае, когда 5 единиц отправляется из u в v, так и в случае, когда 8 единиц отправляется из u в v, а 3 единицы из v в u. Обычно нас не будет интересовать, как в действительности организованы физические поставки, для любой пары вершин учитывается только чистое пересылаемое количество. Если важно знать объемы действительных поставок, следует использовать другую модель, в которой сохраняется информация о поставках в обоих направлениях.

Взаимное уничтожение неявно будет присутствовать в рассматриваемых в данной главе алгоритмах. Предположим, что ребро (u, v) имеет величину потока $f(u, v)$. В процессе выполнения алгоритма мы можем увеличить поток вдоль ребра (v, u) на некоторую величину d. С математической точки зрения, эта операция должна уменьшить $f(u, v)$ на d, следовательно, можно считать, что эти d единиц взаимно уничтожаются с d единицами потока, который уже существует вдоль ребра (u, v).

Сети с несколькими источниками и стоками

В задаче о максимальном потоке может быть несколько источников и стоков. Например, у компании Lucky Puck может быть m фабрик $\{s_1, s_2, \ldots, s_m\}$ и n складов $\{t_1, t_2, \ldots, t_n\}$, как показано на рис. 26.2*а*, где приведен пример транспортной сети с пятью источниками и тремя стоками. К счастью, эта задача не сложнее, чем обычная задача о максимальном потоке.

Задача определения максимального потока в сети с несколькими источниками и несколькими стоками сводится к обычной задаче о максимальном потоке. На рис. 26.2*б* показано, как сеть, представленную на рис. 26.2*а*, можно превратить в обычную транспортную сеть с одним источником и одним стоком. Для

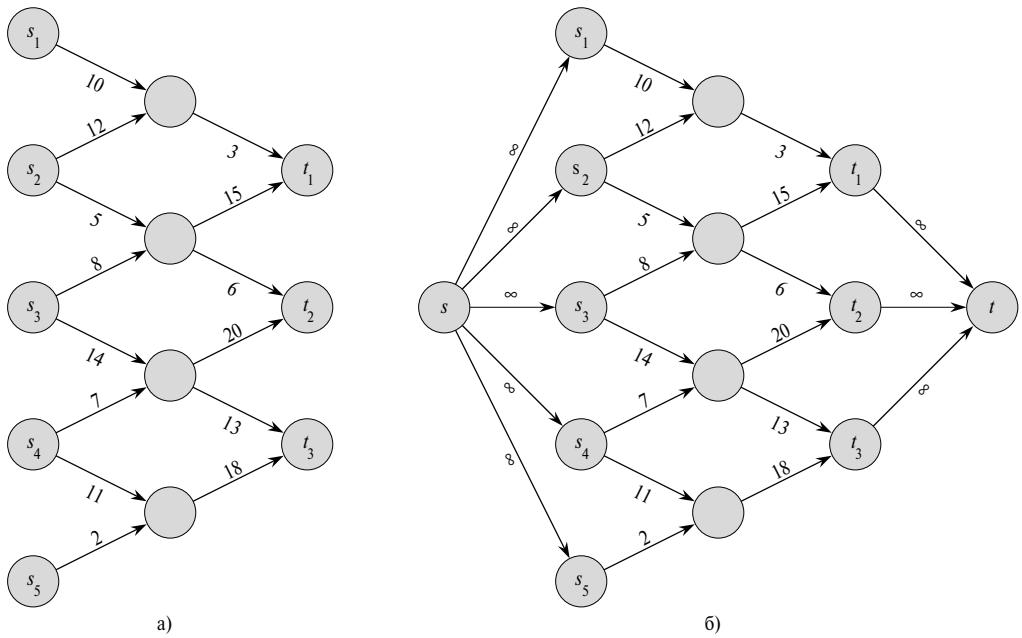


Рис. 26.2. Преобразование задачи о максимальном потоке с несколькими источниками и несколькими стоками к задаче с одним источником и одним стоком

этого добавляется ***фиктивный источник*** (supersource) s и ориентированные ребра (s, s_i) с пропускной способностью $c(s, s_i) = \infty$ для каждого $i = 1, 2, \ldots, m$. Точно так же создается новый ***фиктивный сток*** (supresink) t и добавляются ориентированные ребра (t_i, t) с $c(t_i, t) = \infty$ для каждого $i = 1, 2, \ldots, n$. Интуитивно понятно, что любой поток в сети на рис. 26.2*а* соответствует потоку в сети на рис. 26.2*б* и обратно. Единственный источник s просто обеспечивает поток любого требуемого объема к источникам s_i, а единственный сток t аналогичным образом потребляет поток любого желаемого объема от множественных стоков t_i. В упражнении 26.1-3 предлагается формально доказать эквивалентность этих двух задач.

Как работать с потоками

Далее нам придется работать с функциями, подобными f, аргументами которых являются две вершины транспортной сети. В данной главе мы будем использовать ***неявное обозначение суммирования*** (implicit summation notation), в котором один аргумент или оба могут представлять собой множество вершин, интерпретируя эту запись так, что указанное значение является суммой всех возможных способов замены аргументов их членами. Например, если X и Y — множества вершин, то

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y).$$

Тогда условие сохранения потока можно записать как $f(u, V) = 0$ для всех $u \in V - \{s, t\}$. Для удобства мы также обычно будем опускать фигурные скобки при использовании неявного обозначения суммирования: например, в уравнении $f(s, V - s) = f(s, V)$ запись $V - s$ обозначает множество $V - \{s\}$.

Использование неявного обозначения множеств обычно упрощает уравнения, описывающие потоки. В следующей лемме, доказать которую читателю предлагается в качестве упражнения 26.1-4, формулируются основные тождества, связывающие потоки и множества.

Лемма 26.1. Пусть $G = (V, E)$ — транспортная сеть, и f — некоторый поток в сети G. Тогда справедливы следующие равенства.

1. Для всех $X \subseteq V$ $f(X, X) = 0$.
2. Для всех $X, Y \subseteq V$ $f(X, Y) = -f(Y, X)$.
3. Для всех $X, Y, Z \subseteq V$, таких что $X \cap Y = \emptyset$, $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ и $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$. ■

В качестве примера работы с неявным обозначением суммирования, докажем, что значение потока равно суммарному потоку, входящему в сток; т.е.

$$|f| = f(V, t) \tag{26.3}$$

Интуитивно мы ожидаем, что это свойство справедливо. Согласно условию сохранения потока, все вершины, отличные от источника и стока, имеют одинаковые величины входящего и выходящего положительного потока. Источник по определению имеет суммарный чистый поток, больший 0; т.е. из источника выходит больший положительный поток, чем входит в него. Симметрично, сток является единственной вершиной, которая может иметь суммарный чистый поток, меньший 0; т.е. в сток входит больший положительный поток, чем выходит из него. Формальное доказательство выглядит следующим образом:

$$
\begin{aligned}
|f| &= f(s,V) = && \text{(по определению)}\\
&= f(V,V) - f(V-s,V) = && \text{(согласно лемме 26.1, часть 3)}\\
&= -f(V-s,V) = && \text{(согласно лемме 26.1, часть 1)}\\
&= f(V,V-s) = && \text{(согласно лемме 26.1, часть 2)}\\
&= f(V,t) + f(V,V-s-t) = && \text{(согласно лемме 26.1, часть 3)}\\
&= f(V,t) && \text{(согласно свойству сохранения потока).}
\end{aligned}
$$

Далее в данной главе мы обобщим данный результат (лемма 26.5).

Упражнения

26.1-1. Используя определение потока, докажите, что если $(u,v) \notin E$ и $(v,u) \notin$ $\notin E$, то $f(u,v) = f(v,u) = 0$.

26.1-2. Докажите, что для любой вершины v, отличной от источника и стока, суммарный положительный поток, входящий в v, должен быть равен суммарному положительному потоку, выходящему из v.

26.1-3. Распространите определение и свойства потока на случай задачи с несколькими источниками и несколькими стоками. Покажите, что любой поток в транспортной сети с несколькими источниками и несколькими стоками соответствует некоторому потоку идентичной величины в сети с единственным источником и единственным стоком, полученной путем добавления фиктивного источника и фиктивного стока, и наоборот.

26.1-4. Докажите лемму 26.1.

26.1-5. Для транспортной сети $G = (V,E)$ и потока f, показанных на рис. 26.1*б*, найдите пару подмножеств $X, Y \subseteq V$, для которых $f(X,Y) = -f(V -$ $- X, Y)$. Затем найдите пару подмножеств $X, Y \subseteq V$, для которых $f(X,Y) \neq -f(V-X,Y)$.

26.1-6. Пусть дана транспортная сеть $G = (V,E)$, и пусть f_1 и f_2 — функции, отображающие $V \times V$ в $\mathbf{R}$. ***Суммой потоков*** $f_1 + f_2$ является функция,

отображающая $V \times V$ в $\mathbf{R}$ и определяемая как

$$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v) \tag{26.4}$$

для всех u, $v \in V$. Если f_1 и f_2 — потоки в G, то каким из трех свойств потоков удовлетворяет сумма потоков $f_1 + f_2$, и какие из них могут нарушаться?

26.1-7. Пусть f — поток в сети, а α — некоторое действительное число. ***Произведение потока на скаляр*** (scalar flow product), обозначаемое как αf, — это функция, отображающая $V \times V$ в $\mathbf{R}$, такая что

$$(\alpha f)(u, v) = \alpha \cdot f(u, v).$$

Докажите, что потоки в сети образуют ***выпуклое множество*** (convex set), т.е. покажите, что если f_1 и f_2 являются потоками, то $\alpha f_1 + (1 - \alpha) f_2$ также является потоком для любых $0 \leqslant \alpha \leqslant 1$.

26.1-8. Сформулируйте задачу о максимальном потоке в виде задачи линейного программирования.

26.1-9. У профессора двое детей, которые, к сожалению, терпеть не могут друг друга. Проблема настолько серьезна, что они не только не хотят вместе ходить в школу, но каждый даже отказывается заходить в квартал, в котором в этот день побывал другой. При этом они допускают, что их пути могут пересекаться на углу того или иного квартала. К счастью, и дом профессора, и школа расположены на углах кварталов, однако профессор не уверен, возможно ли отправить обоих детей в одну школу. У профессора есть карта города. Покажите, как сформулировать задачу о возможности отправить детей в одну и ту же школу в виде задачи о максимальном потоке.

26.2 Метод Форда-Фалкерсона

В данном разделе представлен метод Форда-Фалкерсона для решения задачи о максимальном потоке. Мы называем его методом, а не алгоритмом, поскольку он допускает несколько реализаций с различным временем выполнения. Метод Форда-Фалкерсона базируется на трех важных концепциях, которые выходят за рамки данного метода и применяются во многих потоковых алгоритмах и задачах. Это — остаточные сети, увеличивающие пути и разрезы. Данные концепции лежат в основе важной теоремы о максимальном потоке и минимальном разрезе (теорема 26.7), которая определяет значение максимального потока с помощью разрезов транспортной сети. В заключение данного раздела мы предложим одну

конкретную реализацию метода Форда-Фалкерсона и проанализируем время ее выполнения.

Метод Форда-Фалкерсона является итеративным. Вначале величине потока присваивается значение 0: $f(u,v) = 0$ для всех $u, v \in V$. На каждой итерации величина потока увеличивается посредством поиска "увеличивающего пути" (т.е. некого пути от источника s к стоку t, вдоль которого можно послать больший поток) и последующего увеличения потока. Этот процесс повторяется до тех пор, пока уже невозможно отыскать увеличивающий путь. В теореме о максимальном потоке и минимальном разрезе будет показано, что по завершении данного процесса получается максимальный поток.

FORD_FULKERSON_METHOD(G, s, t)

1 Задаем начальное значение потока f равным 0
2 **while** (Пока) существует увеличивающий путь p
3 **do** увеличиваем поток f вдоль пути p
4 **return** f

Остаточные сети

Интуитивно понятно, что если заданы некоторая транспортная сеть и поток, то остаточная сеть — это сеть, состоящая из ребер, допускающих увеличение потока. Более строго, пусть задана транспортная сеть $G = (V, E)$ с источником s и стоком t. Пусть f — некоторый поток в G. Рассмотрим пару вершин $u, v \in \in V$. Величина дополнительного потока, который мы можем направить из u в v, не превысив пропускную способность $c(u,v)$, является ***остаточной пропускной способностью*** (residual capacity) ребра (u,v), и задается формулой

$$c_f(u,v) = c(u,v) - f(u,v). \tag{26.5}$$

Например, если $c(u,v) = 16$ и $f(u,v) = 11$, то $f(u,v)$ можно увеличить на $c_f(u,v) = 5$, не нарушив ограничение пропускной способности для ребра (u,v). Когда поток $f(u,v)$ отрицателен, остаточная пропускная способность $c_f(u,v)$ больше, чем пропускная способность $c(u,v)$. Так, если $c(u,v) = 16$ и $f(u,v) = = -4$, то остаточная пропускная способность $c_f(u,v) = 20$. Эту ситуацию можно интерпретировать следующим образом: существует поток величиной 4 единицы из v в u, который можно аннулировать, направив 4 единицы потока из u в v. Затем можно направить еще 16 единиц из u в v, не нарушив ограничение пропускной способности для ребра (u,v). Таким образом, начиная с потока $f(u,v) = -4$, мы направили дополнительные 20 единиц потока, прежде чем достигли ограничения пропускной способности.

Для заданной транспортной сети $G = (V, E)$ и потока f, ***остаточной сетью*** (residual network) в G, порожденной потоком f, является сеть $G_f = (V, E_f)$, где

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}.$$

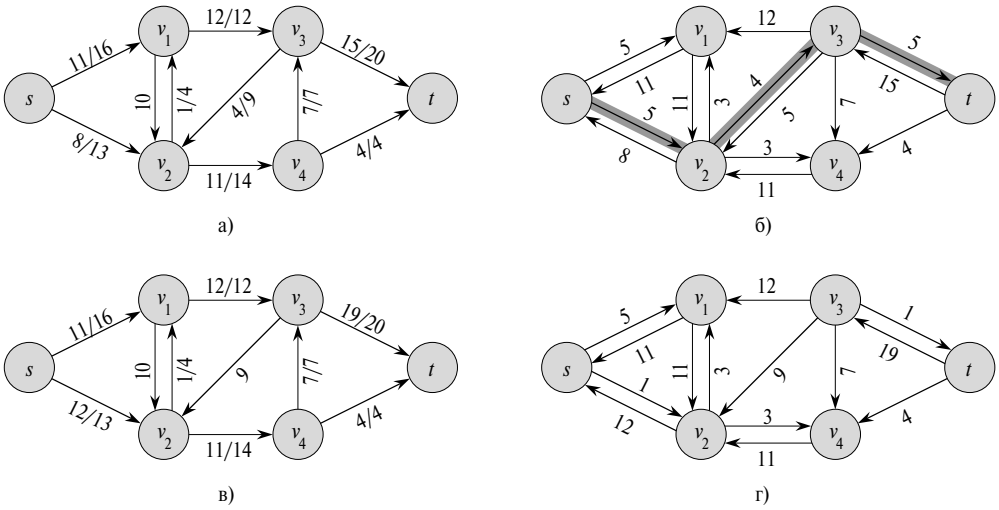


Рис. 26.3. а) Транспортная сеть G и поток f, представленные на рис. 26.1*б*. б) Остаточная сеть G_f с выделенным увеличивающим путем; его остаточная пропускная способность равна 4. в) Поток в сети G, полученный в результате увеличения потока вдоль пути p на величину его остаточной пропускной способности 4. г) Остаточная сеть, порожденная потоком, показанным в части в)

Таким образом, как и отмечалось выше, по каждому ребру остаточной сети, или ***остаточному ребру*** (residual edge), можно направить поток, больший 0. На рис. 26.3*а* воспроизведены транспортная сеть G и поток f, представленные на рис. 26.1*б*, а на рис. 26.3*б* показана соответствующая остаточная сеть G_f.

Ребрами E_f являются или ребра E, или обратные им. Если $f(u,v) < c(u,v)$ для некоторого ребра $(u,v) \in E$, то $c_f(u,v) = c(u,v) - f(u,v) > 0$ и $(u,v) \in E_f$. Если $f(u,v) > 0$ для некоторого ребра $(u,v) \in E$, то $f(v,u) < 0$. В таком случае $c_f(v,u) = c(v,u) - f(v,u) > 0$ и, следовательно, $(v,u) \in E_f$. Если в исходной сети нет ни ребра (u,v), ни (v,u), то $c(u,v) = c(v,u) = 0$, $f(u,v) = f(v,u) = 0$ (согласно результатам упражнения 26.1-1), и $c_f(u,v) = c_f(v,u) = 0$. Таким образом, можно сделать вывод, что ребро (u,v) может оказаться в остаточной сети только в том случае, если хотя бы одно из ребер (u,v) или (v,u) присутствует в исходной транспортной сети, поэтому

$$|E_f| \leqslant 2|E|.$$

Обратите внимание, что остаточная сеть G_f является транспортной сетью со значениями пропускных способностей, заданными c_f. Следующая лемма показывает, как поток в остаточной сети связан с потоком в исходной транспортной сети.

Лемма 26.2. Пусть $G = (V,E)$ — транспортная сеть с источником s и стоком t, а f — поток в G. Пусть G_f — остаточная сеть в G, порожденная потоком f, а f' —

поток в G_f. Тогда сумма потоков $f + f'$, определяемая уравнением (26.4), является потоком в G, и величина этого потока равна $|f + f'| = |f| + |f'|$.

Доказательство. Необходимо проверить, выполняются ли ограничения антисимметричности, пропускной способности и сохранения потока. Для подтверждения антисимметричности заметим, что для всех $u, v \in V$, справедливо

$$\begin{aligned}\left(f + f'\right)(u, v) &= f(u, v) + f'(u, v) = \\ &= -f(v, u) - f'(v, u) = \\ &= -\left(f(v, u) + f'(v, u)\right) = \\ &= -\left(f + f'\right)(v, u).\end{aligned}$$

Покажем соблюдение ограничений пропускной способности. Заметим, что $f'(u, v) \leqslant c_f(u, v)$ для всех $u, v \in V$. Поэтому, согласно уравнению (26.5),

$$\begin{aligned}\left(f + f'\right)(u, v) &= f(u, v) + f'(u, v) \leqslant \\ &\leqslant f(u, v) + (c(u, v) - f(u, v)) = \\ &= c(u, v).\end{aligned}$$

Что касается сохранения потока, заметим, что для всех $u \in V - \{s, t\}$ справедливо равенство

$$\begin{aligned}\sum_{v \in V}\left(f + f'\right)(u, v) &= \sum_{v \in V}\left(f(u, v) + f'(u, v)\right) = \\ &= \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) = \\ &= 0 + 0 = 0.\end{aligned}$$

И наконец,

$$\begin{aligned}\left|f + f'\right| &= \sum_{v \in V}\left(f + f'\right)(s, v) = \\ &= \sum_{v \in V}\left(f(s, v) + f'(s, v)\right) = \\ &= \sum_{v \in V} f(s, v) + \sum_{v \in V} f'(s, v) = \\ &= |f| + \left|f'\right|.\end{aligned}$$

■

Увеличивающие пути

Для заданных транспортной сети $G = (V, E)$ и потока f ***увеличивающим путем*** (augmenting path) p является простой путь из s в t в остаточной сети G_f.

Согласно определению остаточной сети, каждое ребро (u, v) увеличивающего пути допускает некоторый дополнительный положительный поток из u в v без нарушения ограничения пропускной способности для данного ребра.

Выделенный путь на рис. 26.3*б* является увеличивающим путем. Рассматривая представленную на рисунке остаточную сеть G_f как некоторую транспортную сеть, можно увеличивать поток вдоль каждого ребра данного пути вплоть до 4 единиц, не нарушая ограничений пропускной способности, поскольку наименьшая остаточная пропускная способность на данном пути составляет $c_f(v_2, v_3) = 4$. Максимальная величина, на которую можно увеличить поток вдоль каждого ребра увеличивающего пути p, называется ***остаточной пропускной способностью*** (residual capacity) p и задается формулой

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ принадлежит } p\}.$$

Следующая лемма, доказательство которой предлагается провести в качестве упражнения 26.2-7, более строго формулирует приведенные выше рассуждения.

Лемма 26.3. Пусть $G = (V, E)$ — транспортная сеть, а f — некоторый поток в G, и пусть p — некоторый увеличивающий путь в G_f. Определим функцию $f_p : V \times V \to \mathbf{R}$ следующим образом:

$$f_p(u, v) = \begin{cases} c_f(p) & \text{если } (u, v) \text{ принадлежит } p, \\ -c_f(p) & \text{если } (v, u) \text{ принадлежит } p, \\ 0 & \text{в противном случае.} \end{cases} \tag{26.6}$$

Тогда f_p является потоком в G и его величина составляет $|f_p| = c_f(p) > 0$. ■

Вытекающее из данной леммы следствие показывает, что если добавить f_p к f, то мы получим новый поток в G, величина которого ближе к максимальной. На рис. 26.3*в* показан результат добавления f_p, представленного на рис. 26.3*б*, к f, показанному на рис. 26.3*а*.

Следствие 26.4. Пусть $G = (V, E)$ — транспортная сеть, а f — некоторый поток в G, и пусть p — некоторый увеличивающий путь в G_f. Пусть f_p определен в соответствии с уравнением (26.6). Определим функцию $f' : V \times V \to \mathbf{R}$ как $f' = f + f_p$. Тогда f' является потоком в G и имеет величину $|f'| = |f| + |f_p| > |f|$.

Доказательство. Непосредственно вытекает из лемм 26.2 и 26.3. ■

Разрезы транспортных сетей

В методе Форда-Фалкерсона производится неоднократное увеличение потока вдоль увеличивающих путей до тех пор, пока не будет найден максимальный

поток. В теореме о максимальном потоке и минимальном разрезе, которую мы вскоре докажем, утверждается, что поток является максимальным тогда и только тогда, когда его остаточная сеть не содержит увеличивающих путей. Однако для доказательства данной теоремы нам понадобится ввести понятие разреза транспортной сети.

Разрезом (cut) (S,T) транспортной сети $G=(V,E)$ называется разбиение множества вершин на множества S и $T=V-S$, такие что $s\in S$, а $t\in T$. (Это определение аналогично определению разреза, которое использовалось применительно к минимальным связующим деревьям в главе 23, однако здесь речь идет о разрезе в ориентированном графе, а не в неориентированном, и мы требуем, чтобы $s\in S$, а $t\in T$.) Если f — поток, то ***чистый поток*** (net flow) через разрез (S,T) по определению равен $f(S,T)$. ***Пропускной способностью*** (capacity) разреза (S,T) является $c(S,T)$. ***Минимальным разрезом*** (minimum cut) сети является разрез, пропускная способность которого среди всех разрезов сети минимальна.

На рис. 26.4 показан разрез $(\{s,v_1,v_2\},\{v_3,v_4,t\})$ транспортной сети, представленной на рис. 26.1*б*. Чистый поток через данный разрез равен

$$f(v_1,v_3)+f(v_2,v_3)+f(v_2,v_4)=12+(-4)+11=19,$$

а пропускная способность этого разреза равна

$$c(v_1,v_3)+c(v_2,v_4)=12+14=26.$$

Обратите внимание, что чистый поток через разрез может включать в себя отрицательные потоки между вершинами, но пропускная способность разреза слагается исключительно из неотрицательных значений. Иными словами, чистый поток через разрез (S,T) составляется из положительных потоков в обоих направлениях; положительный поток из S в T прибавляется, а положительный поток из

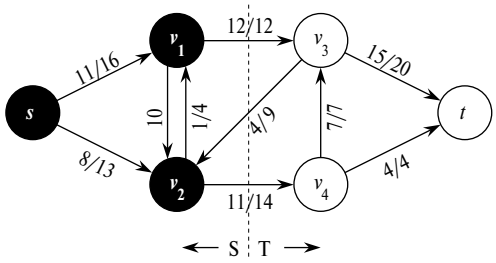


Рис. 26.4. Разрез (S,T) транспортной сети, представленной на рис. 26.1*б*, $S=\{s,v_1,v_2\}$, $T=\{v_3,v_4,t\}$. Вершины, принадлежащие S, отмечены черным цветом, а вершины T — белым

T в S вычитается. С другой стороны, пропускная способность разреза (S,T) вычисляется только по ребрам, идущим из S в T. Ребра, ведущие из T в S, не участвуют в вычислении $c(S,T)$.

Следующая лемма показывает, что чистый поток через любой разрез одинаков и равен величине потока.

Лемма 26.5. Пусть f — некоторый поток в транспортной сети G с источником s и стоком t, и пусть (S,T) — разрез G. Тогда чистый поток через (S,T) равен $f(S,T) = |f|$.

Доказательство. Заметим, что согласно свойству сохранения потока $f(S-s, V) = 0$, так что

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) = && \text{(согласно лемме 26.1, часть (3))} \\ &= f(S,V) = && \text{(согласно лемме 26.1, часть (1))} \\ &= f(s,V) + f(S-s,V) = && \text{(согласно лемме 26.1, часть (3))} \\ &= f(s,V) = && \text{(поскольку } f(S-s,V) = 0\text{)} \\ &= |f|\,. \end{aligned}$$

■

Непосредственным следствием леммы 26.5 является доказанный ранее результат — уравнение (26.3) — что величина потока равна суммарному потоку, входящему в сток.

Другое следствие леммы 26.5 показывает, как пропускные способности разрезов можно использовать для определения границы величины потока.

Следствие 26.6. Величина любого потока f в транспортной сети G не превышает пропускную способность произвольного разреза G.

Доказательство. Пусть (S,T) — произвольный разрез G, а f — некоторый поток. Согласно лемме 26.5 и ограничениям пропускной способности,

$$|f| = f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) \leqslant \sum_{u \in S} \sum_{v \in T} c(u,v) = c(S,T)\,.$$

■

Непосредственно из следствия 26.6 вытекает, что максимальный поток в сети не превышает пропускной способности минимального разреза. Сейчас мы сформулируем и докажем важную теорему о максимальном потоке и минимальном разрезе, в которой утверждается, что значение максимального потока равно пропускной способности минимального разреза.

Теорема 26.7 (О максимальном потоке и минимальном разрезе). Если f — некоторый поток в транспортной сети $G = (V,E)$ с источником s и стоком t, то следующие утверждения эквивалентны.

1. f — максимальный поток в G.
2. Остаточная сеть G_f не содержит увеличивающих путей.
3. $|f| = c(S,T)$ для некоторого разреза (S,T) сети G.

Доказательство. (1) $\Rightarrow$ (2): Предположим противное: пусть f является максимальным потоком в G, но G_f содержит увеличивающий путь p. Согласно следствию 26.4, сумма потоков $f + f_p$, где f_p задается уравнением (26.6), является потоком в G, величина которого строго больше, чем $|f|$, что противоречит предположению, что f — максимальный поток.

(2) $\Rightarrow$ (3): Предположим, что G_f не содержит увеличивающего пути, т.е. G_f не содержит пути из s в t. Определим

$$S = \{v \in V : \text{в } G_f \text{ существует путь из } s \text{ в } v\}$$

и $T = V - S$. Разбиение (S,T) является разрезом: очевидно, что $s \in S$, а $t \notin S$, поскольку в G_f не существует пути из s в t. Для каждой пары вершин $u \in S$, $v \in T$ справедливо соотношение $f(u,v) = c(u,v)$, поскольку в противном случае $(u,v) \in E_f$ и v следует поместить во множество S. Следовательно, согласно лемме 26.5, $|f| = f(S,T) = c(S,T)$.

(3) $\Rightarrow$ (1): Согласно следствию 26.6, $|f| \leqslant c(S,T)$ для всех разрезов (S,T), поэтому из условия $|f| = c(S,T)$ следует, что f — максимальный поток. ■

Базовый алгоритм Форда-Фалкерсона

При выполнении каждой итерации метода Форда-Фалкерсона мы находим некоторый увеличивающий путь p, и поток f вдоль каждого ребра данного пути увеличивается на величину остаточной пропускной способности $c_f(p)$. Приведенная далее реализация данного метода вычисляет максимальный поток в графе $G = (V,E)$ путем обновления потока $f[u,v]$ между каждой парой вершин u и v, соединенных ребром[1]. Если вершины u и v не связаны ребром ни в одном направлении, неявно предполагается, что $f[u,v] = 0$. Предполагается, что значения пропускных способностей задаются вместе с графом и $c(u,v) = 0$, если $(u,v) \notin E$. Остаточная пропускная способность $c_f(u,v)$ вычисляется по формуле (26.5). В коде процедуры $c_f(p)$ в действительности является просто временной переменной, в которой хранится остаточная пропускная способность пути p.

[1]Квадратные скобки используются, когда идентификатор (в данном случае f) трактуется как изменяемое поле, а когда это функция — используются круглые скобки.

```
FORD_FULKERSON(G, s, t)
1  for (для) каждого ребра (u, v) ∈ E[G]
2       do f[u, v] ← 0
3          f[v, u] ← 0
4  while существует путь p из s в t в остаточной сети G_f
5       do c_f(p) ← min {c_f(u, v) : (u, v) принадлежит p}
6          for (для) каждого ребра (u, v) in p
7               do f[u, v] ← f[u, v] + c_f(p)
8                  f[v, u] ← −f[u, v]
```

Приведенный псевдокод алгоритма FORD_FULKERSON является расширением приведенного ранее псевдокода FORD_FULKERSON_METHOD. На рис. 26.5 показаны результаты каждой итерации при тестовом выполнении. Строки 1–3 инициализируют поток f значением 0. В цикле **while** в строках 4–8 выполняется неоднократный поиск увеличивающего пути p в G_f, и поток f вдоль пути p увеличивается на остаточную пропускную способность $c_f(p)$. Когда увеличивающих путей больше нет, поток f является максимальным. Остаточная сеть на рис. 26.5*а* — это исходная сеть G; поток f, показанный на рис. 26.5*д*, является максимальным потоком.

Анализ метода Форда-Фалкерсона

Время выполнения процедуры FORD_FULKERSON зависит от того, как именно выполняется поиск увеличивающего пути p в строке 4. При неудачном методе поиска алгоритм может даже не завершиться: величина потока будет последовательно увеличиваться, но она не обязательно сходится к максимальному значению потока[2]. Если увеличивающий путь выбирается с использованием поиска в ширину (который мы рассматривали в разделе 22.2), алгоритм выполняется за полиномиальное время. Прежде чем доказать этот результат, получим простую границу времени выполнения для случая, когда увеличивающий путь выбирается произвольным образом, а все значения пропускных способностей являются целыми числами.

На практике задача поиска максимального потока чаще всего возникает в целочисленной постановке. Если пропускные способности — рациональные числа, можно использовать соответствующее масштабирование, которое сделает их целыми. В таком предположении непосредственная реализация процедуры FORD_FULKERSON имеет время работы $O(E\,|f^*|)$, где f^* — максимальный поток, найденный данным алгоритмом. Анализ проводится следующим образом. Выполнение строк 1–3 занимает время $\Theta(E)$. Цикл **while** в строках 4–8 выполняется не бо-

[2]Метод Форда-Фалкерсона может работать бесконечно, только если значения пропускной способности ребер являются иррациональными числами.

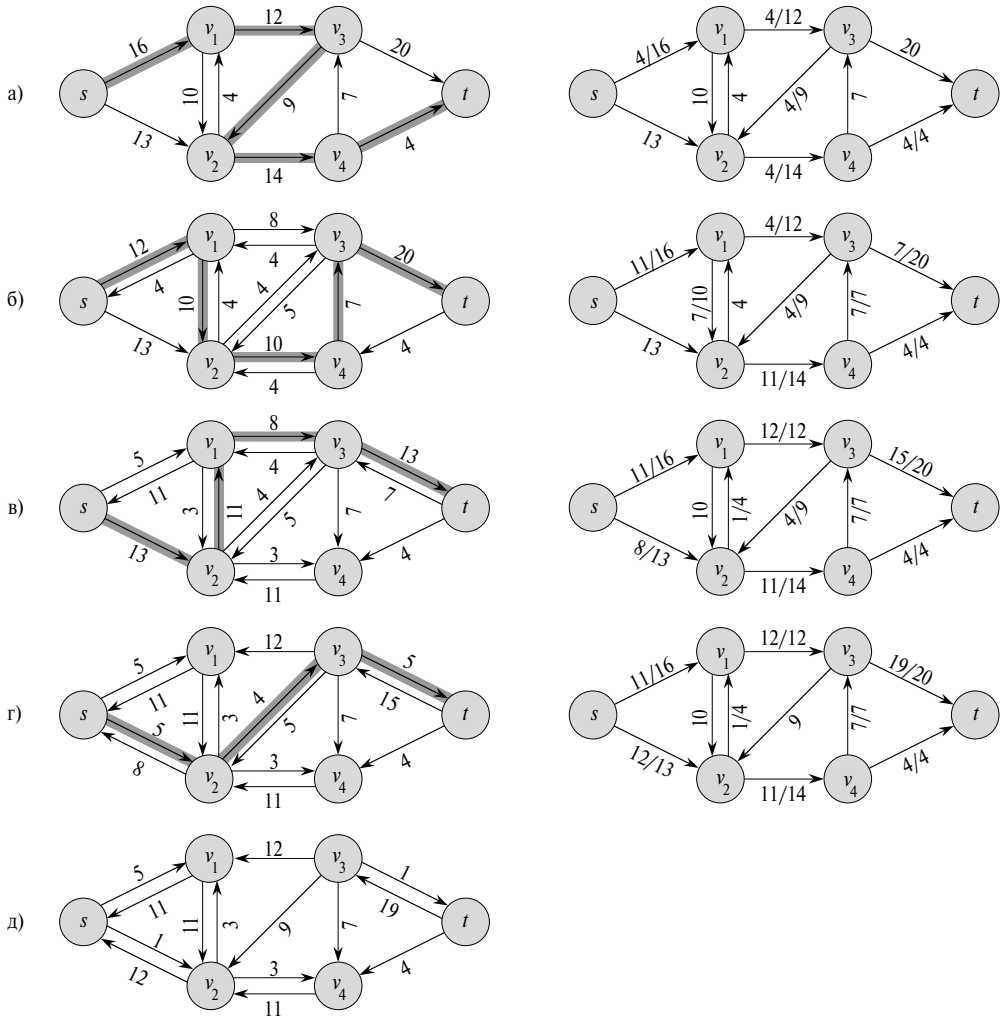


Рис. 26.5. Работа базового алгоритма Форда-Фалкерсона (последовательные итерации цикла **while**). В левой части каждого рисунка показана остаточная сеть G_f с выделенным увеличивающим путем p; в правой части показан новый поток f, который получается в результате прибавления f_p к f

лее $|f^*|$ раз, поскольку величина потока за каждую итерацию увеличивается по крайней мере на одну единицу.

Работа внутри цикла **while** зависит от того, насколько эффективно организовано управление структурой данных, используемой для реализации сети $G = (V, E)$. Предположим, что мы поддерживаем структуру данных, соответствующую ориентированному графу $G' = (V, E')$, где $E' = \{(u, v) : (u, v) \in E \text{ или } (v, u) \in E\}$.

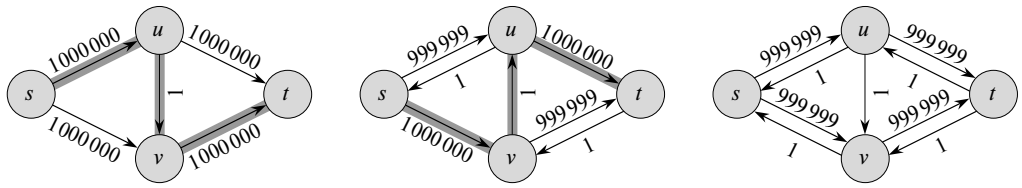


Рис. 26.6. Транспортная сеть, для которой выполнение процедуры FORD_FULKERSON может занимать время $\Theta(E|f^*|)$, $|f^*| = 2\,000\,000$

Ребра сети G являются также ребрами графа G', поэтому в этой структуре данных можно довольно легко хранить пропускные способности и потоки. Для данного потока f в G, ребра остаточной сети G_f состоят из всех ребер (u, v) графа G', таких что $c(u, v) - f[u, v] \neq 0$. Таким образом, время поиска пути в остаточной сети составляет $O(V + E') = O(E)$, если используется только поиск в глубину или поиск в ширину. Каждая итерация цикла **while** занимает время $O(E)$, так что в результате общее время выполнения процедуры FORD_FULKERSON составляет $O(E|f^*|)$.

Когда значения пропускных способностей являются целыми числами и оптимальное значение потока $|f^*|$ невелико, время выполнения алгоритма Форда-Фалкерсона достаточно неплохое. Но на рис. 26.6*а* показан пример того, что может произойти в простой транспортной сети, с большим значением $|f^*|$. Величина максимального потока в данной сети равна 2 000 000: 1 000 000 единиц потока идет по пути $s \to u \to t$, а другие 1 000 000 единиц идут по пути $s \to v \to t$. Если первым увеличивающим путем, найденным процедурой FORD_FULKERSON, является путь $s \to u \to v \to t$, как показано на рис. 26.6*а*, поток после первой итерации имеет значение 1. Полученная остаточная сеть показана на рис. 26.6*б*. Если в ходе выполнения второй итерации найден увеличивающий путь $s \to v \to u \to t$, как показано на рис. 26.6*б*, поток станет равным 2. На рис. 26.6*в* показана соответствующая остаточная сеть. Можно продолжать процедуру, выбирая увеличивающий путь $s \to u \to v \to t$ для итераций с нечетным номером $s \to v \to u \to t$ для итераций с четным номером. В таком случае нам придется выполнить 2 000 000 увеличений, при этом величина потока на каждом шаге увеличивается всего на 1 единицу.

Алгоритм Эдмондса-Карпа

Указанный недостаток метода Форда-Фалкерсона можно преодолеть, если реализовать вычисление увеличивающего пути p в строке 4 как поиск в ширину, т.е. если в качестве увеличивающего пути выбирается *кратчайший* путь из s

в t в остаточной сети, где каждое ребро имеет единичную длину (вес). Такая реализация метода Форда-Фалкерсона называется ***алгоритмом Эдмондса-Карпа*** (Edmonds-Karp algorithm). Докажем, что время выполнения алгоритма Эдмондса-Карпа составляет $O\left(V E^2\right)$.

Анализ зависит от расстояний между вершинами остаточной сети G_f. В следующей лемме длина кратчайшего пути из вершины u в v в остаточной сети G_f, где каждое ребро имеет единичную длину, обозначена как $\delta_f(u, v)$.

Лемма 26.8. Если для некоторой транспортной сети $G = (V, E)$ с источником s и стоком t выполняется алгоритм Эдмондса-Карпа, то для всех вершин $v \in$ $\in V - \{s, t\}$ длина кратчайшего пути $\delta_f(s, v)$ в остаточной сети G_f монотонно возрастает с каждым увеличением потока.

Доказательство. Предположим, что для некоторой вершины $v \in V - \{s, t\}$ существует такое увеличение потока, которое приводит к уменьшению длины кратчайшего пути из s в v, и покажем, что это предположение приведет нас к противоречию. Пусть f — поток, который был непосредственно перед первым увеличением, приведшим к уменьшению длины некого кратчайшего пути, а f' — поток сразу после этого увеличения. Пусть v — вершина с минимальной длиной кратчайшего пути $\delta'_f(s, v)$, которая уменьшилась в результате увеличения потока, т.е. $\delta'_f(s, v) <$ $< \delta_f(s, v)$. Пусть $p = s \rightsquigarrow u \to v$ — кратчайший путь от s к v в G'_f, такой что $(u, v) \in E'_f$ и

$$\delta'_f(s, u) = \delta'_f(s, v) - 1. \tag{26.7}$$

Исходя из того, как мы выбирали v, можно утверждать, что длина пути до вершины u не уменьшилась, т.е.

$$\delta'_f(s, u) \geqslant \delta_f(s, u). \tag{26.8}$$

Мы утверждаем, что в таком случае $(u, v) \notin E_f$. Почему? Если $(u, v) \in E_f$, тогда справедливо следующее:

$$\begin{aligned}
\delta_f(s, v) &\leqslant \delta_f(s, u) + 1 \leqslant && \text{(согласно лемме 24.10, неравенство треугольника)} \\
&\leqslant \delta'_f(s, u) + 1 = && \text{(согласно неравенству (26.8))} \\
&= \delta'_f(s, v) && \text{(согласно уравнению (26.7)),}
\end{aligned}$$

что противоречит предположению $\delta'_f(s, v) \leqslant \delta_f(s, v)$.

Теперь посмотрим, как может получиться, что $(u, v) \notin E_f$, но $(u, v) \in E'_f$? Увеличение должно привести к возрастанию потока из v в u. Алгоритм Эдмондса-Карпа всегда увеличивает поток вдоль кратчайших путей, поэтому последним

ребром кратчайшего пути из s в u в G_f является ребро (v, u). Следовательно,

$$\begin{aligned}\delta_f(s,v) &= \delta_f(s,u) - 1 \leqslant \\ &\leqslant \delta'_f(s,u) - 1 = \qquad \text{(согласно неравенству (26.8))} \\ &= \delta'_f(s,v) - 2 \qquad \text{(согласно уравнению (26.7))},\end{aligned}$$

что противоречит предположению $\delta'_f(s,v) < \delta_f(s,v)$, а значит, наше предположение о существовании вершины v не верно. ■

Следующая теорема устанавливает верхний предел количества итераций алгоритма Эдмондса-Карпа.

Теорема 26.9. Если для некоторой транспортной сети $G = (V, E)$ с источником s и стоком t выполняется алгоритм Эдмондса-Карпа, то общее число увеличений потока, выполняемое данным алгоритмам, составляет $O(VE)$.

Доказательство. Назовем ребро (u, v) остаточной сети G_f ***критическим*** (critical) для увеличивающего пути p, если остаточная пропускная способность p равна остаточной пропускной способности ребра (u, v), т.е. если $c_f(p) = c_f(u, v)$. После увеличения потока вдоль некоего увеличивающего пути, все критические ребра этого пути исчезают из остаточной сети. Кроме того, по крайней мере одно ребро любого увеличивающего пути должно быть критическим. Теперь покажем, что каждое из $|E|$ ребер может становиться критическим не более $|V|/2 - 1$ раз.

Пусть u и v — вершины из множества вершин V, соединенные некоторым ребром из множества E. Поскольку увеличивающие пути — это кратчайшие пути, то когда ребро (u, v) становится критическим первый раз, справедливо равенство

$$\delta_f(s,v) = \delta_f(s,u) + 1.$$

После того как поток увеличен, ребро (u, v) исчезает из остаточной сети. Оно не может появиться в другом увеличивающем пути, пока не будет уменьшен поток из u в v, а это может произойти только в том случае, если на некотором увеличивающем пути встретится ребро (v, u). Если в этот момент поток в сети G составлял f', справедливо следующее равенство:

$$\delta'_f(s,u) = \delta'_f(s,v) + 1.$$

Поскольку, согласно лемме 26.8, $\delta_f(s,v) \leqslant \delta'_f(s,v)$, получаем:

$$\delta'_f(s,u) = \delta'_f(s,v) + 1 \geqslant \delta_f(s,v) + 1 = \delta_f(s,u) + 2.$$

Следовательно, за время, прошедшее с момента, когда ребро (u, v) было критическим, до момента, когда оно становится критическим в следующий раз, расстояние до u от источника увеличивается не менее чем на 2. Расстояние до u от

источника в начальный момент было не меньше 0. Среди промежуточных вершин на кратчайшем пути из s в u не могут находиться s, u или t (поскольку наличие ребра (u, v) в кратчайшем пути подразумевает, что $u \neq t$). Следовательно, к тому моменту, когда вершина u станет недостижимой из источника (если такое произойдет), расстояние до нее будет не более $|V| - 2$. Таким образом, ребро (u, v) может стать критическим не более $(|V| - 2)/2 = |V|/2 - 1$ раз. Поскольку в остаточном графе имеется не более $O(E)$ пар вершин, которые могут быть соединены ребрами, общее количество критических ребер в ходе выполнения алгоритма Эдмондса-Карпа равно $O(V E)$. Каждый увеличивающий путь содержит по крайней мере одно критическое ребро, следовательно, теорема доказана. ■

Если увеличивающий путь находится посредством поиска в ширину, каждую итерацию процедуры FORD_FULKERSON можно выполнить за время $O(E)$, следовательно, суммарное время выполнения алгоритма Эдмондса-Карпа составляет $O(V E^2)$. Мы покажем, что алгоритмы проталкивания предпотока позволяют достичь еще лучших результатов. На основе алгоритма из раздела 26.4 построен метод, который позволяет достичь времени выполнения $O(V^2 E)$; этот метод является основой алгоритма со временем выполнения $O(V^3)$, рассматриваемого в разделе 26.5.

Упражнения

26.2-1. Чему равен поток через разрез $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$ на рис. 26.1*б*? Чему равна пропускная способность данного разреза?

26.2-2. Продемонстрируйте выполнение алгоритма Эдмондса-Карпа на примере транспортной сети, представленной на рис. 26.1*а*.

26.2-3. Укажите минимальный разрез на рис. 26.5, соответствующий показанному максимальному потоку. Какие два из представленных в примере увеличивающих путей взаимно уничтожают поток?

26.2-4. Докажите, что для любой пары вершин u, v и произвольных функций пропускной способности c и потока f справедливо соотношение $c_f(u, v) + c_f(v, u) = c(u, v) + c(v, u)$.

26.2-5. Вспомним предложенную в разделе 26.1 конструкцию, которая преобразует транспортную сеть с несколькими источниками и несколькими стоками в сеть с одним источником и одним стоком путем добавления ребер с бесконечной пропускной способностью. Докажите, что любой поток в полученной сети имеет конечную величину, если ребра исходной сети с множественными источниками и стоками имеют конечную пропускную способность.

26.2-6. Предположим, что каждый источник s_i в задаче со множественными источниками и стоками производит ровно p_i единиц потока, так что $f(s_i, V) = p_i$. Предположим также, что каждый сток t_j потребляет ровно q_j единиц, так что $f(V, t_j) = q_j$, где $\sum_i p_i = \sum_j q_j$. Покажите, как преобразовать данную задачу поиска потока f, удовлетворяющего указанным дополнительным ограничениям, в задачу поиска максимального потока в транспортной сети с одним источником и одним стоком.

26.2-7. Докажите лемму 26.3.

26.2-8. Покажите, что максимальный поток в сети $G = (V, E)$ всегда можно найти с помощью последовательности не более чем из $|E|$ увеличивающих путей. (*Указание*: считая, что максимальный поток известен, покажите, как следует выбирать пути.)

26.2-9. ***Запасом связности*** (edge connectivity) неориентированного графа назовем минимальное число ребер k, которые необходимо удалить, чтобы разъединить граф. Например, запас связности дерева равен 1, а запас связности циклической цепи вершин равен 2. Покажите, как определить запас связности неориентированного графа $G = (V, E)$ с помощью алгоритма максимального потока не более чем для $|V|$ транспортных сетей, каждая из которых содержит $O(V)$ вершин и $O(E)$ ребер.

26.2-10. Предположим, что транспортная сеть $G = (V, E)$ содержит симметричные ребра, т.е. $(u, v) \in E$ тогда и только тогда, когда $(v, u) \in E$. Покажите, что алгоритм Эдмондса-Карпа завершается после не более чем $|V|\,|E|/4$ итераций. (*Указание*: для произвольного ребра (u, v) проследите, как меняются $\delta(s, u)$ и $\delta(v, t)$ между последовательными моментами, когда ребро (u, v) становится критическим.)

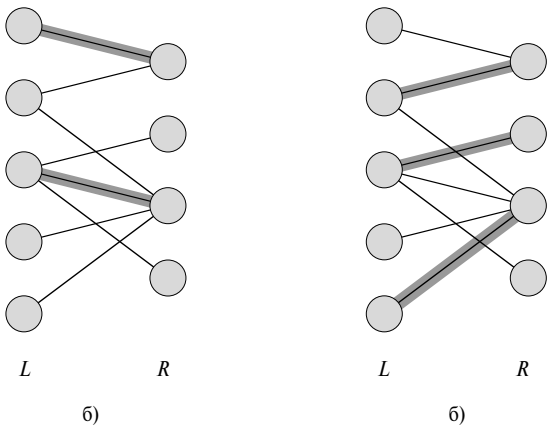
26.3 Максимальное паросочетание

Некоторые комбинаторные задачи можно легко свести к задачам поиска максимального потока. Одной из таких задач является задача определения максимального потока в сети с несколькими источниками и стоками, описанная в разделе 26.1 Существуют другие комбинаторные задачи, которые на первый взгляд имеют мало общего с транспортными сетями, однако могут быть сведены к задачам поиска максимального потока. В данном разделе рассматривается одна из подобных задач: поиск максимального паросочетания в двудольном графе (см. раздел Б.4). Чтобы решить данную задачу, мы воспользуемся свойством полноты, обеспечиваемым методом Форда-Фалкерсона. Мы также покажем, что с помощью метода Форда-Фалкерсона можно за время $O(V E)$ решить задачу поиска максимального паросочетания в двудольном графе $G = (V, E)$.

Задача поиска максимального паросочетания в двудольном графе

Пусть дан неориентированный граф $G = (V, E)$. ***Паросочетанием*** (matching) называется подмножество ребер $M \subseteq E$, такое что для всех вершин $v \in V$ в M содержится не более одного ребра, инцидентного v. Мы говорим, что вершина $v \in V$ является ***связанной*** (matched) паросочетанием M, если в M есть ребро, инцидентное v; в противном случае вершина v называется ***открытой*** (unmatched). Максимальным паросочетанием называется паросочетание максимальной мощности, т.е. такое паросочетание M, что для любого паросочетания M' $|M| \geqslant |M'|$. В данном разделе мы ограничимся рассмотрением задачи поиска максимальных паросочетаний в двудольных графах. Мы предполагаем, что множество вершин можно разбить на два подмножества $V = L \cup R$, где L и R не пересекаются, и все ребра из E проходят между L и R. Далее мы предполагаем, что каждая вершина из V имеет по крайней мере одно инцидентное ребро. Иллюстрация понятия паросочетания показана на рис. 26.7.

Задача поиска максимального паросочетания в двудольном графе имеет множество практических приложений. В качестве примера можно рассмотреть паросочетание множества машин L и множества задач R, которые должны выполняться одновременно. Наличие в E ребра (u, v) означает, что машина $u \in L$ может выполнять задачу $v \in R$. Максимальное паросочетание обеспечивает максимальную загрузку машин.



а) б)

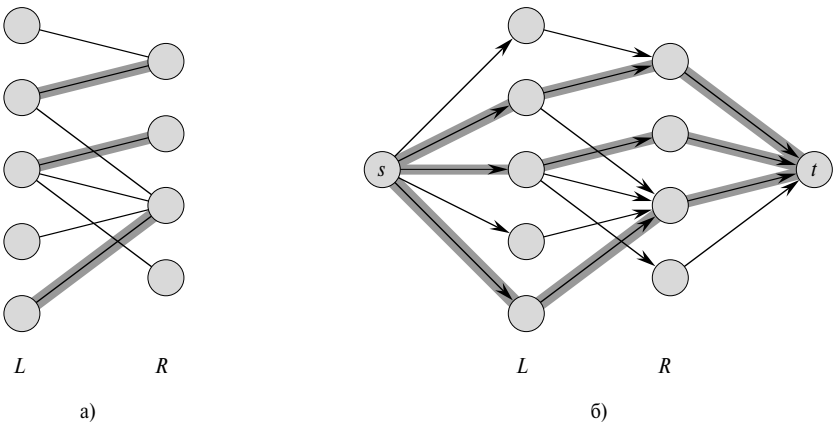
Рис. 26.7. Двудольный граф $G = (V, E)$ с разбиением вершин $V = L \cup R$. *а*) Паросочетание с мощностью 2. *б*) Максимальное паросочетание с мощностью 3.

Поиск максимального паросочетания в двудольном графе

С помощью метода Форда-Фалкерсона можно найти максимальное паросочетание в неориентированном двудольном графе $G = (V, E)$ за время, полиномиально зависящее от $|V|$ и $|E|$. Проблема состоит в том, чтобы построить транспортную сеть, потоки в которой соответствуют паросочетаниям, как показано на рис. 26.8. Определим для заданного двудольного графа G соответствующую транспортную сеть $G' = (V', E')$ следующим образом. Возьмем в качестве источника s и стока t новые вершины, не входящие в V, и пусть $V' = V \cup \{s, t\}$. Если разбиение вершин в графе G задано как $V = L \cup R$, ориентированными ребрами G' будут ребра E, направленные из L в R, а также $|V|$ новых ребер

$$E' = \{(s, u) : u \in L\} \cup \{(u, v) : u \in L, v \in R \text{ и } (u, v) \in E\} \cup \{(v, t) : v \in R\}.$$

Чтобы завершить построение, присвоим каждому ребру E' единичную пропускную способность. Поскольку каждая вершина из множества вершин V имеет по крайней мере одно инцидентное ребро, $|E| \geqslant |V|/2$. Таким образом, $|E| \leqslant |E'| = |E| + |V| \leqslant 3|E|$, так что $|E'| = \Theta(E)$.



а) б)

Рис. 26.8. Двудольный граф (*а*) и соответствующая ему транспортная сеть (*б*). Выделенные ребра обеспечивают максимальный поток и определяют максимальное паросочетание

Следующая лемма показывает, что паросочетание в G непосредственно соответствует некоторому потоку в соответствующей транспортной сети G'. Поток f в транспортной сети $G = (V, E)$ называется ***целочисленным*** (integer-valued), если значения $f(u, v)$ целые для всех $(u, v) \in V \times V$.

Лемма 26.10. Пусть $G = (V, E)$ — двудольный граф с разбиением вершин $V = L \cup R$, и пусть $G' = (V', E')$ — соответствующая ему транспортная сеть. Если

M — паросочетание в G, то существует целочисленный поток f в G', величина которого $|f| = |M|$. Справедливо и обратное утверждение: если f — целочисленный поток в G', то в G существует паросочетание M с мощностью $|M| = |f|$.

Доказательство. Покажем сначала, что паросочетанию M в графе G соответствует некоторый целочисленный поток f в сети G'. Определим f следующим образом. Если $(u, v) \in M$, то $f(s, u) = f(u, v) = f(v, t) = 1$ и $f(u, s) = = f(v, u) = f(t, v) = -1$. Для всех остальных ребер $(u, v) \in E'$ определим $f(u, v) = 0$. Нетрудно убедиться, что f обладает свойством антисимметричности, удовлетворяет ограничениям пропускной способности и сохранения потока.

Интуитивно понятно, что каждое ребро $(u, v) \in M$ соответствует единице потока в G', проходящего по маршруту $s \to u \to v \to t$. Кроме того, пути, порожденные ребрами из M, представляют собой непересекающиеся множества, не считая s и t. Чистый поток через разрез $(L \cup \{s\}, R \cup \{t\})$ равен $|M|$; следовательно, согласно лемме 26.5, величина потока равна $|f| = |M|$.

Чтобы доказать обратное, предположим, что f — некоторый целочисленный поток в G', и пусть

$$M = \{(u, v) : u \in L,\ v \in R \text{ и } f(u, v) > 0\}.$$

Каждая вершина $u \in L$ имеет только одно входящее ребро, а именно (s, u), и его пропускная способность равна 1. Следовательно, в каждую вершину $u \in L$ входит не более одной единицы положительного потока, и если она действительно входит, то из нее должна также выходить одна единица положительного потока согласно свойству сохранения потока. Более того, поскольку f — целочисленный поток, для каждой вершины $u \in L$ одна единица потока может входить не более чем по одному ребру и выходить не более чем по одному ребру. Таким образом, одна единица положительного потока входит в u тогда и только тогда, когда существует в точности одна вершина $v \in R$, такая что $f(u, v) = 1$, и из каждой вершины $u \in L$ выходит не более одного ребра, несущего положительный поток. Симметричные рассуждения применимы для каждой вершины $v \in R$. Следовательно, M является паросочетанием.

Чтобы показать, что $|M| = |f|$, заметим, что $f(s, u) = 1$ для каждой связанной вершины $u \in L$, и $f(u, v) = 0$ для каждого ребра $(u, v) \in E - M$. Следовательно,

$$|M| = f(L, R) = f\left(L, V'\right) - f(L, L) - f(L, s) - f(L, t) \qquad \text{согласно лемме 26.1.}$$

Приведенное выражение можно значительно упростить. Из свойства сохранения потока следует, что $f(L, V') = 0$; из леммы 26.1 следует, что $f(L, L) = 0$; согласно свойству антисимметричности, $-f(L, s) = f(s, L)$; и поскольку нет ребер,

ведущих из L к t, $f(L,t)=0$. Таким образом,

$$\begin{aligned}|M| &= f(s,L) = \\ &= f\left(s,V'\right) = && \text{(поскольку все ребра, выходящие из } s\text{, идут в } L\text{)} \\ &= |f| && \text{(согласно определению } |f|\text{).}\end{aligned}$$

■

На основании леммы 26.10 можно сделать вывод, что максимальное паросочетание в двудольном графе G соответствует максимальному потоку в соответствующей ему транспортной сети G', следовательно, можно находить максимальное паросочетание в G с помощью алгоритма поиска максимального потока в G'. Единственной проблемой в данных рассуждениях является то, что алгоритм поиска максимального потока может вернуть такой поток в G', в котором некоторое значение $f(u,v)$ оказывается нецелым, несмотря на то, что величина $|f|$ должна быть целой. Следующая теорема показывает, что такая проблема не может возникнуть при использовании метода Форда-Фалкерсона.

Теорема 26.11 (Теорема о целочисленности). Если функция пропускной способности c принимает только целые значения, то максимальный поток f, полученный с помощью метода Форда-Фалкерсона, обладает тем свойством, что значение потока $|f|$ является целочисленным. Более того, для всех вершин u и v величина $f(u,v)$ является целой.

Доказательство. Доказательство проводится индукцией по числу итераций. Его предлагается провести в качестве упражнения 26.3-2. ■

Докажем следствие из леммы 26.10.

Следствие 26.12. Мощность максимального паросочетания M в двудольном графе G равна величине максимального потока f в соответствующей транспортной сети G'.

Доказательство. Воспользуемся терминологией леммы 26.10. Предположим, что M — максимальное паросочетание в G, но соответствующий ему поток f в G' не максимален. Тогда в G' существует максимальный поток f', такой что $|f'| > |f|$. Поскольку пропускные способности в G' являются целочисленными, теорема 26.11 позволяет сделать вывод, что поток f' — целочисленный. Следовательно, f' соответствует некоторому паросочетанию M' в G с мощностью $|M'| = |f'| > |f| = |M|$, что противоречит нашему предположению о том, что M — максимальное паросочетание. Аналогично можно показать, что если f — максимальный поток в G', то соответствующее ему паросочетание является максимальным паросочетанием в G. ■

Таким образом, для заданного неориентированного двудольного графа G можно найти максимальное паросочетание путем создания транспортной сети G', применения метода Форда-Фалкерсона и непосредственного получения максимального паросочетания M по найденному максимальному целочисленному потоку f. Поскольку любое паросочетание в двудольном графе имеет мощность не более $\min(L, R) = O(V)$, величина максимального потока в G' составляет $O(V)$. Поэтому максимальное паросочетание в двудольном графе можно найти за время $O(V E') = O(V E)$, поскольку $|E'| = \Theta(E)$.

Упражнения

26.3-1. Примените алгоритм Форда-Фалкерсона для транспортной сети на рис. 26.8*б* и покажите остаточную сеть после каждого увеличения потока. Вершины из множества L пронумеруйте сверху вниз от 1 до 5, а вершины множества R — от 6 до 9. Для каждой итерации укажите лексикографически наименьший увеличивающий путь.

26.3-2. Докажите теорему 26.11.

26.3-3. Пусть $G = (V, E)$ — двудольный граф с разбиением вершин $V = L \cup R$, а G' — соответствующая ему транспортная сеть. Найдите верхнюю границу длины любого увеличивающего пути, найденного в G' в процессе выполнения процедуры FORD_FULKERSON.

⋆ 26.3-4. ***Полное паросочетание*** (perfect matching) — это паросочетание, в котором каждая вершина является связанной. Пусть $G = (V, E)$ — двудольный граф с разбиением вершин $V = L \cup R$, где $|L| = |R|$. Для любого подмножества $X \subseteq V$ определим ***окрестность*** (neighborhood) X как

$$N(X) = \{y \in V : (x, y) \in E \text{ для некоторого } x \in X\},$$

т.е. это множество вершин, смежных с какой-либо из вершин X. Докажите ***теорему Халла*** (Hall's theorem): полное паросочетание в G существует тогда и только тогда, когда $|A| \leqslant |N(A)|$ для любого подмножества $A \subseteq L$.

⋆ 26.3-5. Двудольный граф $G = (V, E)$, где $V = L \cup R$, называется ***d-регулярным*** (d-regular), если каждая вершина $v \in V$ имеет степень, в точности равную d. В каждом d-регулярном двудольном графе $|L| = |R|$. Докажите, что в каждом d-регулярном двудольном графе имеется паросочетание мощности $|L|$, показав, что минимальный разрез соответствующей транспортной сети имеет пропускную способность $|L|$.

⋆ 26.4 Алгоритмы проталкивания предпотока

В данном разделе мы рассмотрим подход к вычислению максимальных потоков, основанный на "проталкивании предпотока". В настоящее время многие наиболее асимптотически быстрые алгоритмы поиска максимального потока принадлежат данному классу, и на этом методе основаны реальные реализации алгоритмов поиска максимального потока. С помощью методов проталкивания предпотока можно решать и другие связанные с потоками задачи, например, задачу поиска потока с минимальными затратами. В данном разделе приводится разработанный Голдбергом (Goldberg) "обобщенный" алгоритм поиска максимального потока, для которого существует простая реализация с временем выполнения $O\left(V^2E\right)$, что лучше времени работы алгоритма Эдмондса-Карпа $O\left(V\,E^2\right)$. В разделе 26.5 данный универсальный алгоритм будет усовершенствован, что позволит получить алгоритм проталкивания предпотока, время выполнения которого составляет $O\left(V^3\right)$.

Алгоритмы проталкивания предпотока работают более локальным способом, чем метод Форда-Фалкерсона. Вместо того чтобы для поиска увеличивающего пути анализировать всю остаточную сеть, алгоритмы проталкивания предпотока обрабатывают вершины по одной, рассматривая только соседей данной вершины в остаточной сети. Кроме того, в отличие от метода Форда-Фалкерсона, алгоритмы проталкивания предпотока не обеспечивают в ходе своего выполнения свойство сохранения потока. При этом, однако, они поддерживают ***предпоток*** (preflow), который представляет собой функцию $f : V \times V \to \mathbf{R}$, обладающую свойством антисимметричности, удовлетворяющую ограничениям пропускной способности и следующему ослабленному условию сохранения потока: $f\left(V,u\right) \geqslant 0$ для всех вершин $u \in V - \{s\}$. Это количество называется ***избыточным потоком*** (excess flow), входящим в вершину u, и обозначается

$$e\left(u\right) = f\left(V,u\right). \tag{26.9}$$

Вершина $u \in V - \{s,t\}$ называется ***переполненной*** (overflowing), если $e\left(u\right) > 0$.

Мы начнем данный раздел с описания интуитивных соображений, приводящих к методу проталкивания предпотока. Затем рассмотрим две применяемые в данном методе операции: "проталкивание" предпотока и подъем (перемаркировка, relabeling) некоторой вершины. Наконец, мы представим универсальный алгоритм проталкивания предпотока и проанализируем его корректность и время выполнения.

Интуитивные соображения

Интуитивные соображения, лежащие в основе метода проталкивания предпотока, лучше всего проиллюстрировать на примере потоков жидкости: пусть транс-

портная сеть $G = (V, E)$ представляет собой систему труб с заданными пропускными способностями. Применяя данную аналогию, о методе Форда-Фалкерсона можно сказать, что каждый увеличивающий путь в сети вызывает дополнительный поток жидкости, без точек ветвления, который течет от источника к стоку. Метод Форда-Фалкерсона многократно добавляет дополнительные потоки, пока дальнейшее добавление станет невозможным.

В основе обобщенного алгоритма проталкивания предпотока лежат другие интуитивные соображения. Пусть, как и ранее, ориентированные ребра соответствуют трубам. Вершины, в которых трубы пересекаются, обладают двумя интересными свойствами. Во-первых, чтобы принять избыточный поток, каждая вершина снабжена выходящей трубой, ведущей в произвольно большой резервуар, способный накапливать жидкость. Во-вторых, каждая вершина, ее резервуар и все трубные соединения находятся на платформе, высота которой увеличивается по мере работы алгоритма.

Высота вершины определяет, как проталкивается поток: поток может проталкиваться только вниз, т.е. от более высокой вершины к более низкой. Поток может быть направлен и от нижестоящей вершины к вышестоящей, но операции проталкивания потока проталкивают его только вниз. Высота источника является фиксированной и составляет $|V|$, а фиксированная высота стока равна 0. Высота всех других вершин сначала равна 0 и увеличивается со временем. Алгоритм сначала посылает максимально возможный поток вниз от источника к стоку. Посылается количество, в точности достаточное для заполнения всех выходящих из источника труб до достижения их пропускной способности; таким образом посылается поток, равный пропускной способности разреза $(s, V - s)$. Когда поток впервые входит в некоторую транзитную вершину, он накапливается в ее резервуаре. Отсюда он со временем проталкивается вниз.

Может случиться так, что все трубы, выходящие из вершины u и еще не заполненные потоком, ведут к вершинам, которые лежат на одном уровне с u или находятся выше нее. В этом случае, чтобы избавить переполненную вершину u от избыточного потока, необходимо увеличить ее высоту — провести операцию подъема (relabeling) вершины u. Ее высота увеличивается и становится на единицу больше, чем высота самой низкой из смежных с ней вершин, к которым ведут незаполненные трубы. Следовательно, после подъема вершины существует по крайней мере одна выходящая труба, по которой можно протолкнуть дополнительный поток.

В конечном итоге весь поток, который может пройти к стоку, оказывается там. Больше пройти не может, поскольку трубы подчиняются ограничениям пропускной способности; количество потока через любой разрез ограничено его пропускной способностью. Чтобы сделать предпоток “нормальным” потоком, алгоритм после этого посылает избытки, содержащиеся в резервуарах переполненных вершин, обратно к источнику, продолжая менять метки вершин, чтобы их высота

превышала фиксированную высоту источника $|V|$. Как будет показано, после того как резервуары окажутся пустыми, предпоток оказывается не только нормальным, но и максимальным потоком.

Основные операции

Как следует из предыдущих рассуждений, в алгоритме проталкивания предпотока выполняются две основные операции: проталкивание избытка потока от вершины к одной из соседних с ней вершин и подъем вершины. Применение этих операций зависит от высот вершин, которым мы сейчас дадим более точные определения.

Пусть $G = (V, E)$ — транспортная сеть с источником s и стоком t, а f — некоторый предпоток в G. Функция $h : V \to \mathbf{N}$ является ***функцией высоты*** (height function)[3], если $h(s) = |V|$, $h(t) = 0$ и

$$h(u) \leqslant h(v) + 1$$

для любого остаточного ребра $(u, v) \in E_f$. Сразу же можно сформулировать следующую лемму.

Лемма 26.13. Пусть $G = (V, E)$ — транспортная сеть, а f — некоторый предпоток в G, и пусть h — функция высоты, заданная на множестве V. Для любых двух вершин $u, v \in V$ справедливо следующее утверждение: если $h(u) > h(v) + 1$, то (u, v) не является ребром остаточного графа. ■

Операция проталкивания

Основная операция PUSH(u, v) может применяться тогда, когда u является переполненной вершиной, $c_f(u, v) > 0$ и $h(u) = h(v) + 1$. Представленный ниже псевдокод обновляет предпоток f в заданной сети $G = (V, E)$. Предполагается, что остаточные пропускные способности при заданных f и c можно вычислить за фиксированное время. Излишний поток, хранящийся в вершине u, поддерживается в виде атрибута $e[u]$, а высота вершины u — в виде атрибута $h[u]$. Выражение $d_f(u, v)$ — это временная переменная, в которой хранится количество потока, которое можно протолкнуть из u в v.

[3] В литературе функция высоты обычно называется “функцией расстояния” (distance function), а высота вершины называется “меткой расстояния” (distance label). Мы используем термин “высота”, поскольку он лучше согласуется с интуитивным обоснованием алгоритма. Высота вершины связана с ее расстоянием от стока t, которое можно найти с помощью поиска в ширину в G^{T}.

PUSH(u, v)
1 $\triangleright$ **Условия применения:** u переполнена, $c_f(u,v) > 0$, и $h[u] = h[v] + 1$.
2 $\triangleright$ **Действие:** Проталкивает $d_f(u,v) = \min(e[u], c_f(u,v))$
единиц потока из u в v.
3 $d_f(u,v) \leftarrow \min(e[u], c_f(u,v))$
4 $f[u,v] \leftarrow f[u,v] + d_f(u,v)$
5 $f[v,u] \leftarrow -f[u,v]$
6 $e[u] \leftarrow e[u] - d_f(u,v)$
7 $e[v] \leftarrow e[v] + d_f(u,v)$

Процедура PUSH работает следующим образом. Предполагается, что вершина u имеет положительный избыток $e[u]$ и остаточная пропускная способность ребра (u,v) положительна. Тогда можно увеличить поток из u в v на величину $d_f(u,v) = \min(e[u], c_f(u,v))$, при этом избыток $e[u]$ не становится отрицательным и не будет превышена пропускная способность $c(u,v)$. В строке 3 вычисляется значение $d_f(u,v)$, после чего в строках 4–5 обновляется f, а в строках 6–7 обновляется e. Таким образом, если функция f являлась предпотоком перед вызовом процедуры PUSH, она останется предпотоком и после ее выполнения.

Обратите внимание, что в коде процедуры PUSH ничто не зависит от высот вершин u и v; тем не менее, мы запретили вызов процедуры, если не выполнено условие $h[u] = h[v] + 1$. Таким образом, избыточный поток проталкивается вниз только при разности высот, равной 1. Согласно лемме 26.13, между двумя вершинами, высоты которых отличаются более чем на 1, не существует остаточных ребер, а значит, поскольку атрибут h является функцией высоты, мы ничего не добьемся, разрешив проталкивать вниз поток при разности высот, превышающей 1.

Процедура PUSH(u,v) называется ***проталкиванием*** (push) из u к v. Если операция проталкивания применяется к некоторому ребру (u,v), выходящему из вершины u, будем говорить, что операция проталкивания применяется к u. Если в результате ребро (u,v) становится ***насыщенным*** (saturated) (после проталкивания $c_f(u,v) = 0$), то это ***насыщающее проталкивание*** (saturating push), в противном случае это ***ненасыщающее проталкивание*** (nonsaturating push). Если ребро насыщено, оно не входит в остаточную сеть. Один из результатов ненасыщающего проталкивания характеризует следующая лемма.

Лемма 26.14. После ненасыщающего проталкивания из u в v вершина u более не является переполненной.

Доказательство. Поскольку проталкивание ненасыщающее, количество посланного потока должно быть равно величине $e[u]$ непосредственно перед проталкиванием. Поскольку избыток $e[u]$ уменьшается на эту величину, после проталкивания он становится равным 0. ■

Операция подъема

Основная операция RELABEL(u) применяется, если вершина u переполнена и $h[u] \leqslant h[v]$ для всех ребер $(u, v) \in E_f$. Иными словами, переполненную вершину u можно подвергнуть подъему, если все вершины v, для которых имеется остаточная пропускная способность от u к v, расположены не ниже u, так что протолкнуть поток из u нельзя. (Напомним, что по определению ни источник s, ни сток t не могут быть переполнены; следовательно, ни s, ни t нельзя подвергать подъему.)

RELABEL(u)
1 ▷ **Условия применения:** u переполнена и для всех $v \in V$
таких что $(u, v) \in E_f$, $h[u] \leq h[v]$.
2 ▷ **Действие:** увеличивает высоту u.
3 $h[u] \leftarrow 1 + \min\{h[v] : (u, v) \in E_f\}$

Когда вызывается операция RELABEL(u), мы говорим, что вершина u подвергается ***подъему*** (relabeled). Заметим, что когда производится подъем u, остаточная сеть E_f должна содержать хотя бы одно ребро, выходящее из u, чтобы минимизация в коде операции производилась по непустому множеству. Это свойство вытекает из предположения, что вершина u переполнена. Поскольку $e[u] > 0$, имеем $e[u] = f(V, u) > 0$ и, следовательно, должна существовать по крайней мере одна вершина v, такая что $f[v, u] > 0$. Но тогда

$$c_f(u, v) = c(u, v) - f[u, v] = c(u, v) + f[v, u] > 0,$$

откуда вытекает, что $(u, v) \in E_f$. Таким образом, операция RELABEL(u) дает u наибольшую высоту, допускаемую наложенными на функцию высоты ограничениями.

Универсальный алгоритм

Универсальный алгоритм проталкивания предпотока использует следующую процедуру для создания начального предпотока в транспортной сети:

INITIALIZE_PREFLOW(G, s)
1 **for** (для) каждой вершины $u \in V[G]$
2 **do** $h[u] \leftarrow 0$
3 $e[u] \leftarrow 0$
4 **for** (для) каждого ребра $(u, v) \in E[G]$
5 **do** $f[u, v] \leftarrow 0$
6 $f[v, u] \leftarrow 0$
7 $h[s] \leftarrow |V[G]|$
8 **for** (для) каждой вершины $u \in Adj[s]$

```
 9      do f[s,u] ← c(s,u)
10         f[u,s] ← −c(s,u)
11         e[u] ← c(s,u)
12         e[s] ← e[s] − c(s,u)
```

Процедура INITIALIZE_PREFLOW(G, s) создает начальный предпоток f, определяемый формулой

$$f[u,v] = \begin{cases} c(u,v) & \text{если } u = s, \\ -c(v,u) & \text{если } v = s, \\ 0 & \text{в противном случае.} \end{cases} \tag{26.10}$$

То есть каждое ребро, выходящее из источника s, заполняется до его пропускной способности, а все остальные ребра не несут потока. Для каждой вершины v, смежной с источником, начальное значение $e[v] = c(s,v)$, а начальное значение $e[s]$ устанавливается равным сумме этих значений с обратным знаком. Универсальный алгоритм начинает работу с начальной функцией высоты

$$h[u] = \begin{cases} |V| & \text{если } u = s, \\ 0 & \text{в противном случае.} \end{cases}$$

Это действительно функция высоты, поскольку единственными ребрами (u,v), для которых $h[u] > h[v] + 1$, являются ребра, для которых $u = s$, и эти ребра заполнены, а это означает, что их нет в остаточной сети.

Инициализация, за которой следует ряд операций проталкивания и подъема, выполняемых без определенного порядка, образует алгоритм GENERIC_PUSH_RELABEL:

```
GENERIC_PUSH_RELABEL(G)
1  INITIALIZE_PREFLOW(G, s)
2  while существует применимая операция проталкивания или подъема
3        do выбрать операцию проталкивания или подъема и выполнить ее
```

Следующая лемма утверждает, что до тех пор пока существует хотя бы одна переполненная вершина, применима хотя бы одна из этих операций.

Лемма 26.15 (Для переполненной вершины можно выполнить либо проталкивание, либо подъем). Пусть $G = (V, E)$ — транспортная сеть с источником s и стоком t, f — предпоток, а h — некоторая функция высоты для f. Если u — некоторая переполненная вершина, то к ней можно применить или операцию проталкивания, или операцию подъема.

Доказательство. Для любого остаточного ребра (u, v) выполняется соотношение $h(u) \leqslant h(v)+1$, поскольку h — функция высоты. Если к u не применима операция проталкивания, то для всех остаточных ребер (u, v) должно выполняться условие $h(u) < h(v) + 1$, откуда следует, что $h(u) \leqslant h(v)$. В этом случае к u можно применить операцию подъема. ■

Корректность метода проталкивания предпотока

Чтобы показать, что универсальный алгоритм проталкивания предпотока позволяет решить задачу максимального потока, сначала докажем, что после его завершения предпоток f является максимальным потоком. Затем докажем, что алгоритм завершается. Начнем с рассмотрения некоторых свойств функции высоты h.

Лемма 26.16 (Высота вершины никогда не уменьшается). При выполнении процедуры GENERIC_PUSH_RELABEL над транспортной сетью $G = (V, E)$, для любой вершины $u \in V$ ее высота $h[u]$ никогда не уменьшается. Более того, всякий раз, когда к вершине u применяется операция подъема, ее высота $h[u]$ увеличивается как минимум на 1.

Доказательство. Поскольку высота вершины меняется только при выполнении операции подъема, достаточно доказать второе утверждение леммы. Если вершина u должна подвергнуться подъему, то для всех вершин v, таких что $(u, v) \in E_f$, выполняется условие $h[u] \leqslant h[v]$. Таким образом, $h[u] < 1 + \min\{h[v] : (u, v) \in \in E_f\}$, и операция должна увеличить значение $h[u]$. ■

Лемма 26.17. Пусть $G = (V, E)$ — транспортная сеть с источником s и стоком t. Во время выполнения процедуры GENERIC_PUSH_RELABEL над сетью G атрибут h сохраняет свойства функции высоты.

Доказательство. Доказательство проводится индукцией по числу выполненных основных операций. Как уже отмечалось, вначале h является функцией высоты.

Утверждается, что если h — функция высоты, то после выполнения операции RELABEL(u) она останется функцией высоты. Если посмотреть на остаточное ребро $(u, v) \in E_f$, выходящее из u, то операция RELABEL(u) гарантирует, что после ее выполнения $h[u] \leqslant h[v] + 1$. Рассмотрим теперь некоторое остаточное ребро (w, u), входящее в u. Согласно лемме 26.16, $h[w] \leqslant h[u] + 1$ перед выполнением операции RELABEL(u); следовательно, после ее выполнения $h[w] < h[u] + 1$. Таким образом, операция RELABEL(u) оставляет h функцией высоты.

Теперь рассмотрим операцию PUSH(u, v). Данная операция может добавить ребро (v, u) к E_f или удалить ребро (u, v) из E_f. В первом случае имеем $h[v] = = h[u] - 1 < h[u] + 1$, так что h остается функцией высоты. Во втором случае

удаление ребра (u, v) из остаточной сети приводит к удалению соответствующего ограничения, так что h по-прежнему остается функцией высоты. ■

Следующая лемма характеризует важное свойство функций высоты.

Лемма 26.18. Пусть $G = (V, E)$ — транспортная сеть с источником s и стоком t, f — предпоток в G, а h — функция высоты, определенная на множестве V. Тогда не существует пути из источника s к стоку t в остаточной сети G_f.

Доказательство. Предположим, что в G_f существует некоторый путь $p = \langle v_0, v_1, \ldots, v_k \rangle$ из s в t, где $v_0 = s$, а $v_k = t$, и покажем, что это приводит к противоречию. Без потери общности можно считать, что p — простой путь, так что $k < |V|$. Для $i = 0, 1, \ldots, k-1$, ребра $(v_i, v_{i+1}) \in E_f$. Поскольку h — функция высоты, для $i = 0, 1, \ldots, k-1$ справедливы соотношения $h(v_i) \leqslant h(v_{i+1}) + 1$. Объединяя эти неравенства вдоль пути p, получим, что $h(s) \leqslant h(t) + k$. Но поскольку $h(t) = 0$, получаем $h(s) \leqslant k < |V|$, что противоречит требованию $h(s) = |V|$ к функции высоты. ■

Теперь покажем, что после завершения универсального алгоритма проталкивания предпотока вычисленный алгоритмом предпоток является максимальным потоком.

Теорема 26.19 (О корректности универсального алгоритма проталкивания предпотока). Если алгоритм GENERIC_PUSH_RELABEL, выполняемый над сетью $G = (V, E)$ с источником s и стоком t, завершается, то вычисленный им предпоток f является максимальным потоком в G.

Доказательство. Мы используем следующий инвариант цикла:

> Всякий раз, когда производится проверка условия цикла **while** в строке 2 процедуры GENERIC_PUSH_RELABEL, f является предпотоком.

Инициализация. Процедура INITIALIZE_PREFLOW делает f предпотоком.

Сохранение. Внутри цикла **while** в строках 2–3 выполняются только операции проталкивания и подъема. Операции подъема влияют только на атрибуты высоты, но не на величины потока, следовательно, от них не зависит, будет ли f предпотоком. Анализируя работу процедуры PUSH, мы доказали, что если f является предпотоком перед выполнением операции проталкивания, он остается предпотоком и после ее выполнения.

Завершение. По завершении процедуры каждая вершина из множества $V - \{s, t\}$ должна иметь избыток, равный 0, поскольку из лемм 26.15 и 26.17 и инварианта, что f всегда остается предпотоком, вытекает, что переполненных вершин нет. Следовательно, f является потоком. Поскольку h — функция

высоты, согласно лемме 26.18 не существует пути из s в t в остаточной сети G_f. Согласно теореме о максимальном потоке и минимальном разрезе (теорема 26.7), f является максимальным потоком. ■

Анализ метода проталкивания предпотока

Чтобы показать, что универсальный алгоритм проталкивания предпотока действительно завершается, найдем границу числа выполняемых им операций. Для каждого вида операций (подъем, насыщающее проталкивание и ненасыщающее проталкивание) имеется своя граница. Зная эти границы, несложно построить алгоритм, время работы которого $O\left(V^2E\right)$. Прежде чем проводить анализ, докажем важную лемму.

Лемма 26.20. Пусть $G = (V, E)$ — транспортная сеть с источником s и стоком t, а f — предпоток в G. Тогда для любой переполненной вершины u существует простой путь из u в s в остаточной сети G_f.

Доказательство. Пусть u — некоторая переполненная вершина, и пусть $U = \{v :$ в G_f существует простой путь из u в $v\}$. Предположим, что $s \notin U$, и покажем, что это приведет нас к противоречию. Обозначим $\bar{U} = V - U$.

Утверждается, что для каждой пары вершин $w \in \bar{U}$ и $v \in U$ выполняется соотношение $f(w, v) \leqslant 0$. Почему? Если $f(w, v) > 0$, то $f(v, w) < 0$, откуда в свою очередь вытекает, что $c_f(v, w) = c(v, w) - f(v, w) > 0$. Следовательно, существует ребро $(v, w) \in E_f$ и существует простой путь вида $u \rightsquigarrow v \to w$ в остаточной сети G_f, что противоречит тому, как мы выбирали w.

Таким образом, должно выполняться неравенство $f\left(\bar{U}, U\right) \leqslant 0$, поскольку каждое слагаемое в этом неявном суммировании неположительно, и, следовательно,

$$\begin{aligned} e(U) = f(V, U) = & \qquad \text{(из уравнения (26.9))} \\ = f\left(\bar{U}, U\right) + f(U, U) = & \qquad \text{(согласно лемме 26.1 (часть 3))} \\ = f\left(\bar{U}, U\right) \leqslant 0 & \qquad \text{(согласно лемме 26.1 (часть 1)).} \end{aligned}$$

Излишки неотрицательны для всех вершин из множества $V - \{s\}$; поскольку мы предположили, что $U \subseteq V - \{s\}$, то для всех вершин $v \in U$ должно выполняться $e(v) = 0$. В частности, $e(u) = 0$, что противоречит предположению о том, что u переполнена. ■

Следующая лемма устанавливает границы высот вершин, а вытекающее из нее следствие устанавливает предел общего числа выполненных операций подъема.

Лемма 26.21. Пусть $G = (V, E)$ — транспортная сеть с источником s и стоком t. В любой момент в процессе выполнения процедуры GENERIC_PUSH_RELABEL в сети G для всех вершин $u \in V$ выполняется соотношение $h[u] \leqslant 2|V| - 1$.

Доказательство. Высоты источника s и стока t никогда не изменяются, поскольку эти вершины по определению не переполняются. Таким образом, всегда $h[s] = |V|$ и $h[t] = 0$, и обе не превышают $2|V| - 1$.

Рассмотрим теперь произвольную вершину $u \in V - \{s, t\}$. Изначально $h[u] = 0 \leqslant 2|V| - 1$. Покажем, что после каждого подъема неравенство $h[u] \leqslant 2|V| - 1$ остается справедливым. При подъеме вершины u она является переполненной и, согласно лемме 26.20, имеется простой путь p из u в s в G_f. Пусть $p = \langle v_0, v_1, \ldots, v_k \rangle$, где $v_0 = u$, а $v_k = s$, и $k \leqslant |V| - 1$, поскольку p — простой путь. Для $i = 0, 1, \ldots, k - 1$ имеем $(v_i, v_{i+1}) \in E_f$, следовательно, $h[v_i] \leqslant h[v_{i+1}] + 1$ согласно лемме 26.17. Расписав неравенства для всех составляющих пути p, получаем $h[u] = h[v_0] \leqslant h[v_k] + k \leqslant h[s] + (|V| - 1) = 2|V| - 1$. ■

Следствие 26.22 (Верхний предел числа подъемов). Пусть $G = (V, E)$ — транспортная сеть с источником s и стоком t. Тогда в процессе выполнения процедуры GENERIC_PUSH_RELABEL в G число подъемов не превышает $2|V| - 1$ для одной вершины, а их общее количество не более $(2|V| - 1)(|V| - 2) < 2|V|^2$.

Доказательство. Во множестве $V - \{s, t\}$ только $|V| - 2$ вершин могут быть подняты. Пусть $u \in V - \{s, t\}$. Операция RELABEL(u) увеличивает высоту $h[u]$. Значение $h[u]$ первоначально равно 0 и, согласно лемме 26.21, возрастает не более чем до $2|V| - 1$. Таким образом, каждая вершина $u \in V - \{s, t\}$ подвергается подъему не более $2|V| - 1$ раз, а общее число выполненных подъемов не превышает $(2|V| - 1)(|V| - 2) < 2|V|^2$. ■

Лемма 26.21 помогает также определить границу количества насыщающих проталкиваний.

Лемма 26.23 (Граница количества насыщающих проталкиваний). В процессе выполнения алгоритма GENERIC_PUSH_RELABEL для любой транспортной сети $G = (V, E)$ число насыщающих проталкиваний меньше, чем $2|V||E|$.

Доказательство. Для любой пары вершин $u, v \in V$ рассмотрим насыщающие проталкивания от u к v и от v к u (в обе стороны), и назовем их насыщающими проталкиваниями между u и v. Если есть хотя бы одно такое проталкивание, то хотя бы одно из ребер (u, v) и (v, u) является ребром E. Теперь предположим, что произошло насыщающее проталкивание из u в v. В этот момент $h[v] = h[u] - 1$. Чтобы позднее могло произойти еще одно проталкивание из u в v, алгоритм сначала должен протолкнуть поток из v в u, что невозможно до тех пор, пока не

будет выполнено условие $h[v] = h[u] + 1$. Поскольку $h[u]$ никогда не уменьшается, для того чтобы выполнялось условие $h[v] = h[u] + 1$, значение $h[v]$ должно увеличиться по меньшей мере на 2. Аналогично, $h[u]$ должно увеличиться между последовательными насыщающими проталкиваниями из v в u как минимум на 2. Высота изначально принимает значение 0 и, согласно лемме 26.21, никогда не превышает $2|V| - 1$, откуда следует, что количество раз, когда высота вершины может увеличиться на 2, меньше $|V|$. Поскольку между двумя насыщающими проталкиваниями между u и v хотя бы одна из высот $h[u]$ и $h[v]$ должна увеличиться на 2, всего имеется меньше $2|V|$ насыщающих проталкиваний между u и v. Умножив это число на число ребер, получим, что общее число насыщающих проталкиваний меньше, чем $2|V||E|$. ■

Следующая лемма устанавливает границу числа ненасыщающих проталкиваний в обобщенном алгоритме проталкивания предпотока.

Лемма 26.24 (Граница количества ненасыщающих проталкиваний). В процессе выполнения алгоритма GENERIC_PUSH_RELABEL для любой транспортной сети $G = (V, E)$ число ненасыщающих проталкиваний меньше $4|V|^2(|V| + |E|)$.

Доказательство. Определим потенциальную функцию $\Phi = \sum_{v:e(v)>0} h[v]$. Изначально $\Phi = 0$ и значение Φ может изменяться после каждого подъема, насыщающего и ненасыщающего проталкивания. Мы найдем предел величины, на которую насыщающие проталкивания и подъемы могут увеличивать Φ. Затем покажем, что каждое ненасыщающее проталкивание должно уменьшать Φ как минимум на 1, и используем эти оценки для определения верхней границы числа ненасыщающих проталкиваний.

Рассмотрим два пути увеличения Φ. Во-первых, подъем вершины u увеличивает Φ менее чем на $2|V|$, поскольку множество, для которого вычисляется сумма, остается прежним, а подъем не может увеличить высоту вершины u больше, чем ее максимально возможная высота, которая составляет не более $2|V| - 1$ согласно лемме 26.21. Во-вторых, насыщающее проталкивание из вершины u в вершину v увеличивает Φ менее чем на $2|V|$, поскольку никаких изменений высот при этом не происходит, и только вершина v, высота которой не более $2|V| - 1$, может стать переполненной.

Теперь покажем, что ненасыщающее проталкивание из u в v уменьшает Φ не менее чем на 1. Почему? Перед ненасыщающим проталкиванием вершина u была переполненной, а v могла быть переполненной или непереполненной. Согласно лемме 26.14, после этого проталкивания u больше не является переполненной. Кроме того, после данного проталкивания v должна быть переполненной, если только она не является источником. Следовательно, потенциальная функция Φ уменьшилась ровно на $h[u]$, а увеличилась на 0 или на $h[v]$. Поскольку $h[u] - h[v] = 1$, в итоге потенциальная функция уменьшается как минимум на 1.

Итак, в ходе выполнения алгоритма увеличение Φ происходит благодаря подъемам и насыщающим проталкиваниям; согласно следствию 26.22 и лемме 26.23, это увеличение ограничено, и составляет менее $(2|V|)\left(2|V|^2\right) + (2|V|)(2|V||E|) = 4|V|^2(|V|+|E|)$. Поскольку $\Phi \geqslant 0$, суммарная величина уменьшения и, следовательно, общее число ненасыщающих проталкиваний меньше, чем $4|V|^2(|V|+|E|)$. ■

Определив границу числа подъемов, насыщающего проталкивания и ненасыщающего проталкивания, мы заложили основу дальнейшего анализа процедуры GENERIC_PUSH_RELABEL, а также любых других алгоритмов, основанных на методе проталкивания предпотока.

Теорема 26.25. При выполнении процедуры GENERIC_PUSH_RELABEL для любой транспортной сети $G = (V, E)$ число основных операций составляет $O\left(V^2E\right)$.

Доказательство. Непосредственно вытекает из уже доказанных следствия 26.22 и лемм 26.23 и 26.24. ■

Таким образом, алгоритм завершается после $O\left(V^2E\right)$ операций. Итак, осталось предложить эффективные методы реализации каждой операции и выбора подходящей выполняемой операции.

Следствие 26.26. Существует реализация обобщенного алгоритма проталкивания предпотока, которая для любой сети $G = (V, E)$ выполняется за время $O\left(V^2E\right)$.

Доказательство. В упражнении 26.4-1 предлагается показать, как реализовать обобщенный алгоритм, в котором на каждый подъем затрачивается время $O(V)$, а на каждое проталкивание — $O(1)$. Там же предлагается разработать структуру данных, которая позволит выбрать применимую операцию за время $O(1)$. Тем самым следствие будет доказано. ■

Упражнения

26.4-1. Покажите, как реализовать обобщенный алгоритм проталкивания предпотока, в котором на каждый подъем затрачивается время $O(V)$, на каждое проталкивание — $O(1)$, и то же время $O(1)$ — на выбор применимой операции; суммарное время выполнения при этом составляет $O\left(V^2E\right)$.

26.4-2. Докажите, что время, затрачиваемое в целом на выполнение всех $O\left(V^2\right)$ подъемов в обобщенном алгоритме проталкивания предпотока, составляет только $O(V\,E)$.

26.4-3. Предположим, что с помощью алгоритма проталкивания предпотока найден максимальный поток для транспортной сети $G = (V, E)$. Разработайте быстрый алгоритм поиска минимального разреза в G.

26.4-4. Разработайте эффективный алгоритм проталкивания предпотока для поиска максимального паросочетания в двудольном графе. Проанализируйте время его работы.

26.4-5. Предположим, что все пропускные способности ребер транспортной сети $G = (V, E)$ принадлежат множеству $\{1, 2, \ldots, k\}$. Проанализируйте время выполнения обобщенного алгоритма проталкивания предпотока, выразив его через $|V|$, $|E|$ и k. (*Указание*: сколько ненасыщающих проталкиваний можно применить к каждому ребру, прежде чем оно станет насыщенным?)

26.4-6. Покажите, что строку 7 процедуры INITIALIZE_PREFLOW можно заменить строкой

7 $h[s] \leftarrow |V[G]| - 2$

и это не повлияет на корректность или асимптотическую производительность универсального алгоритма проталкивания предпотока.

26.4-7. Пусть $\delta_f(u, v)$ — расстояние (количество ребер) от u до v в остаточной сети G_f. Покажите, что в процессе работы процедуры GENERIC_PUSH_RELABEL выполняются следующие свойства: если $h[u] < |V|$, то $h[u] \leqslant \leqslant \delta_f(u, t)$, а если $h[u] \geqslant |V|$, то $h[u] - |V| \leqslant \delta_f(u, s)$.

⋆ 26.4-8. Как и в предыдущем упражнении, пусть $\delta_f(u, v)$ — расстояние от u до v в остаточной сети G_f. Покажите, как можно модифицировать обобщенный алгоритм проталкивания предпотока, чтобы в процессе работы процедуры выполнялись следующие свойства — если $h[u] < |V|$, то $h[u] = \delta_f(u, t)$, а если $h[u] \geqslant |V|$, то $h[u] - |V| = \delta_f(u, s)$. Суммарное время, затраченное на обеспечение выполнения данного свойства, должно составлять $O(V E)$.

26.4-9. Покажите, что количество ненасыщающих проталкиваний, выполняемых процедурой GENERIC_PUSH_RELABEL в транспортной сети $G = (V, E)$, не превышает $4|V|^2|E|$, если $|V| \geqslant 4$.

⋆ 26.5 Алгоритм “поднять-в-начало”

Метод проталкивания предпотока позволяет нам применять основные операции в произвольном порядке. Однако после тщательного выбора порядка их выполнения и при эффективном управлении структурой сетевых данных, можно

решить задачу поиска максимального потока быстрее, чем за предельное время $O\left(V^2E\right)$, определенное следствием 26.26. Далее мы рассмотрим алгоритм "поднять-в-начало" (relabel-to-front), основанный на методе проталкивания предпотока, время выполнения которого составляет $O\left(V^3\right)$, что асимптотически не хуже, чем $O\left(V^2E\right)$, а для плотных сетей — лучше.

Алгоритм "поднять-в-начало" поддерживает список вершин сети. Алгоритм сканирует список с самого начала, выбирает некоторую переполненную вершину u, а затем "разгружает" ее, т.е. выполняет операции проталкивания и подъема до тех пор, пока избыток в u не станет равен 0. Если выполнялось поднятие вершины, то она переносится в начало списка (отсюда и название алгоритма: "поднять-в-начало"), и алгоритм начинает очередное сканирование списка.

Для исследования корректности и временны́х характеристик данного алгоритма используется понятие "допустимых" ребер: это ребра остаточной сети, через которые можно протолкнуть поток. Доказав некоторые свойства сети, состоящей из допустимых ребер, мы рассмотрим операцию разгрузки а затем представим и проанализируем сам алгоритм "поднять-в-начало".

Допустимые ребра и сети

Пусть $G = (V, E)$ — некоторая транспортная сеть с источником s и стоком t, f — предпоток в G, а h — функция высоты. Ребро (u, v) называется ***допустимым ребром*** (admissible edge), если $c_f(u, v) > 0$ и $h(u) = h(v) + 1$. В противном случае ребро (u, v) называется ***недопустимым*** (inadmissible). ***Допустимой сетью*** (admissible network) является сеть $G_{f,h} = (V, E_{f,h})$, где $E_{f,h}$ — множество допустимых ребер.

Допустимая сеть состоит из тех ребер, через которые можно протолкнуть поток. Следующая лемма показывает, что такая сеть является ориентированным ациклическим графом.

Лемма 26.27 (Допустимая сеть является ациклической). Если $G = (V, E)$ — некоторая транспортная сеть с источником s и стоком t, f — предпоток в G, а h — функция высоты, тогда допустимая сеть $G_{f,h} = (V,\ E_{f,h})$ является ациклической.

Доказательство. Доказательство проводится методом от противного. Предположим, что $G_{f,h}$ содержит некоторый циклический путь $p = \langle v_0, v_1, \ldots, v_k\rangle$, где $v_0 = v_k$ и $k > 0$. Поскольку каждое ребро пути p является допустимым, справедливо равенство $h(v_{i-1}) = h(v_i) + 1$ для $i = 1, 2, \ldots, k$. Просуммировав эти равенства вдоль циклического пути, получаем:

$$\sum_{i=1}^{k} h(v_{i-1}) = \sum_{i=1}^{k} (h(v_i) + 1) = \sum_{i=1}^{k} h(v_i) + k.$$

Поскольку каждая вершина циклического пути p встречается при суммировании по одному разу, приходим к выводу, что $k = 0$, что противоречит первоначальному предположению. ■

Следующая лемма показывает, как операции проталкивания и подъема изменяют допустимую сеть.

Лемма 26.28. Пусть $G = (V, E)$ — транспортная сеть с источником s и стоком t, f — предпоток в G, и предположим, что атрибут h — функция высоты. Если вершина u переполнена и ребро (u, v) является допустимым, то применяется операция PUSH(u, v). Данная операция не создает новые допустимые ребра, однако она может привести к тому, что ребро (u, v) станет недопустимым.

Доказательство. По определению допустимого ребра, из u в v можно протолкнуть поток. Поскольку вершина u переполнена, применяется операция PUSH(u, v). В результате проталкивания потока из u в v может быть создано только одно новое остаточное ребро (v, u). Поскольку $h[v] = h[u] - 1$, ребро (v, u) не может стать допустимым. Если примененная операция является насыщающим проталкиванием, то после ее выполнения $c_f(u, v) = 0$ и ребро (u, v) становится недопустимым. ■

Лемма 26.29. Пусть $G = (V, E)$ — транспортная сеть с источником s и стоком t, f — предпоток в G, и предположим, что атрибут h — функция высоты. Если вершина u переполнена и не имеется допустимых ребер, выходящих из u, то применяется операция RELABEL(u). После подъема появляется по крайней мере одно допустимое ребро, выходящее из u, но нет допустимых ребер, входящих в u.

Доказательство. Если вершина u переполнена, то, согласно лемме 26.15, к ней применяется или операция проталкивания, или операция подъема. Если не существует допустимых ребер, выходящих из u, то протолкнуть поток из u невозможно, следовательно, применяется операция RELABEL(u). После данного подъема $h[u] = 1 + \min\{h[v] : (u, v) \in E_f\}$. Таким образом, если v — вершина, в которой реализуется минимум указанного множества, ребро (u, v) становится допустимым. Следовательно, после подъема существует по крайней мере одно допустимое ребро, выходящее из u.

Чтобы показать, что после подъема не существует входящих в u допустимых ребер, предположим, что существует некоторая вершина v, такая что ребро (u, v) допустимо. Тогда после подъема $h[v] = h[u] + 1$, так что непосредственно перед подъемом $h[v] > h[u] + 1$. Но, согласно лемме 26.13, не существует остаточных ребер между вершинами, высоты которых отличаются более чем на 1. Кроме того, подъем вершины не меняет остаточную сеть. Таким образом, ребро (v, u) не принадлежит остаточной сети, следовательно, оно не может находиться в допустимой сети. ■

Списки соседей

Ребра в алгоритме "поднять-в-начало" объединены в так называемые "списки соседей". В заданной транспортной сети $G = (V, E)$ ***списком соседей*** (neighbor list) $N[u]$ некоторой вершины $u \in V$ является односвязный список вершин, смежных с u в G. Таким образом, вершина v оказывается в списке $N[u]$ только в том случае, если $(u, v) \in E$ или $(v, u) \in E$. Список соседей $N[u]$ содержит только такие вершины v, для которых может существовать остаточное ребро (u, v). На первую вершину в списке $N[u]$ указывает указатель $head[N[u]]$. Для вершины, следующей за v в списке соседей, поддерживается указатель $next\text{-}neighbor[v]$; этот указатель имеет значение NIL, если v — последняя вершина в списке соседей.

Алгоритм "поднять-в-начало" циклически обрабатывает каждый список соседей в произвольном порядке, который фиксируется в процессе выполнения алгоритма. Для каждой вершины u поле $current[u]$ указывает на текущую вершину списка $N[u]$. Первоначально $current[u]$ устанавливается равным $head[N[u]]$.

Разгрузка переполненной вершины

Переполненная вершина u ***разгружается*** (discharged) путем проталкивания всего ее избыточного потока через допустимые ребра в смежные вершины, при этом, если необходимо, производится подъем вершины u, чтобы ребра, выходящие из вершины u, стали допустимыми. Псевдокод разгрузки выглядит следующим образом:

```
DISCHARGE(u)
1  while e[u] > 0
2        do v ← current[u]
3           if v = NIL
4             then RELABEL(u)
5                  current[u] ← head[N[u]]
6           elseif c_f(u, v) > 0 и h[u] = h[v] + 1
7             then PUSH(u, v)
8           else current[u] ← next-neighbor[v]
```

На рис. 26.9 показаны несколько итераций цикла **while** (строки 1–8), тело цикла выполняется до тех пор, пока вершина u имеет положительный избыток. Каждая итерация выполняет одно из трех действий в зависимости от текущей вершины v из списка соседей $N[u]$. На рисунке показаны только смежные с y вершины и ребра, входящие в y или выходящие из нее. Число внутри каждой вершины — ее избыток к моменту начала первой итерации, показанной в данном фрагменте; каждая вершина показана на своей высоте. Справа приводится список соседей $N[y]$ в начале каждой итерации (номер итерации указан вверху).

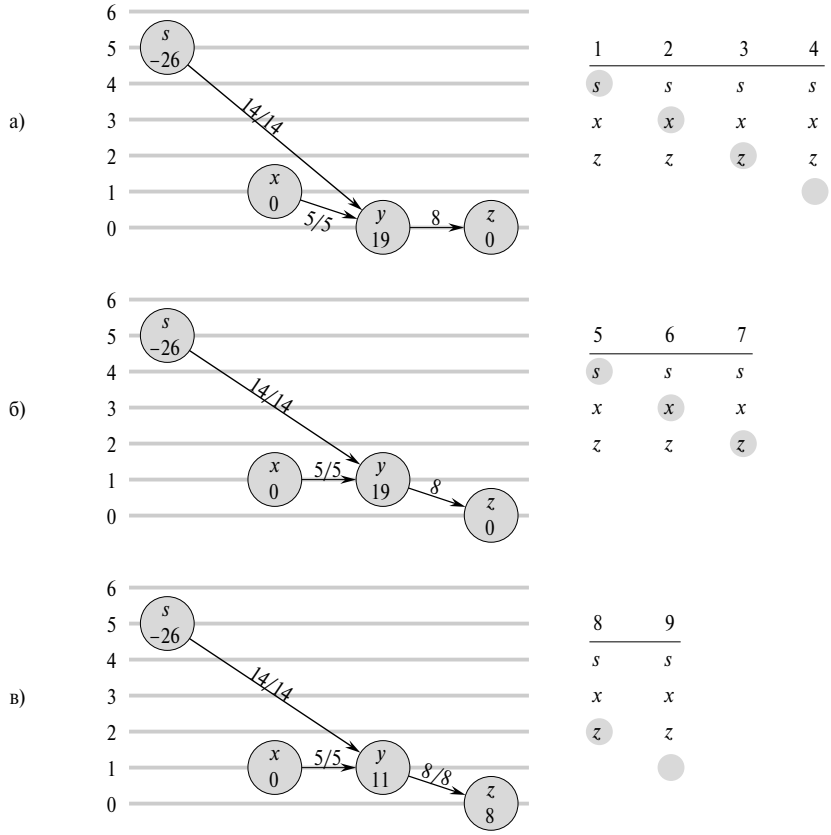
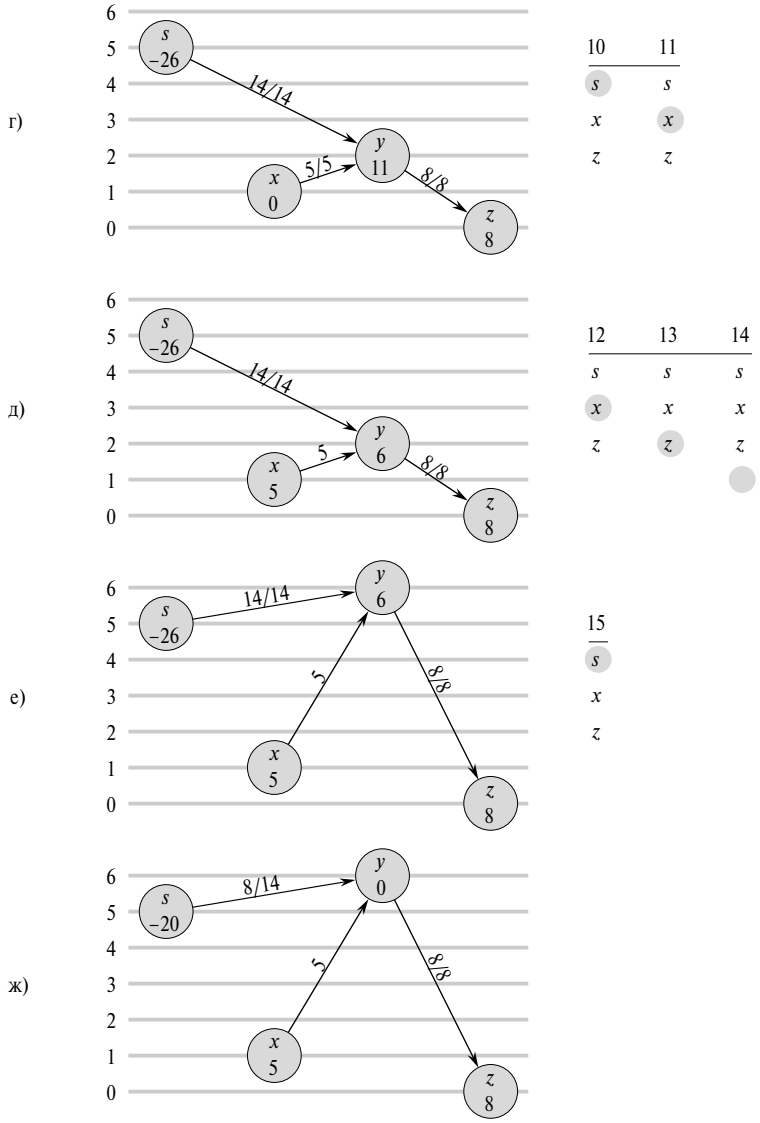


Рис. 26.9. Разгрузка вершины y. Чтобы протолкнуть весь избыток потока из вершины y, требуется 15 итераций цикла **while** процедуры DISCHARGE

1. Если v равно NIL, значит, мы дошли до конца списка $N[u]$. Выполняется подъем вершины u (строка 4), а затем (строка 5) текущей соседней вершиной u делается первая вершина из списка $N[u]$. (В приведенной ниже лемме 26.30 утверждается, что в данной ситуации подъем применим.)
2. Если v не равно NIL, и ребро (u, v) является допустимым (что определяется с помощью проверки в строке 6), тогда (строка 7) производится проталкивание части (или всего) избытка из u в вершину v.
3. Если v не равно NIL, но ребро (u, v) является недопустимым, тогда (строка 8) указатель $current[u]$ в списке $N[u]$ перемещается на одну позицию вперед.

Заметим, что если процедура DISCHARGE вызывается для некоторой переполненной вершины u, последним действием, выполняемым данной процедурой, должно быть проталкивание из u. Почему? Процедура завершается только то-



гда, когда избыток $e[u]$ становится равным нулю, и ни подъем, ни перемещение указателя $current[u]$ не влияют на значение $e[u]$.

Теперь необходимо убедиться, что когда процедура DISCHARGE вызывает процедуры PUSH или RELABEL, эти операции применимы. Следующая лемма доказывает данный факт.

Лемма 26.30. Когда процедура DISCHARGE вызывает в строке 7 процедуру PUSH(u, v), то для пары вершин (u, v) применима операция проталкивания. Когда

процедура DISCHARGE вызывает в строке 4 процедуру RELABEL(u), к вершине u применим подъем.

Доказательство. Проверки в строках 1 и 6 гарантируют, что операция проталкивания вызывается только тогда, когда она применима, таким образом, первое утверждение леммы доказано.

Чтобы доказать второе утверждение, исходя из проверки в строке 1 и леммы 26.29, необходимо только показать, что все ребра, выходящие из u, являются недопустимыми. При повторных вызовах DISCHARGE(u) указатель $current\,[u]$ смещается вниз по списку $N\,[u]$. Каждый "проход" начинается с головы списка $N\,[u]$ и оканчивается при $current\,[u] =$ NIL; в этот момент производится подъем u и начинается новый проход. Во время прохода, чтобы передвинуть указатель $current\,[u]$ с некоторой вершины $v \in N\,[u]$ вперед, ребро (u, v) должно быть признано недопустимым проводимой в строке 6 проверкой. Поэтому к окончанию очередного прохода все ребра, выходящие из u, в некоторый момент этого прохода были признаны недопустимыми. Ключевым является тот факт, что к концу прохода все ребра, покидающие u, остаются недопустимыми. Почему? Согласно лемме 26.28, операции проталкивания не могут приводить к созданию допустимых ребер, в том числе и выходящих из вершины u. Поэтому любое допустимое ребро должно быть создано операцией подъема. Но вершина u не подвергается подъему в процессе прохода, а любая другая вершина v, подвергшаяся подъему в процессе данного прохода, не имеет после подъема допустимых входящих ребер согласно лемме 26.29. Таким образом, в конце прохода все ребра, выходящие из u, остаются недопустимыми, и лемма доказана. ■

Алгоритм "поднять-в-начало"

В алгоритме "поднять-в-начало" поддерживается связанный список L, состоящий из всех вершин $V - \{s, t\}$. Ключевым свойством данного списка является то, что вершины в нем топологически отсортированы в соответствии с допустимой сетью, как будет показано при рассмотрении инварианта цикла ниже. (Напомним, что, согласно лемме 26.27, допустимая сеть является ориентированным ациклическим графом.)

В приведенном ниже псевдокоде алгоритма "поднять-в-начало" предполагается, что для каждой вершины u уже создан список соседей $N\,[u]$. Кроме того, предполагается, что $next\,[u]$ указывает на вершину, следующую за u в списке L, и что, как обычно, если u — последняя вершина данного списка, то $next\,[u] =$ NIL.

```
RELABEL_TO_FRONT(G, s, t)
 1  INITIALIZE_PREFLOW(G, s)
 2  L ← V[G] − {s, t} в произвольном порядке
 3  for (для) каждой вершины u ∈ V[G] − {s, t}
```

```
 4      do current[u] ← head[N[u]]
 5  u ← head[L]
 6  while u ≠ NIL
 7      do old-height ← h[u]
 8         DISCHARGE(u)
 9         if h[u] > old-height
10           then передвинуть u в начало списка L
11         u ← next[u]
```

Алгоритм "поднять-в-начало" работает следующим образом. Строка 1 инициализирует предпоток и высоты теми же значениями, что и обобщенный алгоритм проталкивания предпотока. Строка 2 инициализирует список L, который содержит все потенциально переполненные вершины в произвольном порядке. Строки 3–4 инициализируют указатель $current$ каждой вершины u, чтобы он указывал на первую вершину в списке соседей u.

Как показано на рис. 26.10, цикл **while** (строки 6–11) проходит по списку l, разгружая вершины. Рассмотрение начинается с первой вершины списка L (строка 5). На каждой итерации цикла выполняется разгрузка вершины u (строка 8). Если процедура DISCHARGE изменила высоту вершины u, строка 10 перемещает эту вершину в начало списка L. Чтобы определить, подверглась ли вершина u подъему, перед разгрузкой ее высота сохраняется в переменной *old_height* (строка 7), а затем это значение сравнивается со значением высоты после выполнения процедуры разгрузки (строка 9). Строка 11 обеспечивает выполнение очередной итерации цикла **while** для вершины, следующей за u в списке L. Если u была передвинута в начало списка, рассматриваемая на следующей итерации вершина — это вершина, следующая за u в ее новой позиции в списке.

Чтобы показать, что процедура RELABEL_TO_FRONT вычисляет максимальный поток, покажем, что она является реализацией универсального алгоритма проталкивания предпотока. Во-первых, заметим, что она выполняет операции проталкивания и подъема только тогда, когда они применимы, что гарантируется леммой 26.30. Остается показать, что когда процедура RELABEL_TO_FRONT завершается, не применима ни одна основная операция. Дальнейшее доказательство корректности построено на следующем инварианте цикла:

> При каждом выполнении проверки в строке 6 процедуры RELABEL_TO_FRONT список L является топологическим упорядочением вершин допустимой сети $G_{f,h} = (V, E_{f,h})$, и ни одна вершина, стоящая в списке перед u, не имеет избыточного потока.

Инициализация. Непосредственно после запуска процедуры INITIALIZE_PREFLOW $h[s] = |V|$ и $h[v] = 0$ для всех $v \in V - \{s\}$. Поскольку $|V| \geqslant 2$ (так как V содержит как минимум источник s и сток t), ни одно ребро не

является допустимым. Следовательно, $E_{f,h} = \emptyset$, и любой порядок множества $V - \{s, t\}$ является топологическим упорядочением $G_{f,h}$.

Поскольку изначально вершина u является заголовком списка L, перед ней нет вершин, следовательно, перед ней нет вершин с избытком потока.

Сохранение. Чтобы убедиться, что данное топологическое упорядочение сохраняется при проведении итераций цикла **while**, прежде всего заметим, что к изменению допустимой сети приводят только операции проталкивания и подъема. Согласно лемме 26.28, операции проталкивания не приводят к появлению новых допустимых ребер. Поэтому допустимые ребра могут создаваться только подъемами. После того как вершина u подверглась подъему, согласно лемме 26.29, больше не существует допустимых ребер, входящих в u, но могут быть допустимые ребра, выходящие из нее. Таким образом, перемещая u в начало списка L, алгоритм гарантирует, что все допустимые ребра, выходящие из u, удовлетворяют условию топологического упорядочения.

Чтобы убедиться, что ни одна вершина, предшествующая u в списке L, не имеет избытка потока, обозначим вершину, которая будет текущей вершиной u на следующей итерации, как u'. Среди вершин, которые будут предшествовать u' на следующей итерации, находится текущая вершина u (согласно строке 11) и либо больше таких вершин нет (если u подвергалась подъему), либо там находятся те же вершины, что и ранее (если высота u не изменялась). Поскольку u подверглась разгрузке, она не содержит избытка потока. Следовательно, если u подвергалась подъему в процессе разгрузки, то ни одна вершина, предшествующая u', не содержит избытка потока. Если же высота u в процессе разгрузки не менялась, ни одна вершина, стоящая в списке L перед ней, не получила избыток потока при этой разгрузке, поскольку L остается топологически упорядоченным все время в процессе разгрузки (как уже отмечалось, допустимые ребра создаются только подъемами, а не операциями проталкивания), поэтому каждая операция проталкивания заставляет избыток потока двигаться только к вершинам, расположенным в списке дальше (или же к s или t). В этом случае ни одна вершина, предшествующая u', также не имеет избытка потока.

Завершение. Когда цикл завершается, u оказывается за последним элементом списка L, поэтому инвариант цикла гарантирует, что избыток всех вершин равен 0. Следовательно, ни одна основная операция неприменима.

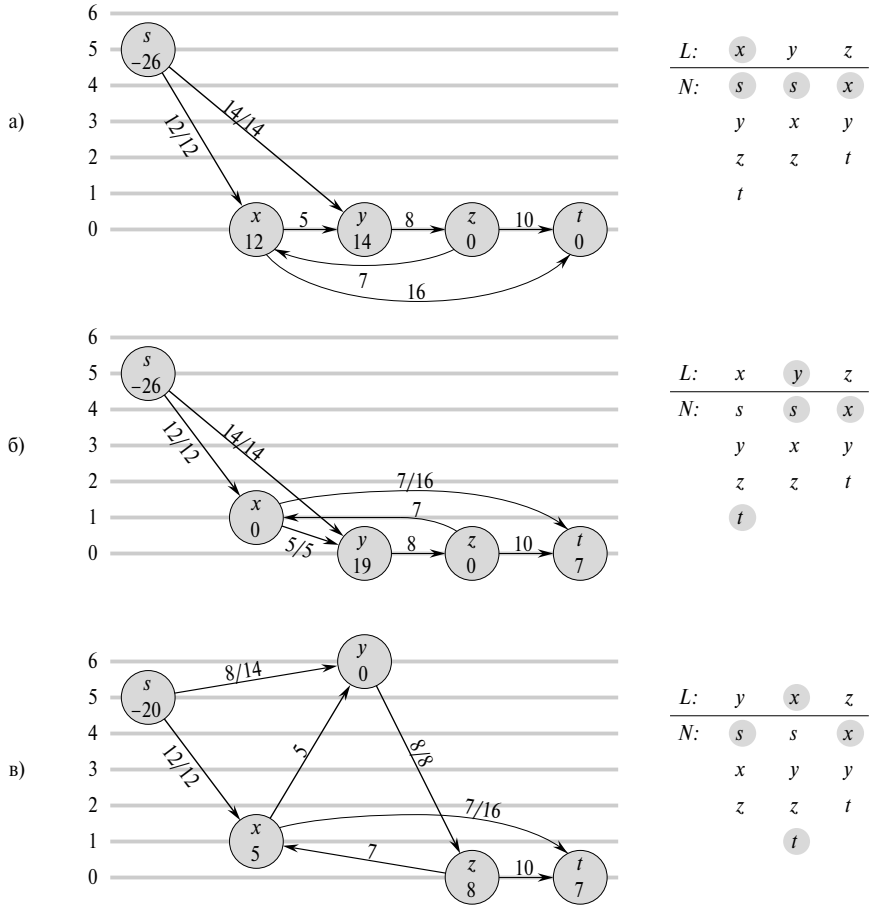
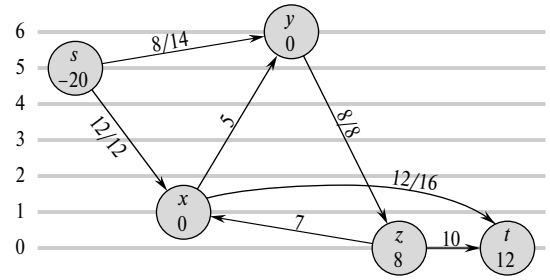


Рис. 26.10. Работа процедуры RELABEL_TO_FRONT. Справа показан список L, где под каждой вершиной приведен список ее соседей, в котором выделена текущая вершина

Анализ

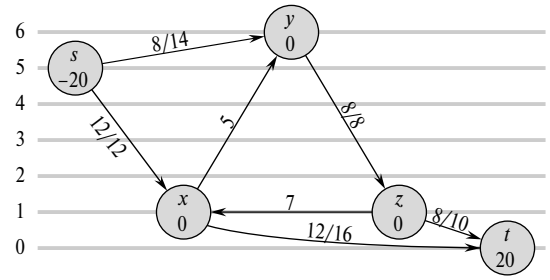
Покажем теперь, что процедура RELABEL_TO_FRONT выполняется за время $O\left(V^3\right)$ для любой сети $G = (V, E)$. Поскольку данный алгоритм является реализацией универсального алгоритма проталкивания предпотока, воспользуемся следствием 26.22, которое устанавливает границу $O\left(V\right)$ для числа подъемов, применяемых к одной вершине, и $O\left(V^2\right)$ для общего числа подъемов. Кроме того, в упражнении 26.4-2 устанавливается граница $O\left(V\,E\right)$ для суммарного времени, затраченного на выполнение подъемов, а лемма 26.23 устанавливает границу $O\left(V\,E\right)$ для суммарного числа операций насыщающих проталкиваний.

г)



L:	*y*	*x*	*z*
N:	*s*	*s*	*x*
	x	*y*	*y*
	z	*z*	*t*
		t	

д)



L:	*z*	*y*	*x*
N:	*x*	*s*	*s*
	y	*x*	*y*
	t	*z*	*z*
			t

Теорема 26.31. Время выполнения процедуры RELABEL_TO_FRONT для любой сети $G = (V, E)$ составляет $O\left(V^3\right)$.

Доказательство. Будем считать "фазой" алгоритма "поднять-в-начало" время между двумя последовательными операциями подъема. Поскольку всего выполняется $O\left(V^2\right)$ подъемов, в алгоритме насчитывается $O\left(V^2\right)$ фаз. Каждая фаза сдержит не более $|V|$ обращений к процедуре DISCHARGE, что можно показать следующим образом. Если процедура DISCHARGE не выполняет подъем, то следующий ее вызов происходит ниже по списку L, а длина L меньше $|V|$. Если процедура DISCHARGE выполняет подъем, то следующий ее вызов происходит уже в другой фазе алгоритма. Поскольку каждая фаза содержит не более $|V|$ обращений к процедуре DISCHARGE и в алгоритме насчитывается $O\left(V^2\right)$ фаз, число вызовов данной процедуры строкой 8 процедуры RELABEL_TO_FRONT составляет $O\left(V^3\right)$.Таким образом, цикл **while** процедуры RELABEL_TO_FRONT выполняет работу (не учитывая работу, выполняемую внутри процедуры DISCHARGE), не превышающую $O\left(V^3\right)$.

Теперь необходимо оценить работу, выполняемую внутри процедуры DISCHARGE в ходе выполнения данного алгоритма. Каждое выполнение цикла **while** в процедуре DISCHARGE заключается в выполнении одного из трех действий. Проанализируем объем работы при выполнении каждого из них.

Начнем с подъемов (строки 4–5). В упражнении 26.4-2 время для выполнения всех $O\left(V^2\right)$ подъемов ограничивается пределом $O\left(V\,E\right)$.

Теперь предположим, что действие заключается в обновлении указателя $current\,[u]$ в строке 8. Это действие выполняется $O\,(\mathrm{degree}(u))$ в том случае, когда вершина u подвергается подъему, что для всех вершин составляет $O(V \cdot \mathrm{degree}(u))$ раз. Следовательно, для всех вершин общий объем работы по перемещению указателей в списке соседей составляет $O\,(V\,E)$ согласно лемме о рукопожатиях (упражнение Б.4-1).

Третий тип действий, выполняемых процедурой DISCHARGE, — операция проталкивания (строка 7). Как уже отмечалось, суммарное число насыщающих операций проталкивания составляет $O\,(V\,E)$. Если выполняется ненасыщающее проталкивание, процедура DISCHARGE немедленно выполняет возврат, поскольку такое проталкивание уменьшает избыток до 0. Поэтому при каждом обращении к процедуре DISCHARGE может выполняться не более одного ненасыщающего проталкивания. Процедура DISCHARGE вызывается $O\left(V^3\right)$ раз, следовательно, общее время, затраченное на ненасыщающие проталкивания, составляет $O\left(V^3\right)$.

Таким образом, время выполнения процедуры RELABEL_TO_FRONT составляет $O\left(V^3 + VE\right)$, что эквивалентно $O\left(V^3\right)$. ■

Упражнения

26.5-1. Проиллюстрируйте (используя в качестве образца рис. 26.10) выполнение процедуры RELABEL_TO_FRONT для сети, представленной на рис. 26.1*a*. Предполагается, что начальный порядок следования вершин в списке $L = \langle v_1, v_2, v_3, v_4 \rangle$, а списки соседей имеют следующий вид:

$$N\,[v_1] = \langle s, v_2, v_3 \rangle\,,$$
$$N\,[v_2] = \langle s, v_1, v_3, v_4 \rangle\,,$$
$$N\,[v_3] = \langle v_1, v_2, v_4, t \rangle\,,$$
$$N\,[v_4] = \langle v_2, v_3, t \rangle\,.$$

⋆ 26.5-2. Мы хотим реализовать алгоритм проталкивания предпотока, в котором поддерживается порядок обслуживания переполненных вершин “первым вошел, первым вышел”. Данный алгоритм разгружает первую вершину в очереди и удаляет ее оттуда, а все вершины, которые перед этой разгрузкой не были переполнены, но после нее стали таковыми, помещаются в конец очереди. Когда очередь становится пустой, алгоритм завершается. Покажите, что можно построить реализацию данного алгоритма, которая вычисляет максимальный поток за время $O\left(V^3\right)$.

26.5-3. Покажите, что обобщенный алгоритм будет работать, даже если процедура RELABEL при обновлении $h\,[u]$ просто использует выражение $h\,[u] \leftarrow$ $\leftarrow h\,[u] + 1$. Как это повлияет на оценку времени выполнения процедуры RELABEL_TO_FRONT?

⋆ 26.5-4. Покажите, что если разгрузке всегда подвергается наивысшая переполненная вершина, метод проталкивания предпотока можно реализовать так, чтобы он выполнялся за время $O\left(V^3\right)$.

26.5-5. Предположим, что в некоторой точке в процессе выполнения алгоритма проталкивания предпотока, существует некоторое целое число $0 < k \leqslant \leqslant |V| - 1$, для которого нет ни одной вершины с данной высотой, т.е. такой, что $h[v] = k$. Покажите, что все вершины с $h[v] > k$ находятся в минимальном разрезе на стороне источника. Если такое значение k существует, ***эвристика промежутка*** (gap heuristic) обновляет каждую вершину $v \in V - s$, для которой $h[v] > k$, устанавливая $h[v] \leftarrow \leftarrow \max(h[v], |V| + 1)$. Покажите, что полученный таким образом атрибут h является функцией высоты. (Эвристика промежутка позволяет получить эффективные реализации метода проталкивания предпотока на практике.)

Задачи

26-1. **Задача о выходе**

Решетка (grid) размером $n \times n$ — это неориентированный граф, состоящий из n строк и n столбцов вершин, как показано на рис. 26.11. Обозначим вершину, находящуюся в i-й строке и j-м столбце как (i, j). Каждая вершина решетки имеет ровно по четыре соседа, за исключением граничных вершин, представляющих собой точки (i, j), для которых $i = 1$, $i = n$, $j = 1$ или $j = n$.

Пусть в решетке задано $m \leqslant n^2$ стартовых точек (x_1, y_1), (x_2, y_2), $\ldots$, (x_m, y_m). ***Задача о выходе*** (escape problem) заключается в том, чтобы определить, существует ли m путей, не имеющих общих вершин, из стартовых точек к любым m различным точкам границы. Например, решетка на рис. 26.11*а* имеет выход, а решетка на рис. 26.11*б* — не имеет.

а) Рассмотрим транспортную сеть, в которой вершины, равно как и ребра, имеют пропускные способности, т.е. суммарный положительный поток, входящий в каждую заданную вершину, должен удовлетворять ограничению пропускной способности. Покажите, что задача определения максимального потока в такой сети, где вершины и ребра имеют пропускные способности, может быть сведена к стандартной задаче о максимальном потоке для транспортной сети сопоставимого размера.

б) разработайте эффективный алгоритм решения задачи о выходе и проанализируйте время его выполнения.

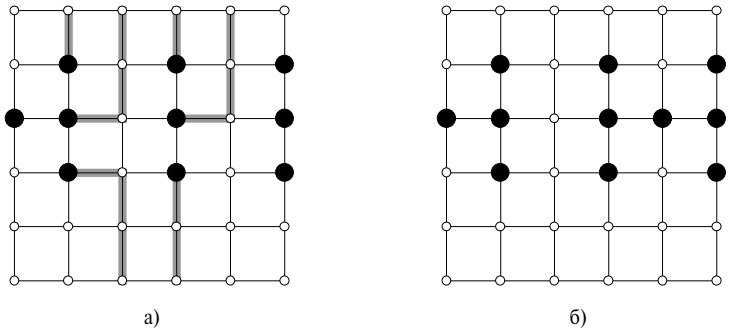


Рис. 26.11. Решетки для задачи о выходе. Стартовые точки отмечены черным цветом, остальные вершины решетки белые. а) Решетка с выходом, пути указаны цветом. б) Решетка без выхода

26-2. **Задача о минимальном покрытии путями**

Покрытие путями (path cover) ориентированного графа $G = (V, E)$ — это множество P не имеющих общих вершин путей, таких что каждая вершина из множества V принадлежит ровно одному пути из P. Пути могут начинаться и заканчиваться где угодно, а также иметь произвольную длину, включая 0. ***Минимальным покрытием путями*** (minimum path cover) графа G называется покрытие, содержащее наименьшее возможное количество путей.

а) Предложите эффективный алгоритм поиска минимального покрытия путями ориентированного ациклического графа $G = (V, E)$. (*Указание*: предположив, что $V = \{1, 2, \ldots, n\}$, постройте граф $G' = (V', E')$, где

$$V' = \{x_0, x_1, \ldots, x_n\} \cup \{y_0, y_1, \ldots, y_n\},$$
$$E' = \{(x_0, x_i) : i \in V\} \cup \{(y_1, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\},$$

и примените алгоритм поиска максимального потока.)

б) Будет ли ваш алгоритм работать для ориентированного графа, содержащего циклы? Объясните ваш ответ.

26-3. **Эксперименты на кораблях многоразового использования**

Профессор консультирует НАСА при планировании серии полетов кораблей многоразового использования. Он должен решить, какие коммерческие эксперименты следует провести и какие инструменты будут на борту во время каждого полета. Для каждого полета рассматривается множество экспериментов $E = \{E_1, E_2, \ldots, E_m\}$, и коммерческий

спонсор эксперимента E_j согласен заплатить НАСА p_j долларов за его результат. В экспериментах используется множество инструментов $I = \{I_1, I_2, \ldots, I_n\}$; каждый эксперимент E_j требует наличия всех инструментов из некого подмножества $R_j \subseteq I$. Затраты перевозки в космическом корабле инструмента I_k составляют c_k долларов. Задача профессора — найти эффективный алгоритм, позволяющий определить, какие эксперименты проводить и какие инструменты размещать на борту корабля при каждом полете, чтобы максимизировать чистый доход, который вычисляется как суммарный доход от выполненных экспериментов минус суммарные затраты на перевозку всех используемых инструментов.

Рассмотрим следующую сеть G. Данная сеть содержит источник s, вершины $I_1, I_2, \ldots, I_n$, вершины $E_1, E_2, \ldots, E_m$ и сток t. Для каждого $k = 1, 2, \ldots, n$ имеется ребро (s, I_k) с пропускной способностью c_k, а для каждого $j = 1, 2, \ldots, m$ имеется ребро (E_j, t) с пропускной способностью p_j. Если для $k = 1, 2, \ldots, n$ и $j = 1, 2, \ldots, m$ выполняется условие $I_k \in R_j$, то существует ребро (I_k, E_j) неограниченной пропускной способности.

а) Покажите, что если для разреза конечной пропускной способности (S, T) сети G $E_j \in T$, то $I_k \in T$ для всех $I_k \in R_j$.

б) Покажите, как определить максимальный чистый доход на основании пропускной способности минимального разреза G и заданных значений p_j.

в) Предложите эффективный алгоритм, позволяющий определить, какие эксперименты проводить и какие инструменты брать в каждый полет. Проанализируйте время выполнения вашего алгоритма, выразив его через m, n и $r = \sum_{j=1}^{m} |R_j|$.

26-4. **Обновление максимального потока**

Пусть $G = (V, E)$ — транспортная сеть с источником s, стоком t и целочисленными пропускными способностями. Предположим, что известен максимальный поток в G.

а) Предположим, что пропускная способность некоторого одного ребра $(u, v) \in E$ увеличена на 1. Предложите алгоритм обновления максимального потока с временем выполнения $O(V + E)$.

б) Предположим, что пропускная способность некоторого одного ребра $(u, v) \in E$ уменьшена на 1. Предложите алгоритм обновления максимального потока с временем выполнения $O(V + E)$.

26-5. **Масштабирование**

Пусть $G = (V, E)$ — транспортная сеть с источником s, стоком t и целочисленными пропускными способностями $c(u, v)$ всех ребер $(u, v) \in E$. Пусть $C = \max_{(u,v)\in E} c(u, v)$.

а) Докажите, что минимальный разрез G имеет пропускную способность не более $C|E|$.

б) Для заданного числа K покажите, что за время $O(E)$ можно найти увеличивающий путь с пропускной способностью не менее K, если такой путь существует.

Для вычисления максимального потока в G можно использовать следующую модификацию процедуры FORD_FULKERSON_METHOD:

```
MAX_FLOW_BY_SCALING(G, s, t)
1  C ← max_{(u,v)∈E} c(u, v)
2  Инициализация потока f значением 0
3  K ← 2^⌊lg C⌋
4  while K ≥ 1
5        do while (пока) существует некоторый увеличивающий путь p
                              с пропускной способностью не менее K
6              do увеличить f вдоль p
7           K ← K/2
8  return f
```

в) Докажите, что процедура MAX_FLOW_BY_SCALING возвращает максимальный поток.

г) Покажите, что каждый раз, когда выполняется строка 4, пропускная способность минимального разреза остаточного графа G_f составляет не более $2K|E|$.

д) Докажите, что внутренний цикл **while** в строках 5–6 при каждом значении K выполняется $O(E)$ раз.

е) Покажите, что процедуру MAX_FLOW_BY_SCALING можно реализовать так, что она будет выполняться за время $O\left(E^2 \lg C\right)$.

26-6. **Отрицательная пропускная способность**

Предположим, что в транспортной сети ребра могут иметь как положительную, так и отрицательную пропускную способность. В такой сети не обязательно существует допустимый поток.

а) Рассмотрим некоторое ребро (u, v) сети $G = (V, E)$, пропускная способность которого $c(u, v) < 0$. Кратко поясните, что такая отрицательная пропускная способность означает в плане потока между u и v, каков ее "физический смысл".

Пусть $G = (V, E)$ — некоторая транспортная сеть с отрицательными пропускными способностями, и пусть s и t — источник и сток данной сети. Построим обычную транспортную сеть $G' = (V', E')$ с функцией пропускной способности c', источником s' и стоком t', где

$$V' = V \cup \{s', t'\}$$

и

$$\begin{aligned} E' = E &\cup \{(u, v) : (v, u) \in E\} \\ &\cup \{(s', v) : v \in V\} \\ &\cup \{(u, t') : u \in V\} \\ &\cup \{(s, t), (t, s)\}. \end{aligned}$$

Присвоим ребрам пропускные способности следующим образом. Для каждого ребра $(u, v) \in E$ присвоим

$$c'(u, v) = c'(v, u) = (c(u, v) + c(v, u))/2.$$

Для каждой вершины $u \in V$ присвоим

$$c'(s', u) = \max(0, (c(V, u) - c(u, V))/2)$$

и

$$c'(u, t') = \max(0, (c(u, V) - c(V, u))/2).$$

Присвоим также $c'(s, t) = c'(t, s) = \infty$.

б) Докажите, что если в G существует допустимый поток, то все пропускные способности в G' неотрицательны и в ней существует максимальный поток, такой что все входящие в сток t' ребра являются насыщенными.

в) Докажите утверждение, обратное утверждению пункта *б*). Доказательство должно быть конструктивным, т.е. если задан поток в G', насыщающий все ребра, входящие в t', вам нужно показать, как получить допустимый поток в G.

г) Разработайте алгоритм, который находит максимальный допустимый поток в G. Обозначим через $MF(|V|, |E|)$ наихудшее время выполнения стандартного алгоритма поиска максимального потока в графе с $|V|$ вершинами и $|E|$ ребрами. Проанализируйте время выполнения вашего алгоритма вычисления максимального потока в транспортной сети с отрицательными пропускными способностями, выразив его через MF.

26-7. **Алгоритм Хопкрофта-Карпа поиска паросочетания в двудольном графе**

В данной задаче представлен более быстрый алгоритм поиска максимального паросочетания в двудольном графе, предложенный Хопкрофтом (Hopcroft) и Карпом (Karp). Этот алгоритм выполняется за время $O\left(\sqrt{V}E\right)$. Задан неориентированный двудольный граф $G = (V, E)$, где $V = L \cup R$ и у всех ребер ровно одна точка находится в L. Пусть M — паросочетание в G. Мы говорим, что простой путь P в G является ***увеличивающим путем*** (augmenting path) по отношению к M, если он начинается в некоторой свободной вершине множества L, заканчивается в некоторой свободной вершине R, а его ребра попеременно принадлежат M и $E - M$. (Это определение увеличивающего пути связано с определением увеличивающего пути в транспортной сети, но несколько отличается от него.) В данной задаче путь трактуется как последовательность ребер, а не последовательность вершин. Кратчайший увеличивающий путь по отношению к паросочетанию M — это увеличивающий путь с минимальным числом ребер.

Для заданных двух множеств A и B, ***симметрическая разность*** (symmetric difference) $A \oplus B$ определяется как $(A - B) \cup (B - A)$, т.е. это элементы, которые входят ровно в одно из двух множеств.

а) Покажите, что если M — некоторое паросочетание, а P — увеличивающий путь по отношению к M, то симметрическая разность множеств $M \oplus P$ является паросочетанием, и $|M \oplus P| = |M| + 1$. Покажите, что если $P_1, P_2, \ldots, P_k$ — увеличивающие пути по отношению к M, не имеющие общих вершин, то симметрическая разность $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$ является паросочетанием с мощностью $|M| + k$.

Общая структура алгоритма имеет следующий вид:

```
HOPCROFT_KARP(G)
1  M ← ∅
2  repeat
3          Пусть P ← {P_1, P_2, ..., P_k} максимальное множество
                   кратчайших увеличивающих путей по отношению
                   к M, не имеющих общих вершин
4          M ← M ⊕ (P_1 ∪ P_2 ∪ ··· ∪ P_k)
5     until P = ∅
6  return M
```

Далее в этой задаче вам предлагается проанализировать число итераций данного алгоритма (т.е. число итераций цикла **repeat**) и предложить реализацию строки 3.

б) Для двух заданных паросочетаний M и M^* в G покажите, что каждая вершина графа $G = (V, M \oplus M^*)$ имеет степень не больше 2. Сделайте вывод, что G' является объединением непересекающихся простых путей или циклов. Докажите, что ребра каждого такого простого пути или цикла по очереди принадлежат M и M^*. Докажите, что если $|M| \leqslant |M^*|$, то $M \oplus M^*$ содержит как минимум $|M^*| - |M|$ увеличивающих путей по отношению к M, не имеющих общих вершин.

Пусть l — длина кратчайшего увеличивающего пути по отношению к паросочетанию M и пусть $P_1, P_2, \ldots, P_k$ — максимальное множество увеличивающих путей длины l по отношению к M, не имеющих общих вершин. Пусть $M' = M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$ и предположим, что P — кратчайший увеличивающий путь по отношению к M'.

в) Покажите, что если P не имеет общих вершин с $P_1, P_2, \ldots, P_k$, то P содержит больше чем l ребер.

г) Теперь предположим, что P имеет общие вершины с $P_1, P_2, \ldots, P_k$. Пусть A — множество ребер $(M \oplus M') \oplus P$. Покажите, что $A = (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P$ и что $|A| \geqslant (k+1)\, l$. Покажите, что P содержит более l ребер.

д) Докажите, что если кратчайший увеличивающий путь для M содержит l ребер, то размер максимального паросочетания составляет не более $|M| + |V|/(l+1)$.

е) Покажите, что число повторений цикла **repeat** в данном алгоритме не превышает $2\sqrt{V}$. (*Указание*: на сколько сможет вырасти M после итерации номер $\sqrt{V}$?)

ж) Предложите алгоритм для поиска максимального множества кратчайших увеличивающих путей $P_1, P_2, \ldots, P_k$ для заданного паросочетания M, не имеющих общих вершин, время работы которого — $O(E)$. Покажите, что суммарное время выполнения процедуры HOPCROFT_ KARP составляет $O\left(\sqrt{V}E\right)$.

Заключительные замечания

Транспортные сети и связанные с ними алгоритмы рассматриваются в работах Ахуя (Ahuja), Магнанти (Magnanti) и Орлина (Orlin) [7], Ивена (Even) [87], Лоулера (Lawler) [196], Пападимитриу (Papadimitriou) и Стейглица (Steiglitz) [237], Таржана (Tarjan) [292]. Широкий обзор алгоритмов для задач поиска потоков в сетях можно найти также в книге Голдберга (Goldberg), Тардоса (Tardos) и Таржана [119]. В работе Шрайвера (Schrijver) [267] предлагается интересный исторический обзор исследований в сфере транспортных сетей.

Метод Форда-Фалкерсона представлен в работе Форда (Ford) и Фалкерсона (Fulkerson) [93], которые являются основоположниками формальных исследований ряда задач в области транспортных сетей, включая задачи поиска максимального потока и паросочетаний. Во многих ранних реализациях метода Форда-Фалкерсона поиск увеличивающих путей осуществляется с помощью поиска в ширину; Эдмондс (Edmonds) и Карп (Karp) [86] и, независимо от них, Диниц (Dinic) [76] доказали, что такая стратегия дает полиномиальный по времени алгоритм. Диницу [76] также принадлежит идея использования "тупиковых потоков" (blocking flows); предпотоки впервые предложил Карзанов (Karzanov) [176]. Метод проталкивания предпотока описан в работах Голдберга [117] и Голдберга и Таржана [121]. Голдберг и Таржан приводят алгоритм со временем работы $O\left(V^3\right)$, в котором для хранения множества переполненных вершин используется очередь, а также алгоритм на основе использования динамических деревьев, время работы которого достигает $O\left(VE \lg\left(V^2/E+2\right)\right)$. Некоторые другие исследователи разработали алгоритмы проталкивания предпотока для поиска максимального потока. В работах Ахуя и Орлина [9] и Ахуя, Орлина и Таржана [10] приводятся алгоритмы, использующие масштабирование. Чериян (Cheriyan) и Махешвари (Maheshvari) [55] предложили проталкивать поток из переполненной вершины с максимальной высотой. В работе Черияна и Хейджерапа (Hagerup) [54] предлагается использовать случайные перестановки списков соседей; другие исследователи [14, 178, 241] развили данную идею, предложив искусные методы дерандомизации, что позволило получить ряд более быстрых алгоритмов. Алгоритм, предложенный Кингом (King), Рао (Rao) и Таржаном [178], является самым быстрым из них — время его работы составляет $O(VE \log_{E/(V \lg V)} V)$.

Асимптотически самый быстрый из известных в настоящее время алгоритмов для задачи максимального потока разработан Голдбергом и Рао [120], время его работы равно $O(\min(V^{2/3}, E^{1/2})E \lg(V^2/E+2) \lg C)$, где $C = \max_{(u,v)\in E} c\,(u,v)$. Этот алгоритм не использует метод проталкивания предпотока, он основан на нахождении тупиковых потоков. Все предыдущие алгоритмы, включая рассмотренные в данной главе, используют некоторое понятие расстояния (в алгоритмах проталкивания предпотока используется аналогичное понятие высоты), где каждому ребру неявно присвоена длина 1. В этом же алгоритме используется другой под-

ход: ребрам с высокой пропускной способностью присваивается длина 0, а ребрам с низкой пропускной способностью — длина 1. Неформально при таком выборе длин кратчайшие пути от источника к стоку будут иметь высокую пропускную способность, следовательно, потребуется меньшее количество итераций.

На практике на сегодняшний день при решении задач поиска максимального потока алгоритмы проталкивания предпотока превосходят алгоритмы, основанные на увеличивающих путях и линейном программировании. В исследованиях Черкасски (Cherkassky) и Голдберга [56] подчеркивается важность использования при реализации алгоритма проталкивания предпотока двух эвристик. Первая состоит в том, что в остаточном графе периодически производится поиск в ширину, чтобы получить более точные значения высот. Вторая эвристика — это“эвристика промежутка” (gap heuristic), описанная в упражнении 26.5-5. Авторы пришли к заключению, что наилучшим вариантом метода проталкивания предпотока является вариант, в котором для разгрузки выбирается переполненная вершина с максимальной высотой.

Наилучший известный к настоящему времени алгоритм поиска максимального паросочетания (описанный в задаче 26-7) был предложен Хопкрофтом (Hopcroft) и Карпом (Karp) [152]; время его работы составляет $O\left(\sqrt{V}E\right)$. Задачи поиска паросочетаний подробно рассматриваются в книге Ловаса (Lovasz) и Пламмера (Plummer) [207].

ЧАСТЬ VII

Избранные темы

Введение

Эта часть содержит избранные темы теории алгоритмов, расширяющие и дополняющие материал, ранее изложенный в данной книге. В некоторых главах вводятся новые вычислительные модели, такие как комбинационные схемы или параллельные вычислительные машины. Другие главы охватывают специализированные области знаний, такие как вычислительная геометрия или теория чисел. В двух последних главах обсуждаются некоторые известные ограничения, возникающие при разработке эффективных алгоритмов, а также излагаются основы методов, позволяющих справиться с этими ограничениями.

В главе 27 представлена параллельная модель вычислений: сравнивающие сети. Грубо говоря, сравнивающая сеть — это алгоритм, основанный на одновременном выполнении большого количества сравнений. В этой главе показано, как построить сравнивающую сеть, позволяющую сортировать n чисел за время $O\left(\lg^2 n\right)$.

В главе 28 изучаются эффективные алгоритмы, предназначенные для работы с матрицами. После изучения некоторых основных свойств матриц исследуется алгоритм Штрассена (Strassen), позволяющий перемножить две матрицы $n \times n$ за время $O\left(n^{2.81}\right)$. Затем представлены два общие метода LU-разложения и LUP-разложения, предназначенные для решения системы линейных уравнений по методу Гаусса (методу исключений) за время $O\left(n^3\right)$. Здесь также показано, что перемножение и обращение матриц можно выполнять с одинаковой скоростью. В заключительной части главы показано, как получить приближенное решение системы линейных уравнений методом наименьших квадратов, если эта система не имеет точного решения.

В главе 29 исследуется линейное программирование, цель которого — минимизировать или максимизировать целевую функцию при заданных ограниченных ресурсах и конкурирующих ограничениях. Линейное программирование применяется в самых различных прикладных областях. В главе описывается постановка задач линейного программирования и их решение. В качестве метода решения изложен симплекс-алгоритм, который является одним из древнейших алгоритмов, используемых в линейном программировании. В отличие от многих других алгоритмов, о которых идет речь в этой книге, для симплекс-алгоритма время работы в наихудшем случае не выражается полиномиальной функцией, однако он достаточно эффективен и широко применяется на практике.

В главе 30 изучаются операции над полиномами. Здесь показано, что с помощью такого известного метода обработки сигналов, как быстрое преобразование Фурье (Fast Fourier Transform, FFT), два полинома n-й степени можно перемножить за время $O\left(n \lg n\right)$. В этой главе также исследуются методы эффективной реализации FFT, включая параллельные вычисления.

В главе 31 представлены теоретико-числовые алгоритмы. После обзора элементарной теории чисел здесь описан алгоритм Евклида, предназначенный для вычисления наибольшего общего делителя. Далее представлены алгоритмы для решения модульных линейных уравнений и для возведения числа в степень по модулю другого числа. Затем читатель сможет ознакомиться с важным приложением теоретико-числовых алгоритмов: криптографической системой с открытым ключом RSA. С ее помощью можно не только кодировать сообщения таким образом, чтобы их не могли прочитать посторонние, но и создавать цифровые подписи. Далее в главе представлен рандомизированный тест для простых чисел Миллера-Рабина (Miller-Rabin), позволяющий производить эффективный поиск больших целых чисел, необходимый для реализации схемы RSA. В заключительной части главы описан эвристический ρ-метод Полларда (Pollard) для разбиения целых чисел на множители, а также обсуждаются успехи, достигнутые в этой области.

В главе 32 исследуется задача поиска всех вхождений заданной строки-образца в имеющуюся строку текста; эта задача часто возникает при написании программ, предназначенных для редактирования текста. После ознакомления с "наивным" подходом в этой главе представлен элегантный метод решения данной задачи, разработанный Рабином (Rabin) и Карпом (Karp). Затем, после демонстрации эффективного решения, основанного на теории конечных автоматов, здесь вниманию читателя предложен алгоритм Кнута-Морриса-Пратта (Knuth-Morris-Pratt), позволяющий достичь высокой эффективности за счет предварительной обработки образца.

Тема главы 33 — вычислительная геометрия. После обсуждения основных принципов этого раздела вычислительной математики, в главе показано, как с помощью метода "обметания" можно эффективно определить, содержатся ли какие-нибудь пересечения в множестве прямолинейных отрезков. Два остроумных алгоритма, предназначенных для поиска выпуклой оболочки заданного множества точек, — метод сканирования по Грэхему (Graham's scan) и метод продвижения по Джарвису (Jarvis's march), — также иллюстрируют мощь метода обметания. В заключение в главе описан эффективный алгоритм, предназначенный для поиска пары самых близких точек в заданном множестве точек на плоскости.

Глава 35 посвящена NP-полным задачам. Многие интересные вычислительные задачи являются NP-полными, однако неизвестен ни один алгоритм решения какой бы то ни было из этих задач, время работы которого выражалось бы полиномиальной функцией. В данной главе представлены методы определения того, является ли задача NP-полной. Доказана NP-полнота для нескольких классических задач: определение того, содержит ли граф цикл Гамильтона, определение того, выполнима ли заданная булева формула, и определение того, содержит ли заданное множество чисел такое подмножество, сумма элементов в котором была бы равна заданному значению. В этой главе также доказано, что знаменитая задача о коммивояжере также является NP-полной.

В главе 35 показано, как эффективно находить приближенные решения NP-полных задач с помощью приближенных алгоритмов. Для некоторых NP-полных задач достаточно легко выразить приближенные решения, достаточно близкие к оптимальным, в то время как для других задач даже самые лучшие из известных приближенных алгоритмов работают все хуже по мере увеличения размера задачи. Есть также другой класс задач, для которых наблюдается возрастание времени вычисления с увеличением точности приближенных решений. Эти возможности проиллюстрированы на задаче о вершинном покрытии (представлены невзвешенная и взвешенная версии), на задаче о коммивояжере и других.

ГЛАВА 27

Сортирующие сети

Во второй части этой книги были рассмотрены алгоритмы сортировки, предназначенные для последовательных компьютеров (вычислительных машин, оснащенных памятью с произвольным доступом), в которых в заданный момент времени может выполняться лишь одна операция. В этой главе исследуются алгоритмы сортировки, основанные на модели сравнивающих сетей, в которых одновременно можно выполнять большое количество операций сравнения.

Сравнивающие сети имеют два важных отличия от последовательных компьютеров. Во-первых, в них могут выполняться только сравнения. Это означает, что в этой сети нельзя реализовать такой алгоритм, как сортировка подсчетом. Во-вторых, в отличие от последовательных компьютеров, в которых операции выполняются последовательно, т.е. одна за другой, операции в сравнивающей сети могут выполняться одновременно или "параллельно". Мы сможем убедиться, что эта характеристика позволяет строить сравнивающие сети, сортирующие n величин за время, меньшее линейного.

В начальном разделе главы, разделе 27.1, дается определение сравнивающих и сортирующих сетей. Здесь также представлено естественное определение "времени работы" сравнивающей сети в терминах глубины сети. В разделе 27.2 доказывается "нуль-единичный принцип", позволяющий значительно упростить задачу анализа корректности работы сортирующих сетей.

Эффективная сортирующая сеть, которая будет разработана в этой главе, по сути, представляет собой параллельную версию алгоритма слиянием, изложенную в разделе 2.3.1. Процесс разработки состоит из трех этапов. В разделе 27.3 представлена схема "битонического" сортировщика, который станет нашим основным строительным блоком. В разделе 27.4 битонический сортировщик будет слегка

модифицирован, с тем чтобы получить объединяющую сеть, позволяющую объединять две отсортированные последовательности в одну. Наконец, в разделе 27.5 из этих объединяющих сетей будет собрана сортирующая сеть, позволяющая выполнить сортировку n величин за время $O\left(\lg^2 n\right)$.

27.1 Сравнивающие сети

Сортирующие сети — это такие сравнивающие сети, в которых входные данные всегда сортируются, поэтому есть смысл вначале обсудить сравнивающие сети и их характеристики. Сравнивающая сеть состоит только из сравнивающих устройств и соединяющих их проводов. ***Компаратор***, или ***сравнивающее устройство*** (comparator, рис. 27.1), — это устройство с двумя входами, x и y, и двумя выходами, x' и y', выполняющее такую операцию:

$$x' = \min(x, y),$$
$$y' = \max(x, y).$$

Поскольку представленное на рис. 27.1*а* схематическое изображение сравнивающего устройства слишком громоздкое, мы примем в качестве условного обозначения этого устройства вертикальную линию, показанную на рис. 27.1*б*. Входные величины изображены слева, а выходные — справа, причем меньшая входная величина на выходе переходит вверх, а бóльшая — вниз. Таким образом, можно сказать, что сравнивающее устройство сортирует две входных величины.

В дальнейшем предполагается, что сравнивающее устройство выполняет требуемое действие за время $O(1)$. Другими словами, предполагается, что время, которое проходит с момента подачи входных величин x и y до момента выдачи выходных величин x' и y', равно константе.

Значение передается из одной точки в другую по ***проводу*** (wire). Провода соединяют выход одного сравнивающего устройства с входом другого; в противном случае они являются либо входными каналами в сеть, либо выходными каналами из нее. В этой главе предполагается, что сравнивающая сеть содержит n ***входных проводов*** (input wires) $a_1, a_2, \ldots, a_n$, по которым в сеть поступают сортируемые величины, и n ***выходных проводов*** (output wires) $b_1, b_2, \ldots, b_n$, по которым выдаются вычисленные сетью результаты. Кроме того, мы будем говорить о ***входной***

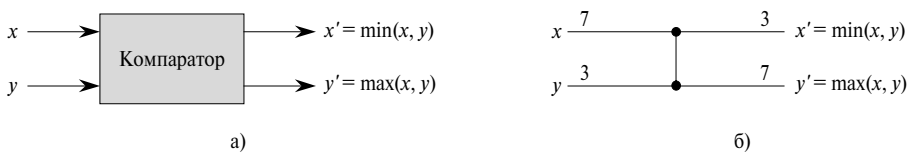


Рис. 27.1. Схематические представления сравнивающего устройства

последовательности (input sequence) $\langle a_1, a_2, \ldots, a_n \rangle$ и ***выходной последовательности*** (output sequence) $\langle b_1, b_2, \ldots, b_n \rangle$, подразумевая под ними наборы величин во входных и выходных проводах. Таким образом, одно и то же название будет использоваться и для проводов, и для значений, которые по ним передаются. Что именно имеется в виду — всегда будет понятно из контекста.

На рис. 27.2 показана ***сравнивающая сеть*** (comparison network), представляющая собой набор сравнивающих устройств, соединенных между собой проводами. В такой сети n входов изображаются в виде n горизонтальных ***линий*** (lines), а сравнивающие устройства — вертикальными отрезками. Заметим, что линия представляет *не* отдельный провод, а последовательность различных проводов, соединяющих разные сравнивающие устройства. Например, верхней линией на рис. 27.2 представлены три провода: входной провод a_1, который подсоединен к входу сравнивающего устройства A; провод, соединяющий верхний выход сравнивающего устройства A со входом сравнивающего устройства C; выходной провод b_1, присоединенный к верхнему выходу сравнивающего устройства C. Ко входу каждого сравнивающего устройства подсоединен провод, который либо является одним из n входных проводов $a_1, a_2, \ldots, a_n$, либо присоединен к выходу другого сравнивающего устройства. Аналогично, к выходу каждого сравнивающего устройства подсоединен провод, который либо является одним из n выходных проводов $b_1, b_2, \ldots, b_n$, либо подсоединен к входу другого сравнивающего устройства. Основной принцип, по которому сравнивающие устройства соединяются между собой, заключается в том, что получившийся в результате граф соединений должен быть ациклическим: если проследить путь, проходящий от выхода какого-нибудь сравнивающего устройства к входу другого сравнивающего устройства, затем — от его выхода к входу следующего сравнивающего устройства и т.д., то этот путь никогда не должен зацикливаться, дважды проходя через одно и то же сравнивающее устройство. Таким образом, как видно из рис. 27.2, на схеме сравнивающей сети входы можно расположить слева, а выходы — справа; при этом данные в процессе обработки будут продвигаться слева направо.

Выходные значения производятся сравнивающим устройством лишь в том случае, когда на его вход подаются обе сравниваемые величины. Например, на рис. 27.2*а* предполагается, что в начальный момент времени на входные провода последовательности подается последовательность $\langle 9, 5, 2, 6 \rangle$. Таким образом, в этот момент времени только сравнивающие устройства A и B располагают всеми необходимыми входными величинами. С учетом того что для вычисления выходных значений сравнивающему устройству требуется единичный промежуток времени, сравнивающие устройства A и B выдадут выходные величины в момент времени 1; результат показан на рис. 27.2*б*. Заметим, что результат в сравнивающих устройствах A и B появляется одновременно, или “параллельно”. Далее, в момент времени 1 полным набором входных величин располагают сравнивающие устройства C и D, но не E. По истечении еще одного единичного интервала

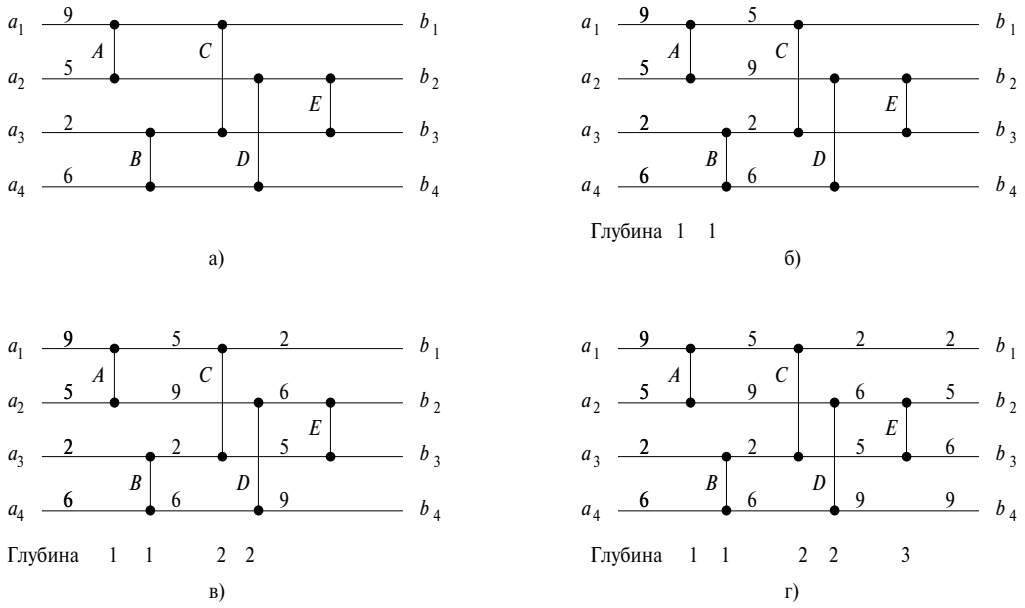


Рис. 27.2. Сравнивающая сеть на 4 входа и 4 выхода, которая фактически является сортирующей сетью

времени, в момент времени 2, выдадут результат сравнивающие устройства C и D (рис. 27.2*в*). Они также работают параллельно. К верхнему выходу сравнивающего устройства C и нижнему выходу сравнивающего устройства D подсоединены выходные провода b_1 и b_4 сравнивающей сети соответственно. Поэтому в момент времени 2 на эти выходные провода подаются их результирующие значения. В этот же момент времени входные значения подаются на сравнивающее устройство E, и, как видно из рис. 27.2*г*, результат выдается на этом сравнивающем устройстве в момент времени 3. Полученные величины передаются по сети по выходным проводам b_2 и b_3, и теперь вычисление выходной последовательности $\langle 2, 5, 6, 9 \rangle$ завершено.

Исходя из предположения, что каждое сравнение выполняется в течение единичного промежутка времени, можно определить "время работы" сравнивающей сети, т.е. время, которое требуется для подачи всех выходных значений на выходы сети с момента, когда на ее входы поданы все необходимые сигналы. Это время естественно считать равным максимальному количеству сравнивающих устройств, через которые проходит входное значение на пути следования от входного провода к выходному. Выражаясь более строго, дадим определение ***глубины*** (depth) провода. Глубина входного провода сравнивающей сети равна нулю. Далее, если глубины входных проводов сравнивающего устройства равны d_x и d_y, то глубина его выходных проводов равна $\max(d_x, d_y) + 1$. Поскольку в сравниваю-

щей сети отсутствуют циклы, глубина провода определяется однозначно, и можно определить глубину сравнивающего устройства как величину, равную глубине его выходных проводов. На рис. 27.2 показаны глубины изображенных на нем сравнивающих устройств. Глубина сравнивающей сети равна максимальной глубине выходного провода или, что эквивалентно, максимальной глубине сравнивающего устройства. Например, глубина сравнивающей сети, представленной на рис. 27.2, равна 3, поскольку именно этому значению равна глубина сравнивающего устройства E. Если для вычисления результата каждому сравнивающему устройству требуется единичный интервал времени, и если входные значения подаются в нулевой момент времени, сравнивающее устройство на глубине d выдает результат в момент времени d. Таким образом, глубина сети равна времени, в течение которого сеть обрабатывает входные значения и выдает результат на все выходные провода.

Сортирующая сеть (sorting network) — это сравнивающая сеть, в которой выходная последовательность является монотонно неубывающей (т.е. выполняется цепочка неравенств $b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n$) для *любой* входной последовательности. Конечно же, не все сравнивающие сети являются сортирующими, но сеть на рис. 27.2 именно такая. Чтобы понять причину этого, заметим, что по истечении единичного интервала времени минимальная из четырех входных величин появится либо на верхнем выходе сравнивающего устройства A, либо на верхнем выходе сравнивающего устройства B. Поэтому по истечении двух единичных интервалов времени эта величина должна оказаться на верхнем выходе сравнивающего устройства C. Аналогично, по истечении двух единичных интервалов времени максимальная из четырех входных величин появится на нижнем выходе сравнивающего устройства D. Все, что остается сравнивающему устройству E, — это убедиться, что два средних значения займут правильные позиции, что и произойдет в момент времени 3.

Сравнивающая сеть напоминает процедуру в том смысле, что она указывает, как должны выполняться операции сравнения, но отличается от нее тем, что ее ***размер*** (size), т.е. количество содержащихся в сети сравнивающих устройств, зависит от количества входных и выходных величин. Таким образом, мы должны описывать “семейства” сравнивающих сетей. Например, цель этой главы — разработать семейство эффективных сортирующих сетей SORTER. Конкретная сеть в пределах этого семейства задается семейным именем и количеством входов (которое равно количеству выходов). Например, сортирующая сеть с n входами и n выходами в семействе SORTER получает имя SORTER$[n]$.

Упражнения

27.1-1. Какие значения появятся на всех проводах сети, представленной на рис. 27.2, если входная последовательность имеет вид $\langle 9, 6, 5, 2\rangle$.

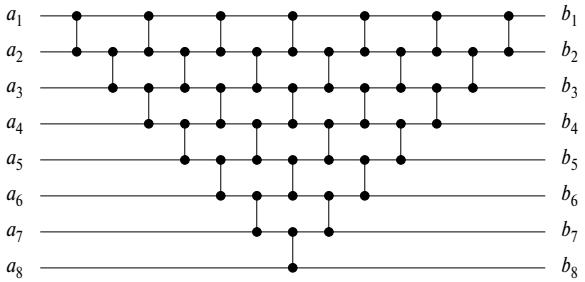


Рис. 27.3. Сортирующая сеть, основанная на методе сортировки вставками, для упражнения 27.1-6

27.1-2. Пусть n — степень двойки. Покажите, как сконструировать сравнивающую сеть глубиной $\lg n$ с n входами и n выходами, в которой верхний выходной провод всегда передает минимальное входное значение, а нижний выходной провод — максимальное.

27.1-3. К сортирующей сети всегда можно добавить сравнивающее устройство таким образом, чтобы в результате получилась сравнивающая сеть, которая не является сортирующей. Покажите, как добавить сравнивающее устройство в сеть, изображенную на рис. 27.2, чтобы получившаяся сеть перестала сортировать все входные перестановки.

27.1-4. Докажите, что глубина сортирующей сети на n входов ограничена снизу величиной $\lg n$.

27.1-5. Докажите, что количество сравнивающих устройств в произвольной сортирующей сети равно $\Omega(n \lg n)$.

27.1-6. Рассмотрим сортирующую сеть, изображенную на рис. 27.3. Докажите, что это на самом деле сортирующая сеть и опишите, как ее структура связана со структурой алгоритма, работающего по методу вставок (раздел 2.1).

27.1-7. Сравнивающую сеть на n входов, содержащую c сравнивающих устройств, можно представить в виде списка c пар целых чисел, которые находятся в интервале от 1 до n. Если две пары содержат общее целое число, порядок соответствующих сравнивающих устройств сети определяется порядком пар в списке. Опишите с помощью этого представления (последовательный) алгоритм со временем работы $O(n + c)$, позволяющий определить глубину сравнивающей сети.

⋆ 27.1-8. Предположим, что кроме обычных сравнивающих устройств мы располагаем “перевернутыми” сравнивающими устройствами, минимальное значение в которых поступает на нижний провод, а максимальное —

на верхний. Покажите, как преобразовать любую сортирующую сеть, содержащую c стандартных и перевернутых сравнивающих устройств в сортирующую сеть, в которой используется c стандартных сравнивающих устройств. Докажите корректность предложенного метода преобразования.

27.2 Нуль-единичный принцип

Нуль-единичный принцип (zero-one principle) утверждает, что если сортировочная сеть правильно работает для всех наборов входных величин, составленных из элементов множества $\{0, 1\}$, то она правильно работает для любых входных чисел. (Эти числа могут быть целыми, действительными или любым другим подмножеством величин, извлеченных из произвольного линейно упорядоченного множества.) При построении сортирующих сетей или других сравнивающих сетей нуль-единичный принцип позволяет ограничиться операциями, выполняющимися над последовательностью, составленной только из нулей и единиц. Построив сортирующую сеть и доказав, что с ее помощью сортируются все последовательности из нулей и единиц, можно обратиться к нуль-единичному принципу и доказать, что она надлежащим образом сортирует последовательности, составленные из произвольных величин.

Доказательство нуль-единичного принципа основано на понятии монотонно неубывающей функции (раздел 3.2).

Лемма 27.1. Если сравнивающая сеть преобразует входную последовательность $a = \langle a_1, a_2, \ldots, a_n \rangle$ в выходную последовательность $b = \langle b_1, b_2, \ldots, b_n \rangle$, то для любой монотонно неубывающей функции f эта сеть преобразует входную последовательность $f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$ в выходную последовательность $f(b) = \langle f(b_1), f(b_2), \ldots, f(b_n) \rangle$.

Доказательство. Сначала докажем, что если функция f монотонно неубывает, то отдельное сравнивающее устройство с входными значениями $f(x)$ и $f(y)$ выдаст результат $f(\min(x, y))$ и $f(\max(x, y))$. Затем докажем лемму по индукции.

Чтобы доказать сформулированное выше утверждение, рассмотрим сравнивающее устройство, на вход которого подаются величины x и y. Верхнее выходное значение такого устройства равно $\min(x, y)$, а его нижнее выходное значение — $\max(x, y)$. Предположим, что теперь на вход сравнивающего устройства подаются значения $f(x)$ и $f(y)$ (рис. 27.4). На верхний выход сравнивающего устройства выдается значение $\min(f(x), f(y))$, а на нижний выход — значение $\max(f(x), f(y))$. Поскольку функция f монотонно неубывающая, из неравенства

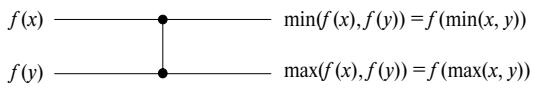


Рис. 27.4. Работа сравнивающего устройства, иллюстрирующая доказательство леммы 27.1; функция f — монотонно неубывающая

$x \leqslant y$ следует неравенство $f(x) \leqslant f(y)$. Следовательно, выполняются тождества

$$\min(f(x), f(y)) = f(\min(x, y)),$$
$$\max(f(x), f(y)) = f(\max(x, y)).$$

Таким образом, если на вход сравнивающего устройства подать значения $f(x)$ и $f(y)$, то на его выходе получим значения $\min(f(x), f(y))$ и $\max(f(x), f(y))$. На этом доказательство утверждения завершается.

Методом индукции по шагам продвижения вглубь по каждому проводу в сравнивающей сети общего вида можно доказать более сильное утверждение, чем то, которое сформулировано в лемме. Оно заключается в том, что если какой-нибудь провод получает значение a_i в результате подачи в сеть входной последовательности a, то при подаче входной последовательности $f(a)$ он получает значение $f(a_i)$. Поскольку в этом утверждении идет речь обо всех проводах, в том числе и о выходных, его доказательство станет доказательством леммы.

В качестве базиса индукции рассмотрим провод на глубине 0, т.е. входной провод a_i. Результат получается тривиальным образом: если на вход сети подается сигнал $f(a)$, то по входному проводу передается сигнал $f(a_i)$. В качестве шага индукции рассмотрим провод, расположенный на глубине $d \geqslant 1$. Этот провод является выходным в сравнивающем устройстве, расположенном на глубине d, а входные провода этого устройства находятся на глубине, строго меньшей d. В соответствии с гипотезой индукции, если входные провода сравнивающего устройства передают сигналы a_i и a_j при подаче в сеть входной последовательности a, то при подаче входной последовательности $f(a)$ они передают сигналы $f(a_i)$ и $f(a_j)$. Согласно доказанному ранее утверждению, выходные провода рассматриваемого сравнивающего устройства передают сигналы $f(\min(a_i, a_j))$ и $f(\max(a_i, a_j))$. Так как при подаче в сеть входной последовательности a они передают сигналы $\min(a_i, a_j)$ и $\max(a_i, a_j)$, то лемма доказана. ■

В качестве примера применения леммы 27.1 рассмотрим рис. 27.5. Сортирующая сеть, изображенная в части *а* этого рисунка, идентична той, что показана на рис. 27.5. В части *б* этого рисунка показана эта же сеть в ситуации, когда ее входная последовательность образована путем применения к входной последовательности из части *а* монотонно неубывающей функции $f(x) = \lceil x/2 \rceil$.

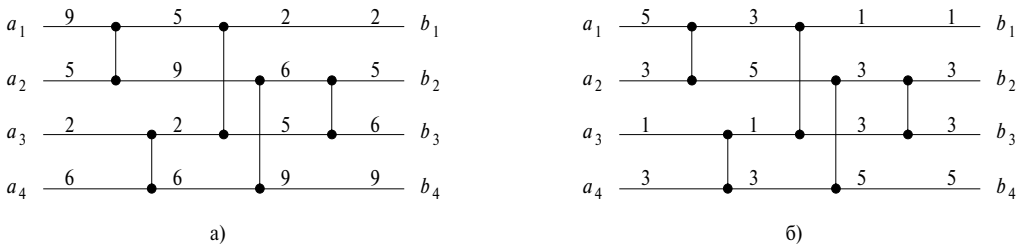


Рис. 27.5. Применение неубывающей функции ко входу сортирующей сети

Если сравнивающая сеть является сортирующей, то лемма 27.1 позволяет доказать сформулированное ниже замечательное утверждение.

Теорема 27.2 (Нуль-единичный принцип). Если сравнивающая сеть с n входами правильно сортирует все 2^n возможных последовательностей, состоящих из нулей и единиц, то она правильно сортирует все последовательности чисел произвольного вида.

Доказательство. Предположим, что это не так, и сеть сортирует все нуль-единичные последовательности, но существует последовательность, состоящая из произвольных чисел, которую сеть не может правильно отсортировать. Другими словами, существует входная последовательность $\langle a_1, a_2, \ldots a_n \rangle$, которая содержит элементы a_i и a_j, связанные соотношением $a_i < a_j$, но после прохождения сети значение a_j расположено в выходной последовательности перед значением a_i. Определим следующую монотонно неубывающую функцию f:

$$f(x) = \begin{cases} 0 & \text{при } x \leqslant a_i, \\ 1 & \text{при } x > a_i. \end{cases}.$$

Поскольку в выходной последовательности сеть помещает значение a_j перед значением a_i, если на вход подается последовательность $\langle a_1, a_2, \ldots a_n \rangle$, то из леммы 27.1 следует, что она разместит значение $f(a_j)$ перед значением $f(a_i)$ в выходной последовательности, если на вход сети подается последовательность $\langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$. Но так как $f(a_j) = 1$ и $f(a_i) = 0$, мы приходим к противоречию, что сеть не в состоянии правильно отсортировать нуль-единичную последовательность $\langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$. ■

Упражнения

27.2-1. Докажите, что в результате применения монотонно неубывающей функции к отсортированной последовательности получится отсортированная последовательность.

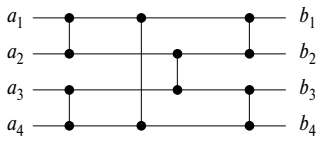


Рис. 27.6. Сортирующая сеть, предназначенная для сортировки четырех чисел

27.2-2. Докажите, что сравнивающая сеть с n входами правильно сортирует входную последовательность $\langle n, n-1, \ldots, 1\rangle$ тогда и только тогда, когда она правильно сортирует нуль-единичные последовательности $\langle 1, 0, 0, \ldots, 0, 0\rangle$, $\langle 1, 1, 0, \ldots, 0, 0\rangle$, $\ldots$, $\langle 1, 1, 1, \ldots, 1, 0\rangle$, состоящие из $n-1$ членов.

27.2-3. Докажите с помощью нуль-единичного принципа, что сравнивающая сеть, показанная на рис. 27.6, является сортирующей.

27.2-4. Сформулируйте и докажите утверждение, аналогичное нуль-единичному принципу, для модели дерева решений. (*Указание*: не забудьте корректно обработать равенство.)

27.2-5. Докажите, что для всех $i = 1, 2, \ldots, n-1$ в сортирующей сети на n входов между i-й и $(i+1)$-й линиями должно содержаться по крайней мере одно сравнивающее устройство.

27.3 Битоническая сортирующая сеть

Первый этап конструирования эффективной сортирующей сети состоит в том, чтобы построить сравнивающую сеть, которая может сортировать любую ***битоническую последовательность*** (bitonic sequence): последовательность, которая сначала монотонно возрастает, а затем монотонно убывает, или последовательность, которая приводится к такому виду путем циклического сдвига. Примерами битонических последовательностей являются последовательности $\langle 1, 4, 6, 8, 3, 2\rangle$, $\langle 6, 9, 4, 2, 3, 5\rangle$ и $\langle 9, 8, 3, 2, 4, 6\rangle$. Любая последовательность, состоящая всего из одного числа или двух чисел, является битонической. Такие последовательности представляют собой своего рода граничный случай. Нуль-единичные битонические последовательности обладают простой структурой. Они имеют вид $0^i1^j0^k$ или $1^i0^j1^k$ для некоторых $i, j, k \geqslant 0$. Заметим, что последовательность, которая либо монотонно возрастает, либо монотонно убывает, — тоже битоническая.

Битонический сортировщик, который нам предстоит сконструировать, представляет собой сортирующую сеть, которая сортирует битонические последовательности, состоящие из нулей и единиц. В упражнении 27.3-6 предлагается по-

казать, что битонический сортировщик может сортировать битонические последовательности, образованные произвольными числами.

Полуфильтр

Битонический сортировщик состоит из нескольких каскадов, каждый из которых называется ***полуфильтром*** (half-cleaner). Каждый полуфильтр — это сравнивающая сеть глубиной 1, в которой i-я входная линия сравнивается со входной линией под номером $i + n/2$, где $i = 1, 2, \ldots, n/2$. (Предполагается, что n — четное.) На рис. 27.7 показана сеть HALF_CLEANER[8], — полуфильтр на 8 входов и 8 выходов, и приведены два примера нуль-единичных входных и выходных наборов значений. Если входные последовательности — битонические, то каждый элемент из верхней половины выходной последовательности не превышает значения любого элемента из нижней половины этой последовательности. Кроме того, обе половины образуют битонические последовательности.

Если на вход полуфильтра подается битоническая последовательность из нулей и единиц, то на его выход выдается последовательность, в которой меньшие значения расположены в верхней половине, большие — в нижней половине, причем обе половины образуют битонические последовательности. По крайней мере одна из половинных подпоследовательностей выходной последовательности является ***однородной*** (clean) — состоящей либо из нулей, либо из единиц. Именно благодаря этому свойству такая сеть получила название “полуфильтр”. (Заметим,

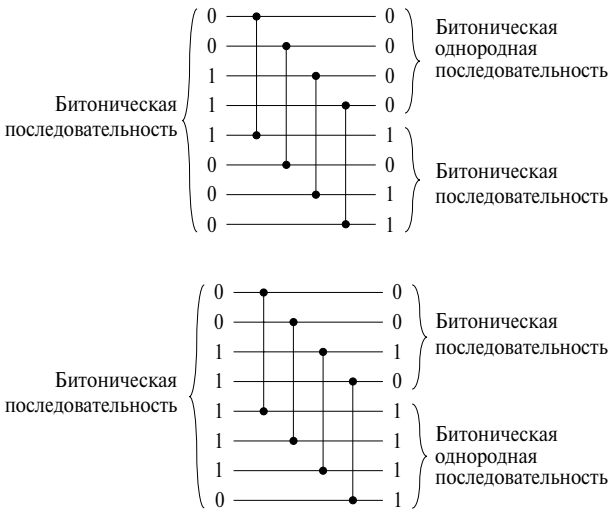


Рис. 27.7. Сравнивающая сеть HALF_CLEANER[8]

что все однородные последовательности — битонические.) В сформулированной ниже лемме доказываются эти свойства полуфильтров.

Лемма 27.3. Если на вход полуфильтра подается битоническая последовательность из нулей и единиц, то его выход обладает такими свойствами: верхняя и нижняя половины образуют битонические последовательности; каждый элемент из верхней половины по величине не превосходит любой элемент из нижней половины; хотя бы одна половина является однородной.

Доказательство. Для всех $i = 1, 2, \ldots, n/2$ сравнивающая сеть HALF_CLEANER$[n]$ сравнивает входные величины с номерами i и $i + n/2$. Без потери общности предположим, что входная последовательность имеет вид $00\ldots011\ldots100\ldots0$. (Рассуждения для входной последовательности вида $11\ldots100\ldots011\ldots1$ аналогичны.) В зависимости от того в каком блоке из последовательно расположенных нулей или единиц находится средняя точка $n/2$ входной последовательности, можно выделить три случая, причем один из этих случаев (тот, когда средняя точка приходится на блок из единиц), в свою очередь, разбивается на два частных случая. Все эти четыре частных случая показаны на рис. 27.8. Последовательности из нулей выделены белым фоном, а последовательности из единиц — серым. Случаи, в которых точка деления пополам приходится на последовательность единиц, представлены в частях *а–б*, а случаи, в которых она приходится на последовательность нулей, — в частях *в–г*. Для каждого из них лемма выполняется. ■

Битонический сортировщик

Рекурсивно комбинируя полуфильтры, как показано на рис. 27.9, можно составить ***битонический сортировщик*** (bitonic sorter), представляющий собой сеть, которая сортирует битонические последовательности. Первый модуль сети BITONIC_SORTER$[n]$ представляет собой модуль HALF_CLEANER$[n]$, который, согласно лемме 27.3, выдает две битонические последовательности половинного размера, причем каждый элемент из верхней половины не превышает по величине любой элемент из нижней половины. Таким образом, сортировку можно завершить с помощью двух копий модуля BITONIC_SORTER$[n/2]$, рекурсивно сортирующий обе половины. На рис. 27.9*а* рекурсия показана явно. Здесь после модуля HALF_CLEANER$[n]$ расположены два модуля HALF_CLEANER$[n/2]$. На рис. 27.9*б* схема представлена в развернутом виде, демонстрируя полуфильтры все меньшего размера, образующие остальную часть битонического сортировщика. Каждый полуфильтр выделен затененным фоном. Возле проводов приведены примеры возможных нуль-единичных значений. Глубина $D(n)$ сети BITONIC_SORTER$[n]$ выра-

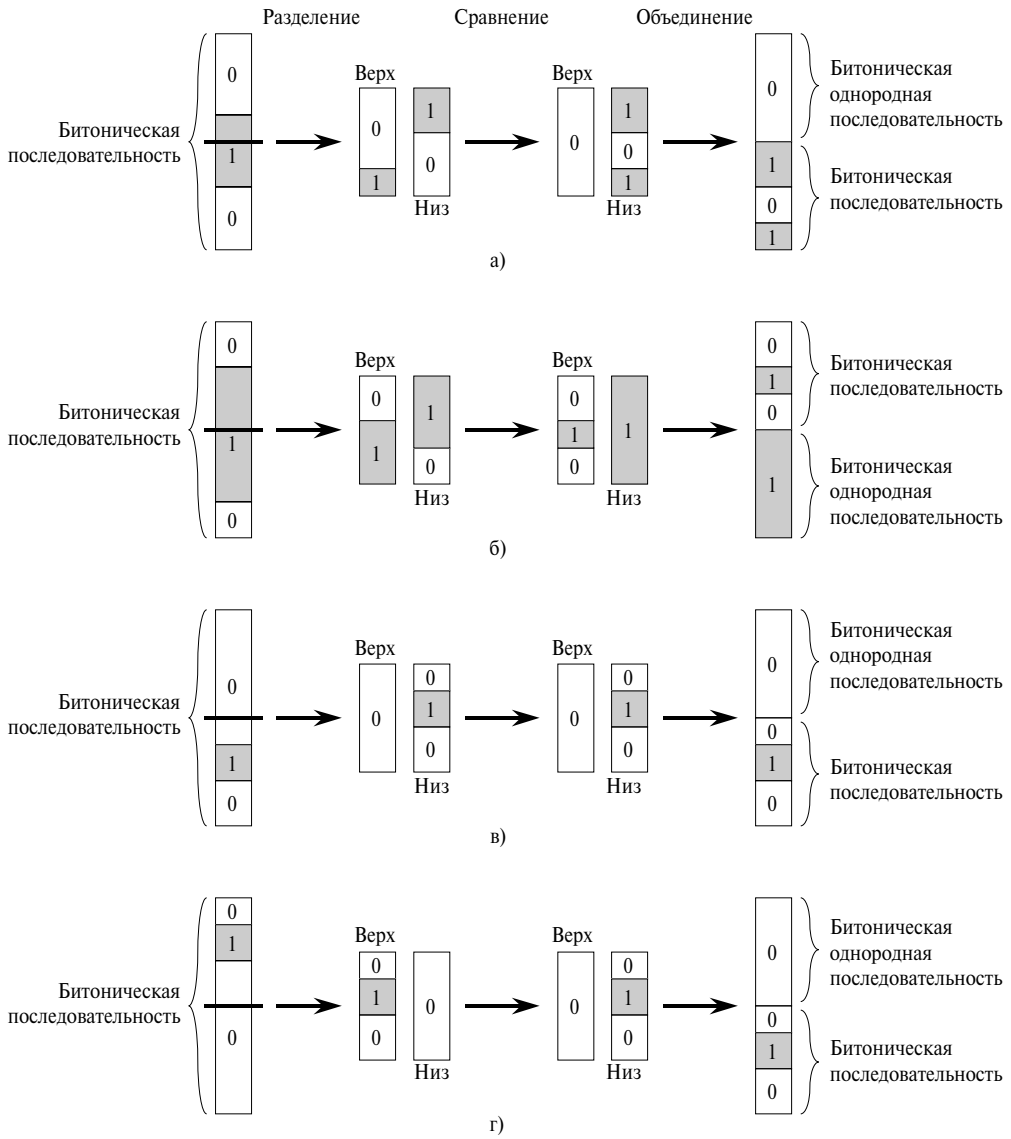


Рис. 27.8. Возможные частные случаи сравнения в сети HALF_CLEANER$[n]$

жается рекуррентным соотношением

$$D(n) = \begin{cases} 0 & \text{при } n = 1, \\ D(n/2) + 1 & \text{при } n = 2^k,\ k \geqslant 1, \end{cases}$$

решение которого имеет вид $D(n) = \lg n$.

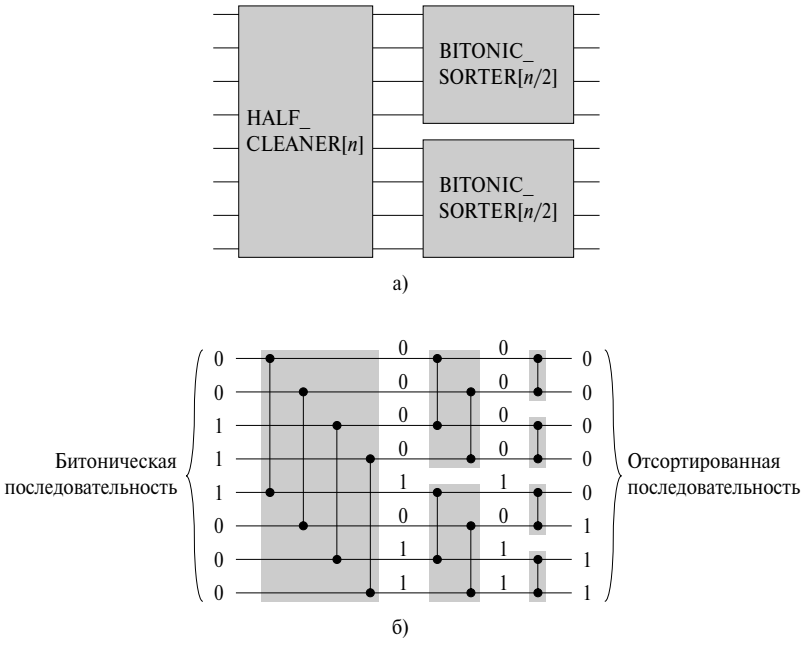


Рис. 27.9. Сравнивающая сеть BITONIC_SORTER$[n]$ для $n = 8$

Таким образом, нуль-единичную битоническую последовательность можно отсортировать с помощью сети BITONIC_SORTER, глубина которой равна $\lg n$. Из утверждения, аналогичному нуль-единичному принципу, которое предлагается доказать в упражнении 27.3-6, следует, что с помощью такой сети можно отсортировать любую битоническую последовательность, состоящую из произвольных чисел.

Упражнения

27.3-1. Сколько нуль-единичных битонических последовательностей длины n можно образовать?

27.3-2. Покажите, что сеть BITONIC_SORTER$[n]$, где n — степень двойки, содержит $\Theta(n \lg n)$ сравнивающих устройств.

27.3-3. Опишите, как сконструировать битонический сортировщик глубиной $O(\lg n)$, если n не равно степени двойки.

27.3-4. Докажите, что если на вход полуфильтра подается битоническая последовательность, состоящая из произвольных чисел, то выходная последовательность обладает следующими свойствами: верхняя и нижняя половины выходной последовательности являются битоническими, и каждый

элемент из верхней половины не превышает по величине ни одного элемента из нижней половины.

27.3-5. Рассмотрим две последовательности, состоящие из нулей и единиц. Докажите, что если каждый элемент одной из них не превышает по величине элементы другой последовательности, то одна из двух последовательностей является однородной.

27.3-6. Докажите такой аналог нуль-единичного принципа для битонических сортирующих сетей: сравнивающая сеть, которая может отсортировать любую битоническую последовательность из нулей и единиц, может также отсортировать любую битоническую последовательность, состоящую из произвольных чисел.

27.4 Объединяющая сеть

Наша сортирующая сеть будет составлена из ***объединяющих сетей*** (merging networks), в роли которых выступают сети, способные объединять две отсортированные входные последовательности в одну отсортированную выходную последовательность. Модифицируем сеть BITONIC_SORTER$[n]$, с тем чтобы создать объединяющую сеть MERGER$[n]$. Как и в случае с битоническим сортировщиком, корректность работы объединяющей сети достаточно доказать для входных последовательностей, состоящих из нулей и единиц. В упражнении 27.4-1 предлагается обобщить это доказательство на случай произвольных входных значений.

Идея объединяющей сети основывается на таком интуитивном соображении. Если имеются две отсортированные последовательности, то в результате инвертирования второй последовательности и объединения ее с первой последовательностью получится битоническая последовательность. Например, пусть нуль-единичные последовательности X и Y имеют вид: $X = 00000111$ и $Y = 00001111$. После инвертирования последовательности Y получаем $Y^{\mathrm{R}} = 11110000$. Объединив последовательности X и Y^{R}, получим битоническую последовательность 0000011111110000. Таким образом, чтобы объединить две входные последовательности X и Y, достаточно выполнить битоническую сортировку последовательности X, объединенной с последовательностью Y^{R}.

Чтобы составить сеть MERGER$[n]$, можно модифицировать первый полуфильтр, входящий в состав сети BITONIC_SORTER$[n]$. Главная проблема в том, чтобы выполнить явное инвертирование второй половины входной последовательности. Чтобы объединить две заданные отсортированные последовательности $\langle a_1, a_2, \ldots, a_{n/2} \rangle$ и $\langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n \rangle$, нужно добиться, чтобы выполнялась битоническая сортировка последовательности $\langle a_1, a_2, \ldots, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+1} \rangle$. Поскольку в первом полуфильтре сети BITONIC_SORTER$[n]$ сравниваются входные значения под номерами i и $n/2 + 1$ для всех $i = 1, 2, \ldots, n/2$, в первом моду-

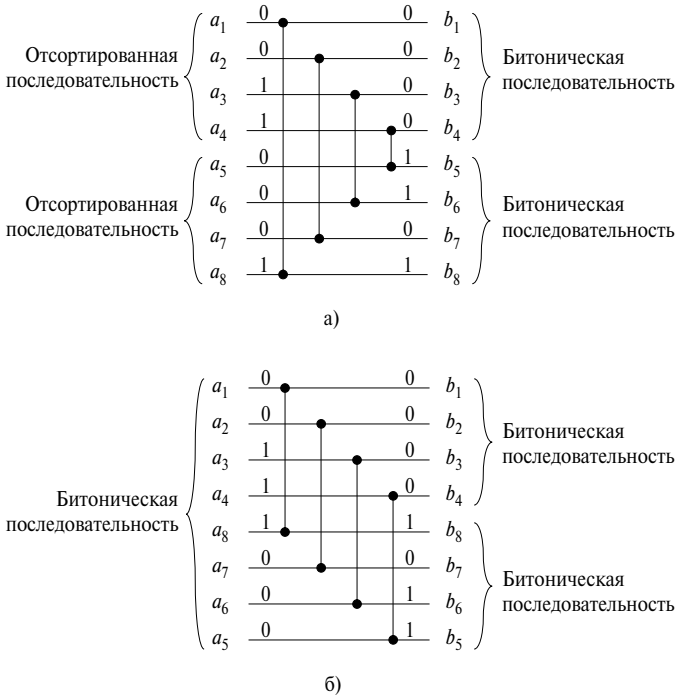


Рис. 27.10. Сравнение первого модуля сети MERGER$[n]$ с первым модулем HALF_CLEANER$[n]$ при $n = 8$

ле объединяющей сети необходимо сравнивать входные значения под номерами i и $n - 1 + i$. На рис. 27.10 показано данное соответствие. На первом этапе в сети MERGER$[n]$ две монотонные входные последовательности $\langle a_1, a_2, \ldots, a_{n/2} \rangle$ и $\langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n \rangle$ преобразуются в битонические последовательности $\langle b_1, b_2, \ldots, b_{n/2} \rangle$ и $\langle b_{n/2+1}, b_{n/2+2}, \ldots, b_n \rangle$ (часть *а* рисунка). В части *б* этого рисунка проиллюстрирована эквивалентная операция в сети HALF_CLEANER$[n]$. Здесь битоническая входная последовательность $\langle a_1, a_2, \ldots, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+1} \rangle$ преобразуется в две битонические последовательности $\langle b_1, b_2, \ldots, b_{n/2} \rangle$ и $\langle b_n, b_{n-1}, \ldots, b_{n/2+1} \rangle$. Единственное отличие этой схемы заключается в том, что выходы в нижней части первого модуля сети MERGER$[n]$ расположены в обратном порядке по сравнению с выходами обычного полуфильтра. Однако поскольку последовательность, полученная в результате инвертирования битонической последовательности, тоже является битонической, верхняя и нижняя половины последовательности, полученной на выходе первого модуля объединяющей сети, удовлетворяют свойствам леммы 27.3. Таким образом, можно параллельно выполнить битоническую сортировку верхней и нижней частей последовательности, чтобы на выходе объединяющей сети получить отсортированную последовательность.

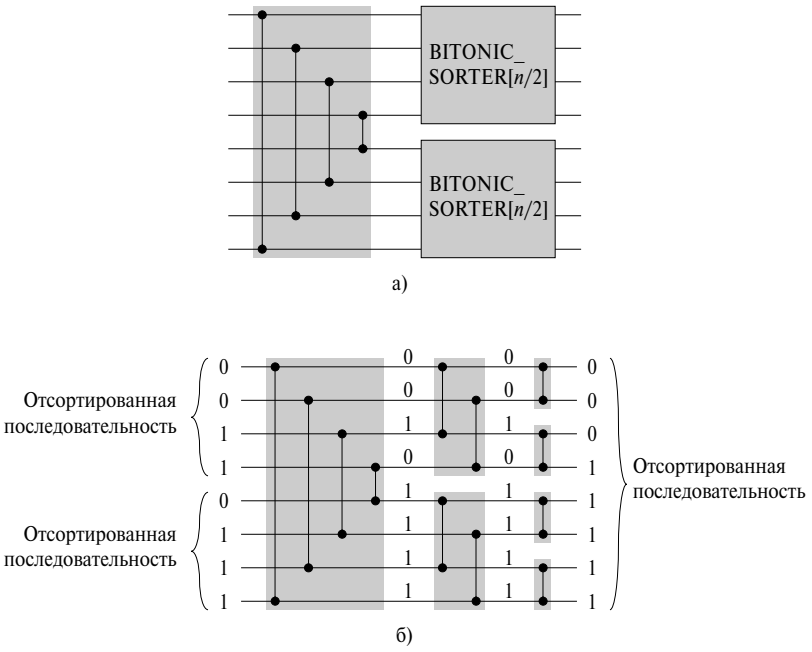


Рис. 27.11. Сеть, объединяющая две входные отсортированные последовательности, в одну выходную отсортированную последовательность при $n = 8$

Получившаяся в результате объединяющая сеть показана на рис. 27.11. Лишь первый модуль сети MERGER$[n]$ отличается от модуля сети BITONIC_SORTER$[n]$, следовательно, глубина обеих сетей равна $\lg n$. Сеть MERGER$[n]$ можно рассматривать как сеть BITONIC_SORTER$[n]$, в которой первый полуфильтр изменен таким образом, что он сравнивает входные значения с номерами i и $n - i + 1$ при $i = 1, 2, \ldots, n/2$. В части *а* этого рисунка сеть разбита на модули; за первым модулем в ней следуют два параллельно расположенных модуля BITONIC_SORTER$[n/2]$. В части *б* изображена та же сеть, в которой рекурсия раскрыта подробнее. Возле проводов указаны примеры нуль-единичных величин, а отдельные модули выделены затенением.

Упражнения

27.4-1. Докажите для объединяющих сетей утверждение, аналогичное нуль-единичному принципу. Другими словами, покажите, что сравнивающая сеть, которая объединяет две произвольные монотонно неубывающие последовательности из нулей и единиц, могут объединять две произвольные монотонно неубывающие последовательности, состоящие из любых чисел.

27.4-2. Сколько различных нуль-единичных последовательностей необходимо подать на вход сравнивающей сети, чтобы проверить, что она действительно является объединяющей сетью?

27.4-3. Покажите, что любая сеть, которая объединяет один элемент с $n-1$ отсортированными элементами, выдавая отсортированную последовательность длины n, должна иметь глубину не менее $\lg n$.

⋆ 27.4-4. Рассмотрим объединяющую сеть с входной последовательностью $a_1, a_2, \ldots, a_n$ для n, равных степени двойки, в которых две монотонные объединяемые последовательности имеют вид $\langle a_1, a_3, \ldots, a_{n-1}\rangle$ и $\langle a_2, a_4, \ldots, a_n\rangle$. Докажите, что количество сравнений, которые выполняются в объединяющих сетях этого вида, равно $\Omega(n \lg n)$. Почему эта нижняя граница представляет интерес? (*Указание*: разбейте сравнивающие устройства на три множества.)

⋆ 27.4-5. Докажите, что для любой объединяющей сети, независимо от порядка входных последовательностей, требуется $\Omega(n \lg n)$ сравнивающих устройств.

27.5 Сортирующая сеть

Теперь у нас имеются все инструменты, необходимые для построения сети, которая способна отсортировать любую входную последовательность. В сортирующей сети SORTER$[n]$ с помощью объединяющей сети реализована параллельная версия сортировки слиянием, описанная в разделе 2.3.1. Построение и работа сортирующей сети проиллюстрирована на рис. 27.12.

На рис. 27.12*а* показана рекурсивная схема сети SORTER$[n]$. Последовательность из n входных элементов сортируется путем (параллельной) рекурсивной сортировки двух подпоследовательностей длиной $n/2$, каждая в двух модулях SORTER$[n/2]$. Затем обе полученные в результате последовательности объединяются с помощью модуля MERGER$[n]$. Граничный случай рекурсии имеет место при $n = 1$. При этом 1-элементную последовательность можно отсортировать с помощью провода, потому что она уже отсортирована. На рис. 27.12*б* показан результат более подробного представления рекурсии, а на рис. 27.12*в* — сама сеть, полученная в результате замены блоков MERGER, изображенных на рис. 27.12*б*, реальными объединяющими сетями. Кроме того, на рис. 27.12*в* указаны глубины всех сравнивающих устройств, а возле проводов — примеры нуль-единичных значений.

В сети SORTER$[n]$ данные проходят $\lg n$ этапов. Каждый отдельный входной элемент сети представляет собой уже отсортированную одноэлементную последовательность. Первый модуль сети SORTER$[n]$ состоит из $n/2$ копий подсети MERGER$[2]$, которые подключены параллельно и объединяют пары 1-элементных

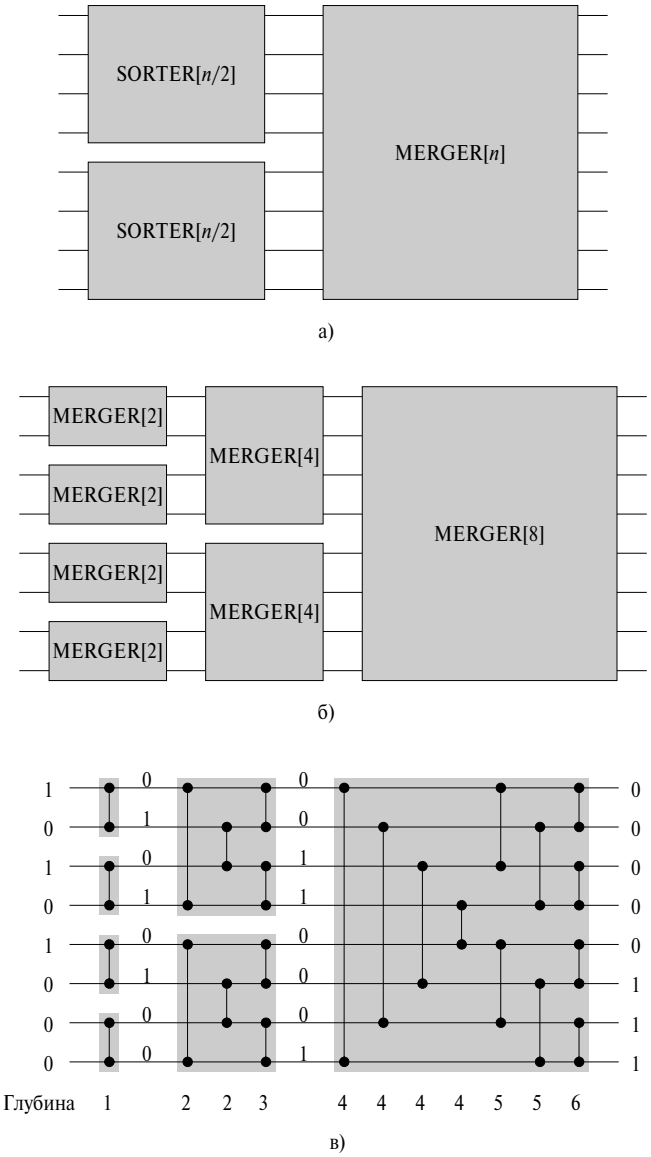


Рис. 27.12. Сортирующая сеть SORTER$[n]$, построенная путем рекурсивного совмещения объединяющих сетей

последовательностей в отсортированные последовательности из двух элементов. Второй модуль состоит из $n/4$ копий подсети MERGER$[4]$, объединяющих пары этих двухэлементных отсортированных последовательностей в отсортированные последовательности из четырех элементов. В общем случае модуль под номером

k, где $k = 1, 2, \ldots, \lg n$, состоит из $n/2^k$ копий подсети MERGER$[2^k]$, объединяющих пары 2^{k-1}-элементных отсортированных последовательностей в отсортированные последовательности длины 2^k. На выходе последнего модуля выдается одна отсортированная последовательность, содержащая все входные значения. По индукции можно показать, что такая сеть сортирует все нуль-единичные последовательности, а значит, согласно нуль-единичному принципу (теорема 27.2), она способна сортировать произвольные значения.

Можно провести рекурсивный анализ глубины сортирующей сети. Глубина $D(n)$ сети SORTER$[n]$ равна сумме глубины $D(n/2)$ подсети SORTER$[n/2]$ (всего имеется две копии этой подсети, но они работают параллельно) и глубины $\lg n$ модуля MERGER$[n]$. Следовательно, глубина сети SORTER$[n]$ выражается рекуррентным соотношением

$$D(n) = \begin{cases} 0 & \text{при } n = 1, \\ D(n/2) + \lg n & \text{при } n = 2^k,\ k \geqslant 1, \end{cases}$$

решение которого имеет вид $D(n) = \Theta\left(\lg^2 n\right)$. (Здесь использована версия основного метода из упражнения 4.4-2.) Таким образом, в параллельной сети n чисел можно отсортировать за время $O\left(\lg^2 n\right)$.

Упражнения

27.5-1. Сколько сравнений выполняется в сети SORTER$[n]$?

27.5-2. Покажите, что глубина сети SORTER$[n]$ равна $(\lg n)(\lg n + 1)/2$.

27.5-3. Предположим, что всего имеется $2n$ элементов $\langle a_1, a_2, \ldots, a_{2n}\rangle$ и нужно разбить их на две части, в одной из которых содержалось бы n меньших элементов, а в другой — n бóльших. Докажите, что это можно сделать путем увеличения на единицу глубины сети, выполняющей раздельную сортировку последовательностей $\langle a_1, a_2, \ldots, a_n\rangle$ и $\langle a_{n+1}, a_{n+2}, \ldots, a_{2n}\rangle$.

⋆ 27.5-4. Пусть $S(k)$ — глубина сортирующей сети на k входов, а $M(k)$ — глубина объединяющей сети на $2k$ входов. Предположим, что задана последовательность из n чисел, которую нужно отсортировать. Также известно, что каждое число находится в пределах k позиций от правильного положения, которое оно займет после сортировки. Покажите, что n таких чисел можно отсортировать с помощью сети глубиной $S(k) + 2M(k)$.

⋆ 27.5-5. Элементы матрицы $m \times m$ можно отсортировать путем k-кратного повторения описанной ниже процедуры.

1) Сортировка всех нечетных строк в порядке возрастания.

2) Сортировка всех четных строк в порядке убывания.

3) Сортировка всех столбцов в порядке возрастания.

Сколько итераций потребуется выполнить в этой процедуре сортировки? В каком порядке следует считывать матричные элементы после k итераций, чтобы получилась отсортированная выходная последовательность?

Задачи

27-1. **Транспозиционные сортирующие сети**

Сравнивающая сеть является ***транспозиционной*** (transposition network), если каждое сортирующее устройство в ней соединяет соседние линии, как в сети, изображенной на рис. 27.3.

а) Покажите, что любая сортирующая транспозиционная сеть с n входами содержит $\Omega\left(n^2\right)$ сравнивающих устройств.

б) Докажите, что транспозиционная сеть с n входами является сортирующей тогда и только тогда, когда она сортирует последовательность $\langle n, n-1, \ldots, 1\rangle$. (*Указание*: используйте метод математической индукции, аналогично доказательству леммы 27.1.)

Нечетно-четная сортирующая сеть (odd-even sorting network) на n входов $\langle a_1, a_2, \ldots, a_n\rangle$ — это транспозиционная сортирующая сеть с n уровнями сравнивающих устройств, соединенных между собой по схеме “кирпичной кладки” (рис. 27.13). Как видно из рисунка, при $i = = 1, 2, \ldots, n$ и $d = 1, 2, \ldots, n$ линия i соединена сравнивающим устройством на глубине d с линией $j = i + (-1)^{i+d}$, если $1 \leqslant j \leqslant n$.

в) Докажите, что нечетно-четные сортирующие сети действительно выполняют сортировку.

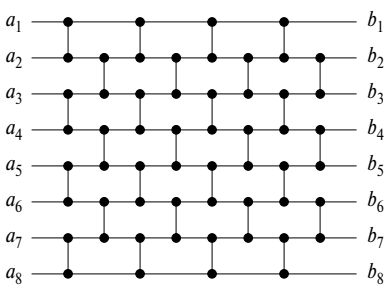


Рис. 27.13. Нечетно-четная сортирующая сеть на 8 входов

27-2. **Нечетно-четная объединяющая сеть Батчера (Batcher)**

В разделе 27.4 было показано, как построить объединяющую сеть на основе битонической сортировки. В этой задаче будет сконструирована ***нечетно-четная объединяющая сеть*** (odd-even merging network). Предполагается, что n равно степени двойки, а задание заключается в том, чтобы объединить отсортированную последовательность, элементы которой подаются на линии $\langle a_1, a_2, \ldots, a_n\rangle$, с отсортированной последовательностью, элементы которой подаются на линии $\langle a_{n+1}, a_{n+2}, \ldots, a_{2n}\rangle$. Если $n = 1$, то сравнивающее устройство размещается между линиями a_1 и a_2. В противном случае по принципу рекурсии составляются две нечетно-четные объединяющие сети, которые работают параллельно. Первая из них объединяет последовательность, которая подается на линии $\langle a_1, a_3, \ldots, a_{n-1}\rangle$, с последовательностью, которая подается на линии $\langle a_{n+1}, a_{n+3}, \ldots, a_{2n-1}\rangle$ (нечетные элементы). Вторая сеть выполняет ту же самую операцию над последовательностями $\langle a_2, a_4, \ldots, a_n\rangle$ и $\langle a_{n+2}, a_{n+4}, \ldots, a_{2n}\rangle$ (четные элементы). Чтобы скомбинировать две отсортированные последовательности, сравнивающие устройства помещаются между линиями a_{2i} и a_{2i+1} при $i = 1, 2, \ldots, n-1$.

а) Изобразите объединяющую сеть на $2n$ входов при $n = 4$.

б) Профессор предложил объединять две отсортированные последовательности, полученные путем рекурсивного объединения, помещая сортирующие устройства не между линиями a_{2i} и a_{2i+1} для $i = 1, 2, \ldots, n-1$, а между линиями a_{2i-1} и a_{2i} для $i = 1, 2, \ldots, n$. Нарисуйте схему такой сети на $2n$ входов при $n = 4$ и приведите контрпример, из которого видно, что профессор ошибается, и получившаяся в результате сеть не является объединяющей. На этом же примере покажите, что объединяющая сеть на $2n$ входов, описанная в части *а*, работает правильно.

в) Докажите с помощью нуль-единичного принципа, что любая нечетно-четная объединяющая сеть на $2n$ входов действительно является объединяющей сетью.

г) Чему равна глубина нечетно-четной объединяющей сети на $2n$ входов? Чему равен ее размер?

27-3. **Перестановочные сети**

Перестановочная сеть (permutation network) на n входов и n выходов содержит переключатели, позволяющие соединять входы сети с ее выходами в соответствии с любой из $n!$ возможных перестановок. На рис. 27.14*а* показана перестановочная сеть P_2 на 2 входа и 2 выхода,

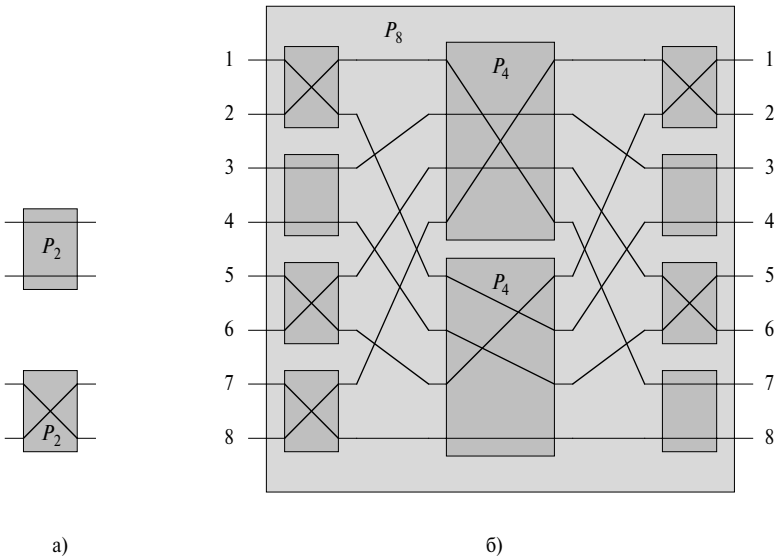


Рис. 27.14. Перестановочная сеть

состоящая из одного переключателя, с помощью которого входы можно соединить с выходами напрямую или крест-накрест.

а) Докажите, что если каждое сравнивающее устройство в сортирующей сети заменить переключателем, показанным на рис. 27.14*а*, то полученная в результате сеть будет перестановочной. Другими словами, для любой перестановки π существует способ установить переключатели в сети таким образом, чтобы соединить вход i с выходом $\pi(i)$.

На рис. 27.14*б* показана рекурсивная схема перестановочной сети P_8 на 8 входов и 8 выходов, которая содержит две копии сети P_4 и 8 переключателей. Переключатели установлены таким образом, чтобы реализовать перестановку $\pi = \langle \pi(1), \pi(2), \ldots, \pi(8) \rangle = \langle 4, 7, 3, 5, 1, 6, 8, 2 \rangle$. При этом требуется (по рекурсии), чтобы в верхней сети P_4 реализовалась перестановка $\langle 4, 2, 3, 1 \rangle$, а в нижней — перестановка $\langle 2, 3, 1, 4 \rangle$.

б) Покажите, как в сети P_8 реализовать перестановку $\langle 5, 3, 4, 6, 1, 8, 2, 7 \rangle$ путем изменения положений переключателей и перестановок, выполняющихся в двух сетях P_4.

Пусть n равно степени двойки. Дайте рекурсивное определение сети P_n в терминах двух сетей $P_{n/2}$, аналогичное определению сети P_8.

в) Опишите алгоритм, который в течение времени $O(n)$ (на обычной машине с произвольным доступом к памяти) позволяет уста-

новить n переключателей, соединенных с входами и выходами сети P_n, и определить перестановки, которые необходимо реализовать в каждой из сетей $P_{n/2}$, чтобы выполнить любую перестановку n элементов. Докажите корректность вашего алгоритма.

г) Чему равны глубина и размер сети P_n? Сколько времени потребуется, чтобы определить на обычном компьютере с произвольным доступом к памяти положения всех переключателей, включая те, которые содержатся в сетях $P_{n/2}$?

д) Докажите, что при $n > 2$ любая перестановочная сеть, а не только P_n, должна реализовать некоторую перестановку путем установки двух различных сочетаний положений переключателей.

Заключительные замечания

В книге Кнута (Knuth) [185] обсуждаются сортирующие сети и приводится их краткая история. Однозначно можно сказать, что впервые они исследовались в 1954 году Армстронгом (P.N. Armstrong), Нельсоном (R.J. Nelson) и О'Коннором (D.J. O'Connor). В начале 1960-х Батчер (K.E. Batcher) разработал первую сеть, способную объединять две последовательности из n чисел в течение времени $O(\lg n)$. Он воспользовался нечетно-четным объединением (см. задачу 27-2), а также показал, как с помощью этого метода выполнить сортировку n чисел за время $O\left(\lg^2 n\right)$. Вскоре после этого он разработал битонический сортировщик глубиной $O(\lg n)$, аналогичный тому, который представлен на рис. 27.3. Авторство нуль-единичного принципа Кнут приписывает Бурайсиусу (W.G. Bouricius) (1954 г.), доказавшему его в контексте деревьев решений.

В течение долгого времени открытым оставался вопрос о существовании сортирующих сетей глубиной $O(\lg n)$. В 1983 году на него удалось ответить утвердительно, но оказалось, что этот ответ нельзя считать удовлетворительным. Сортирующая сеть AKS (названная так в честь своих разработчиков Ajtai (Айтаи), Комлёса (Komlos) и Семереди (Szemeredi) [11]) имеет глубину $O(\lg n)$ и сортирует n чисел с помощью $O(n \lg n)$ сравнений. К сожалению, константы, скрытые в O-обозначении, получаются слишком большими (порядка нескольких тысяч), поэтому считается, что такая сеть не имеет практической пользы.

ГЛАВА 28

Работа с матрицами

Работа с матрицами — сердце научных расчетов, поэтому эффективные алгоритмы для работы с матрицами представляют значительный практический интерес. Эта глава представляет собой краткое введение в теорию матриц и операции над матрицами, делая особый упор на задачу умножения матриц и решение систем линейных уравнений.

После раздела 28.1, который знакомит нас с основными концепциями теории матриц и используемыми обозначениями, в разделе 28.2 представлен алгоритм Штрассена, позволяющий выполнить умножение двух матриц размером $n \times n$ за время $\Theta\left(n^{\lg 7}\right) = O\left(n^{2.81}\right)$. В разделе 28.3 показано, как решать системы линейных уравнений с использованием LUP-разложения. Затем в разделе 28.4 показывается тесная связь задач умножения и обращения матриц. И наконец, в разделе 28.5 рассматривается важный класс симметричных положительно определенных матриц и их применение для поиска решения переопределенных систем линейных уравнений методом наименьших квадратов.

Один из важнейших вопросов, возникающих на практике, — ***численная устойчивость*** (numerical stability). Из-за ограниченной точности представления действительных чисел в реальном компьютере в процессе вычислений могут резко нарастать ошибки округления, что приводит к неверным результатам. Такие вычисления являются численно неустойчивыми. Несмотря на важность данного вопроса, мы лишь поверхностно коснемся его в данной главе, так что мы рекомендуем читателям обратиться к отличной книге Голуба (Golub) и Ван Лоана (Van Loan) [125], в которой детально рассматриваются вопросы численной устойчивости.

28.1 Свойства матриц

В этом разделе мы рассмотрим некоторые базовые концепции теории матриц и их фундаментальные свойства, обращая особое внимание на те из них, которые понадобятся нам в следующих разделах.

Матрицы и векторы

Матрица (matrix) представляет собой прямоугольный массив чисел. Например,

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \tag{28.1}$$

является матрицей размера 2×3 $A = (a_{ij})$, где $i = 1, 2$ и $j = 1, 2, 3$. Элемент на пересечении i-й строки и j-го столбца матрицы — a_{ij}. Мы используем заглавные буквы для обозначения матриц, а их элементы обозначаются соответствующими строчными буквами с нижними индексами. Множество всех матриц размером $m \times n$, элементами которых являются действительные числа, обозначается как $\mathbf{R}^{m \times n}$. В общем случае множество матриц размером $m \times n$, элементы которых принадлежат множеству S, обозначается как $S^{m \times n}$.

Транспонированная (transpose) матрица A^{T} получается из матрицы A путем обмена местами ее строк и столбцов. Так, для матрицы A из (28.1)

$$A^{\mathrm{T}} = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Вектор (vector) представляет собой одномерный массив чисел. Например,

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} \tag{28.2}$$

является вектором размером 3. Для обозначения векторов мы используем строчные буквы и обозначаем i-й элемент вектора x как x_i. Стандартной формой вектора будем считать ***вектор-столбец*** (column vector), эквивалентный матрице $n \times 1$. Соответствующий ***вектор-строка*** (row vector) получается путем транспонирования вектора-столбца:

$$x^{\mathrm{T}} = \begin{pmatrix} 2 & 3 & 5 \end{pmatrix}$$

Единичным вектором (unit vector) e_i называется вектор, i-й элемент которого равен 1, а все остальные элементы равны 0. Обычно размер единичного вектора ясен из контекста.

Нулевая матрица (zero matrix) — это матрица, все элементы которой равны 0. Такая матрица часто записывается просто как 0, поскольку неоднозначность между числом 0 и нулевой матрицей легко разрешается при помощи контекста. Если размер нулевой матрицы не указан, то он также выводится из контекста.

Часто приходится иметь дело с ***квадратными*** (square) матрицами размером $n \times n$. Некоторые из квадратных матриц представляют особый интерес.

1. ***Диагональная матрица*** (diagonal matrix) обладает тем свойством, что $a_{ij} = 0$ при $i \neq j$. Поскольку все недиагональные элементы такой матрицы равны 0, диагональную матрицу можно определить путем перечисления ее элементов вдоль диагонали:

$$\mathrm{diag}(a_{11}, a_{22}, \ldots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \ldots & 0 \\ 0 & a_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & a_{nn} \end{pmatrix} .$$

2. ***Единичная матрица*** (identity matrix) I_n размером $n \times n$ представляет собой диагональную матрицу, все диагональные элементы которой равны 1:

$$I_n = \mathrm{diag}(1, 1, \ldots, 1) = \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{pmatrix} .$$

   Если используется обозначение I без индекса, размер единичной матрицы определяется из контекста. Заметим, что i-м столбцом единичной матрицы является единичный вектор e_i.

3. Элементы ***трехдиагональной матрицы*** (tridiagonal matrix) T обладают тем свойством, что если $|i - j| > 1$, то $t_{ij} = 0$. Ненулевые элементы такой матрицы располагаются на главной диагонали, непосредственно над ней ($t_{i,i+1}$ для $i = 1, 2, \ldots, n - 1$) и непосредственно под ней ($t_{i+1,i}$ для $i = 1, 2, \ldots, n - 1$):

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \ldots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \ldots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \ldots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \ldots & 0 & t_{n,n-1} & t_{nn} \end{pmatrix} .$$

4. ***Верхне-треугольной матрицей*** (upper-triangular matrix) U называется матрица, у которой все элементы ниже диагонали равны 0 ($u_{ij} = 0$ при $i > j$):

$$U = \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{pmatrix}.$$

   Верхне-треугольная матрица является ***единичной верхне-треугольной матрицей*** (unit upper-triangular), если все ее диагональные элементы равны 1.

5. ***Нижне-треугольной матрицей*** (lower-triangular matrix) L называется матрица, у которой все элементы выше диагонали равны 0 ($l_{ij} = 0$ при $i < j$):

$$L = \begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix}.$$

   Нижне-треугольная матрица является ***единичной нижне-треугольной матрицей*** (unit lower-triangular), если все ее диагональные элементы равны 1.

6. ***Матрица перестановки*** (permutation matrix) P имеет в каждой строке и столбце ровно по одной единице, а на всех прочих местах располагаются нули. Примером матрицы перестановки может служить матрица

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

   Такая матрица называется матрицей перестановки, потому что умножение вектора x на матрицу перестановки приводит к перестановке элементов вектора.

7. ***Симметричная матрица*** (symmetric matrix) A удовлетворяет условию $A = A^{\mathrm{T}}$. Например, матрица

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

   является симметричной.

Операции над матрицами

Элементами матриц и векторов служат элементы некоторой числовой системы, такие как действительные числа, комплексные числа или, например, целые числа по модулю простого числа. Числовая система определяет, каким образом должны складываться и перемножаться числа. Эти определения можно распространить и на матрицы.

Определим ***сложение матриц*** (matrix addition) следующим образом. Если $A = (a_{ij})$ и $B = (b_{ij})$ — матрицы размером $m \times n$, то их суммой является матрица $C = (c_{ij}) = A + B$ размером $m \times n$, определяемая соотношением $c_{ij} = a_{ij} + b_{ij}$ для $i = 1, 2, \ldots, m$ и $j = 1, 2, \ldots, n$. Другими словами, сложение матриц выполняется поэлементно. Нулевая матрица нейтральна по отношению к сложению матриц:

$$A + 0 = A = 0 + A.$$

Если λ — число, а $A = (a_{ij})$ — матрица, то соотношение $\lambda A = (\lambda a_{ij})$ определяет ***скалярное произведение*** (scalar multiple) матрицы на число, которое также выполняется поэлементно. Частным случаем скалярного произведения является умножение на -1, которое дает ***противоположную*** (negative) матрицу $-1 \cdot A = -A$, обладающую тем свойством, что

$$A + (-A) = 0 = (-A) + A.$$

Соответственно, можно определить ***вычитание*** матриц (matrix subtraction) как сложение с противоположной матрицей: $A - B = A + (-B)$.

Матричное умножение (matrix multiplication) определяется следующим образом. Матрицы A и B могут быть перемножены, если они ***совместимы*** (compatible) в том смысле, что число столбцов A равно числу строк B (в общем случае выражение, содержащее матричное произведение AB, всегда подразумевает совместимость матриц A и B). Если $A = (a_{ij})$ — матрица размером $m \times n$, а $B = (b_{ij})$ — матрица размером $n \times p$, то их произведение $C = AB$ представляет собой матрицу $C = (c_{ij})$ размером $m \times p$, элементы которой определяются уравнением

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} \tag{28.3}$$

для $i = 1, 2, \ldots, m$ и $k = 1, 2, \ldots, p$. Процедура MATRIX_MULTIPLY из раздела 25.1 реализует матричное умножение квадратных матриц ($m = n = p$) непосредственно по формуле (28.3). Для умножения матриц размером $n \times n$ процедура MATRIX_MULTIPLY выполняет n^3 умножений и $n^2(n-1)$ сложений, так что время ее работы равно $\Theta(n^3)$.

Матрицы обладают многими (но не всеми) алгебраическими свойствами, присущими обычным числам. Единичная матрица является нейтральным элементом

по отношению к умножению:

$$I_m A = A I_n = A$$

для любой матрицы A размером $m \times n$. Умножение на нулевую матрицу дает нулевую матрицу:

$$A\,0 = 0\,A = 0.$$

Умножение матриц ассоциативно:

$$A\,(BC) = (AB)\,C \tag{28.4}$$

для любых совместимых матриц A, B и C. Умножение матриц дистрибутивно относительно сложения:

$$\begin{aligned} A\,(B + C) &= AB + AC, \\ (B + C)\,D &= BD + CD. \end{aligned} \tag{28.5}$$

Для $n > 1$ умножение матриц размером $n \times n$ не коммутативно. Например, если $A = \left(\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right)$ и $B = \left(\begin{smallmatrix} 0 & 0 \\ 1 & 0 \end{smallmatrix}\right)$, то $AB = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$, но $BA = \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right)$.

Произведения матрицы и вектора иди двух векторов вычисляются путем представления вектора как матрицы размером $n \times 1$ (или $1 \times n$ в случае вектора-строки). Таким образом, если A — матрица размером $m \times n$, а x — вектор размером n, то Ax является вектором размера m. Если x и y — векторы размера n, то произведение

$$x^{\mathrm{T}} y = \sum_{i=1}^{n} x_i y_i$$

представляет собой число (в действительности — матрицу размером 1×1), называющееся ***скалярным произведением*** (inner product) векторов x и y. Матрица размером $n \times n$ $Z = xy^{\mathrm{T}}$ с элементами $z_{ij} = x_i y_j$ называется ***тензорным произведением*** (outer product) этих же векторов. (***Евклидова***) ***норма*** ((euclidean) norm) $\|x\|$ вектора x размером n определяется соотношением

$$\|x\| = \sqrt{x_1^2 + x_2^2 + \cdots x_n^2} = \sqrt{x^{\mathrm{T}} x},$$

т.е. норма x — это длина вектора в n-мерном евклидовом пространстве.

Обратные матрицы, ранги и детерминанты

Матрицей, ***обратной*** (inverse) к данной матрице A размером $n \times n$, является матрица размером $n \times n$, обозначаемая как A^{-1} (если таковая существует), такая что $A\,A^{-1} = I_n = A^{-1}\,A$. Например,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}.$$

Многие ненулевые квадратные матрицы не имеют обратных матриц. Матрица, для которой не существует обратная матрица, называется ***необращаемой*** (noninvertible), или ***вырожденной*** (singular). Вот пример ненулевой вырожденной матрицы:

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}.$$

Если матрица имеет обратную матрицу, она называется ***обращаемой*** (invertible), или ***невырожденной*** (nonsingular). Если обратная матрица существует, то она единственная (см. упражнение 28.1-3). Если A и B — невырожденные матрицы размером $n \times n$, то

$$(AB)^{-1} = A^{-1}B^{-1}. \tag{28.6}$$

Операция обращения коммутативна с операцией транспонирования:

$$\left(A^{-1}\right)^{\mathrm{T}} = \left(A^{\mathrm{T}}\right)^{-1}.$$

Векторы $x_1, x_2, \ldots x_n$ ***линейно зависимы*** (linearly dependent), если существуют коэффициенты $c_1, c_2, \ldots, c_n$, среди которых есть ненулевые, такие что $c_1x_1 + + c_2x_2 + \cdots + c_nx_n = 0$. Например, векторы $x_1 = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$, $x_2 = \begin{pmatrix} 2 & 6 & 4 \end{pmatrix}$ и $x_3 = \begin{pmatrix} 4 & 11 & 9 \end{pmatrix}$ линейно зависимы, поскольку $2x_1 + 3x_2 - 2x_3 = 0$. Если векторы не являются линейно зависимыми, они называются ***линейно независимыми*** (linearly independent). Например, столбцы единичной матрицы линейно независимы.

Столбцовым рангом (column rank) ненулевой матрицы A размером $m \times n$ называется размер наибольшего множества линейно независимых столбцов A. Аналогично, ***строчным рангом*** (row rank) ненулевой матрицы A размером $m \times n$ называется размер наибольшего множества линейно независимых строк A. Фундаментальным свойством любой матрицы A является равенство ее строчного и столбцового рангов, так что мы можем говорить просто о ***ранге*** (rank) матрицы. Ранг матрицы размером $m \times n$ представляет собой целое число от 0 до $\min(m, n)$ включительно (ранг нулевой матрицы равен 0, ранг единичной матрицы размером $n \times n$ равен n). Другое эквивалентное (и зачастую более полезное) определение ранга ненулевой матрицы A размером $m \times n$ — это наименьшее число r такое, что существуют матрицы B и C размером соответственно $m \times r$ и $r \times n$ такие, что $A = BC$.

Квадратная матрица размером $n \times n$ имеет ***полный ранг*** (full rank), если ее ранг равен n. Матрица размером $m \times n$ имеет ***полный столбцовый ранг*** (full column rank), если ее ранг равен n. Фундаментальное свойство рангов приведено в следующей теореме.

Теорема 28.1. Квадратная матрица имеет полный ранг тогда и только тогда, когда она является невырожденной. ■

Ненулевой вектор x, такой что $Ax = 0$, называется ***аннулирующим вектором*** (null vector) матрицы A. Приведенная далее теорема (доказательство которой оставлено в качестве упражнения 28.1-9) и ее следствие связывают столбцовый ранг и вырожденность с аннулирующим вектором.

Теорема 28.2. Матрица A имеет полный столбцовый ранг тогда и только тогда, когда для нее не существует аннулирующий вектор. ■

Следствие 28.3. Квадратная матрица A является вырожденной тогда и только тогда, когда она имеет аннулирующий вектор. ■

Минором (minor) элемента a_{ij} (ij-минор) матрицы A размером $n \times n$ ($n > 1$) называется матрица $A_{[ij]}$ размером $(n-1) \times (n-1)$,получаемая из A удалением i-й строки и j-го столбца. ***Определитель***, или ***детерминант*** (determinant) матрицы A размером $n \times n$ можно определить рекурсивно при помощи миноров следующим образом:

$$\det(A) = \begin{cases} a_{11} & \text{при } n = 1, \\ \sum_{j=1}^{n} (-1)^{1+j} a_{1j} \det\left(A_{[1j]}\right) & \text{при } n > 1. \end{cases} \tag{28.7}$$

Множитель $(-1)^{i+j} \det\left(A_{[ij]}\right)$ называется ***алгебраическим дополнением*** (cofactor) элемента a_{ij}.

В приведенных ниже теоремах, доказательство которых здесь опущено, описывают фундаментальные свойства определителей.

Теорема 28.4 (Свойства определителя). Определитель квадратной матрицы A обладает следующими свойствами.

- Если любая строка или любой столбец A нулевой, то $\det(A) = 0$.
- Если все элементы одного произвольного столбца (или строки) матрицы умножаются на λ, то ее определитель также умножается на λ.
- Определитель матрицы A остается неизменным, если все элементы одной строки (или столбца) прибавить к элементам другой строки (столбца).
- Определитель матрицы A равен определителю транспонированной матрицы A^{T}.
- Определитель матрицы A умножается на -1, если обменять местами любые два ее столбца (или строки).

Кроме того, для любых квадратных матриц A и B $\det(AB) = \det(A)\det(B)$. ■

Теорема 28.5. Квадратная матрица A вырождена тогда и только тогда, когда $\det(A) = 0$. ■

Положительно определенные матрицы

Во многих приложениях важную роль играют положительно определенные матрицы. Матрица A размером $n \times n$ является ***положительно определенной*** (positive-definite), если $x^{\mathrm{T}}Ax > 0$ для любого ненулевого вектора x размера n. Например, единичная матрица положительно определенная, поскольку для произвольного ненулевого вектора $x = \begin{pmatrix} x_1 & x_2 & \dots & x_n \end{pmatrix}^{\mathrm{T}}$

$$x^{\mathrm{T}} I_n x = x^{\mathrm{T}} x = \sum_{i=1}^{n} x_i^2 > 0.$$

Как мы увидим, зачастую встречающиеся в приложениях матрицы положительно определены в силу следующей теоремы.

Теорема 28.6. Для произвольной матрицы A с полным столбцовым рангом матрица $A^{\mathrm{T}}A$ положительно определена.

Доказательство. Мы должны показать, что $x^{\mathrm{T}}\left(A^{\mathrm{T}}A\right)x > 0$ для любого ненулевого вектора x. Для произвольного вектора x

$$x^{\mathrm{T}}\left(A^{\mathrm{T}}A\right)x = (Ax)^{\mathrm{T}}(Ax) = \|Ax\|^2,$$

где первое равенство следует из упражнения 28.1-2. Заметим, что $\|Ax\|^2$ представляет собой просто сумму квадратов элементов вектора Ax. Таким образом, $\|Ax\|^2 \geqslant 0$. Если $\|Ax\|^2 = 0$, все элементы вектора Ax равны 0, т.е. $Ax = 0$. Поскольку A — матрица с полным столбцовым рангом, из $Ax = 0$ согласно теореме 28.2 следует, что $x = 0$. Следовательно, матрица $A^{\mathrm{T}}A$ положительно определенная. ■

Прочие свойства положительно определенных матриц рассматриваются в разделе 28.5.

Упражнения

28.1-1. Покажите, что если A и B — симметричные матрицы размером $n \times n$, то симметричными будут их сумма и разность.

28.1-2. Докажите, что $(AB)^{\mathrm{T}} = B^{\mathrm{T}}A^{\mathrm{T}}$ и что $A^{\mathrm{T}}A$ всегда является симметричной матрицей.

28.1-3. Докажите единственность обратной матрицы, т.е. что если B и C — обратные к A матрицы, то $B = C$.

28.1-4. Докажите, что произведение двух нижне-треугольных матриц дает нижне-треугольную матрицу. Докажите, что определитель нижне-треугольной или верхне-треугольной матрицы равен произведению диагональных элементов. Докажите, что если существует матрица, обратная к нижне-треугольной матрице, то она также является нижне-треугольной.

28.1-5. Докажите, что если P — матрица перестановок размером $n \times n$, а A — матрица размером $n \times n$, то PA можно получить из A путем перестановки ее строк, а AP — путем перестановки столбцов A. Докажите, что произведение двух матриц перестановки является матрицей перестановки. Докажите, что если P — матрица перестановки, то P обращаема, причем обратной к P является матрица P^{T}, которая также является матрицей перестановки.

28.1-6. Пусть A и B — матрицы размером $n \times n$ такие, что $AB = 1$. Докажите, что если A' получена из A путем прибавления к строке i строки j, то обратную к A' матрицу B' можно получить, вычитая в матрице B столбец i из столбца j.

28.1-7. Пусть A — невырожденная матрица размером $n \times n$ с комплексными элементами. Покажите, что все элементы A^{-1} вещественны тогда и только тогда, когда вещественны все элементы A.

28.1-8. Покажите, что если A — невырожденная симметричная матрица размером $n \times n$, то матрица A^{-1} симметрична. Покажите, что если B — произвольная матрица размером $m \times n$, то матрица размером $m \times m$, получающаяся в результате умножения BAB^{T}, симметрична.

28.1-9. Докажите теорему 28.2, т.е. покажите, что матрица A имеет полный столбцовый ранг тогда и только тогда, когда из $Ax = 0$ следует $x = 0$. (*Указание*: запишите условие линейной зависимости одного столбца от остальных в виде матрично-векторного уравнения.)

28.1-10. Докажите, что для любых двух совместимых матриц A и B $rank\,(A\,B) \leqslant$ $\leqslant \min\left(rank\,(A), rank\,(B)\right)$, причем равенство достигается если либо A, либо B — невырожденная квадратная матрица. (*Указание*: воспользуйтесь альтернативным определением ранга матрицы.)

28.1-11. Докажите, что определитель ***матрицы Вандермонда*** (Vandermonde matrix)

$$V(x_0, x_1, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}$$

равен

$$\det(V(x_0, x_1, \ldots, x_{n-1})) = \prod_{0 \le j < k \le n-1} (x_k - x_j) \ .$$

(*Указание*: умножьте столбец i на $-x_0$ и прибавьте к столбцу $i+1$ для $i = n-1, n-2, \ldots, 1$, а затем примените индукцию.)

28.2 Алгоритм умножения матриц Штрассена

В этом разделе представлен замечательный рекурсивный алгоритм умножения матриц размера $n \times n$, разработанный Штрассеном (Strassen). Время его работы равно $\Theta\left(n^{\lg 7}\right) = O\left(n^{2.81}\right)$. При достаточно больших значениях n этот алгоритм работает быстрее алгоритма MATRIX_MULTIPLY из раздела 25.1, время работы которого — $\Theta\left(n^3\right)$.

Обзор алгоритма

Алгоритм Штрассена можно рассматривать как применение уже знакомой нам технологии декомпозиции "разделяй и властвуй". Предположим, что мы хотим вычислить произведение $C = AB$, где каждая из матриц имеет размер $n \times n$. Считая, что n является точной степенью 2, поделим каждую из матриц на четыре матрицы размером $n/2 \times n/2$ и перепишем уравнение $C = AB$ следующим образом:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} \ . \tag{28.8}$$

(В упражнении 28.2-2 рассматривается ситуация, когда n не является точной степенью 2.) Уравнение (28.8) соответствует четырем уравнениям:

$$r = ae + bg, \tag{28.9}$$
$$s = af + bh, \tag{28.10}$$
$$t = ce + dg, \tag{28.11}$$
$$u = cf + dh. \tag{28.12}$$

Каждое из этих четырех уравнений требует двух умножений матриц размера $n/2 \times$ $\times\ n/2$ и сложения получившихся произведений. Используя эти уравнения как определение простейшего рекурсивного алгоритма, мы получим следующее рекуррентное соотношение для времени его работы при умножении матриц размера $n \times n$:

$$T(n) = 8T(n/2) + \Theta\left(n^2\right) . \tag{28.13}$$

К сожалению, рекуррентное соотношение (28.13) имеет решение $T(n) = \Theta\left(n^3\right)$, так что этот метод получается не быстрее, чем обычное стандартное умножение матриц.

Штрассен разработал рекурсивный алгоритм, который требует только 7 умножений матриц размером $n/2 \times n/2$ и $\Theta\left(n^2\right)$ скалярных сложений и умножений, что приводит к рекуррентному соотношению

$$T(n) = 7T(n/2) + \Theta\left(n^2\right), \tag{28.14}$$

которое имеет решение $T(n) = \Theta\left(n^{\lg 7}\right) = O\left(n^{2.81}\right)$.

Метод Штрассена состоит из четырех шагов.

1. Разделяем матрицы A и B на подматрицы размером $n/2 \times n/2$, как показано в 28.8.
2. Используя $\Theta\left(n^2\right)$ скалярных сложений им умножений, вычисляем 14 матриц $A_1, B_1, A_2, B_2, \ldots, A_7, B_7$, каждая из которых имеет размер $n/2 \times n/2$.
3. Рекурсивно вычисляем семь матричных произведений $P_i = A_i B_i$ для $i = = 1, 2, \ldots, 7$.
4. Вычисляем подматрицы r, s, t и u результирующей матрицы C путем сложения и/или вычитания различных комбинаций матриц P_i с использованием $\Theta\left(n^2\right)$ скалярных сложений и вычитаний.

Описанная процедура удовлетворяет рекуррентному соотношению (28.14). Все, что осталось сделать, — изложить опущенные детали.

Определение произведений подматриц

Не совсем понятно, как именно Штрассен нашел необходимые подматрицы, которые следует перемножать. Давайте попытаемся воспроизвести один из возможных способов разработки алгоритма.

Предположим, что каждое матричное произведение P_i можно записать в виде

$$P_i = A_i B_i = (\alpha_{1i} a + \alpha_{i2} b + \alpha_{i3} c + \alpha_{i4} d) \cdot (\beta_{i1} e + \beta_{i2} f + \beta_{i3} g + \beta_{i4} h), \tag{28.15}$$

где коэффициенты α_{ij} и β_{ij} принимают значения из множества $\{-1, 0, 1\}$. Т.е. мы предполагаем, что каждое произведение вычисляется при помощи сложения или вычитания некоторых из подматриц A, сложения или вычитания некоторых из подматриц B, и перемножения получающихся результатов. Хотя, естественно, можно искать более общий вид произведения P_i, описанного вида произведения оказывается достаточно для разработки алгоритма.

Если использовать описанный вид произведения, то мы можем использовать его рекурсивно без оглядки на некоммутативность умножения матриц, так как

в каждом произведении все подматрицы A оказываются слева, а подматрицы B — справа.

Для удобства представления линейных комбинаций произведений подматриц воспользуемся матрицами размером 4×4. Например, мы можем записать уравнение (28.9) как

$$r = ae + bg = \\ = \begin{pmatrix} a & b & c & d \end{pmatrix} \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = \\ = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array}.$$

Последнее выражение использует сокращенную запись, где каждый "+" представляет $+1$, "$\cdot$" представляет 0, а "$-$" — -1. Кроме того, далее мы опускаем метки строк и столбцов. Используя данные обозначения, мы получим следующие уравнения для остальных подматриц, составляющих результирующую матрицу C:

$$s = af + bh = \begin{pmatrix} \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

$$t = ce + dg = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \end{pmatrix},$$

$$u = cf + dh = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.$$

Начнем наш поиск алгоритма быстрого перемножения матриц со следующего наблюдения: подматрицу s можно вычислить как $s = P_1 + P_2$, где матрицы P_1 и P_2 вычисляются с использованием одного матричного произведения каждая:

$$P_1 = A_1 B_1 = a \cdot (f - h) = af - ah = \begin{pmatrix} \cdot & + & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

$$P_2 = A_2 B_2 = (a + b) \cdot h = ah + bh = \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Матрица t может быть вычислена как $t = P_3 + P_4$, где

$$P_3 = A_3 B_3 = (c + d) \cdot e = ce + de = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}$$

и

$$P_4 = A_4 B_4 = d \cdot (g - e) = dg - de = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & + & \cdot \end{pmatrix}.$$

Определим ***существенное слагаемое*** (essential term) как один из восьми членов, появляющихся в правой части уравнений (28.9)–(28.12). Мы использовали 4 произведения для вычисления двух подматриц s и t, существенными слагаемыми которых являются af, bh, ce и dg. Заметим, что P_1 вычисляет существенное слагаемое af, P_2 вычисляет существенное слагаемое bh, P_3 вычисляет существенное слагаемое ce, а P_4 вычисляет существенное слагаемое dg. Таким образом, нам остается вычислить две оставшиеся подматрицы r и u, чьими существенными слагаемыми являются ae, bg, cf и dh, с использованием не более чем трех дополнительных произведений. Попробуем вычислить P_5 по-новому, чтобы охватить два существенных слагаемых одновременно:

$$P_5 = A_5 B_5 = (a + d) \cdot (e + h) = ae + ah + de + dh = \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix}.$$

В дополнение к существенным слагаемым ae и dh P_5 вычисляет несущественные слагаемые ah и de, которые требуется каким-то образом сократить. Мы можем воспользоваться для этого произведениями P_2 и P_4, но при этом появятся новые несущественные слагаемые:

$$P_5 + P_4 - P_2 = ae + dh + dg - bh = \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \end{pmatrix}.$$

Прибавляя дополнительное произведение

$$P_6 = A_6 B_6 = (b-d)\cdot(g+h) = bg + bh - dg - dh = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & - & - \end{pmatrix},$$

мы получим:

$$r = P_5 + P_4 - P_2 + P_6 = ae + bg = \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Мы можем получить u аналогичным способом из P_5 с использованием P_1 и P_3 для удаления несущественных слагаемых из P_5:

$$P_5 + P_1 - P_3 = ae + af - ce + dh = \begin{pmatrix} + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.$$

Вычитая дополнительное произведение

$$P_7 = A_7 B_7 = (a-c)\cdot(e+f) = ae + af - ce - cf = \begin{pmatrix} + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & - & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

мы получим

$$u = P_5 + P_1 - P_3 - P_7 = cf + dh = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.$$

Таким образом, для вычисления произведения $C = AB$ можно использовать 7 произведений подматриц $P_1, P_2, \ldots, P_7$, что и завершает описание метода Штрассена.

Обсуждение метода

Алгоритм Штрассена редко применяется на практике по следующим причинам.

1. Постоянный множитель, скрытый во времени работы алгоритма Штрассена, превышает постоянный множитель во времени работы $\Theta\left(n^3\right)$ простого алгоритма умножения.
2. В случае разреженных матриц имеются специализированные более эффективные методы умножения.
3. Алгоритм Штрассена не настолько численно устойчив, как простой алгоритм умножения матриц.
4. Построение подматриц на каждом шаге рекурсии приводит к повышенному расходу памяти.

Последние две причины потеряли актуальность в 1990-х годах. Хайем (Higham) [145] показал, что отличия в численной устойчивости преувеличены; хотя алгоритм Штрассена для ряда приложений действительно слишком численно неустойчив, его устойчивости все же вполне достаточно для большого количества приложений. Бейли (Bailey) и др. в работе [30] рассмотрели методы снижения требующейся для работы алгоритма Штрассена памяти.

На практике реализации быстрого умножения плотных матриц с использованием алгоритма Штрассена имеют “точку пересечения” (в смысле размера перемножаемых матриц) с обычным алгоритмом умножения, и переключаются на использование простого алгоритма при перемножении матриц с размером, меньшим точки пересечения. Точное значение точки пересечения сильно зависит от реализации и используемой вычислительной системы. Анализ, учитывающий количество операций и игнорирующий кэширование и конвейерную обработку, дает для точки пересечения различные оценки — 8 у Хайема [145] и 12 у Хасс-Ледермана (Huss-Lederman) [163]. Эмпирические измерения обычно приводят к более высокой точке пересечения, обычно порядка 20 и выше. В каждой конкретной системе поиск точки пересечения лучше выполнять экспериментально.

Использование более сложных методов, изложение которых выходит за рамки книги, позволяет вычислять произведение матриц еще быстрее. В настоящий момент наилучшая верхняя граница составляет примерно $O\left(n^{2.376}\right)$. Наилучшая нижняя граница, очевидно, равна $\Omega\left(n^2\right)$ (ее очевидность вытекает из необходимости заполнения n^2 элементов матрицы), так что разрыв достаточно велик и точная граница в настоящий момент неизвестна.

Упражнения

28.2-1. Воспользуйтесь алгоритмом Штрассена для вычисления произведения $\left(\begin{smallmatrix}1 & 3\\ 5 & 7\end{smallmatrix}\right)\left(\begin{smallmatrix}8 & 4\\ 6 & 2\end{smallmatrix}\right)$. Приведите промежуточные результаты.

28.2-2. Каким образом следует модифицировать алгоритм Штрассена для умножения матриц размером $n \times n$, если n не является точной степенью 2? Покажите, что модифицированный алгоритм имеет время работы $O\left(n^{\lg 7}\right)$.

28.2-3. Допустим, что мы можем перемножить матрицы размером 3×3 с использованием k умножений (не используя коммутативности умножения). Чему равно наибольшее значение k, позволяющее перемножить матрицы размером $n \times n$ за время $o\left(n^{\lg 7}\right)$? Чему равно время работы алгоритма в этом случае?

28.2-4. Имеется способ перемножения матриц размером 68×68 при помощи 132 464 умножений чисел, матриц размером 70×70 за 143 640 умножений чисел и матриц размером 72×72 за 155 424 умножения. Какой метод дает лучшее асимптотическое время работы при использовании в алгоритме умножения матриц "разделяй и властвуй"? Как это время работы соотносится со временем работы алгоритма Штрассена?

28.2-5. Как быстро можно умножить матрицу размером $kn \times n$ на матрицу размером $n \times kn$, используя алгоритм Штрассена в качестве подпрограммы? А в случае перемножения матриц в обратном порядке?

28.2-6. Покажите, как можно перемножить два комплексных числа $a + bi$ и $c + di$, используя только три действительных умножения. Алгоритм должен получать в качестве входных параметров числа a, b, c и d, и давать на выходе действительную $(ac - bd)$ и мнимую $(ad + bc)$ части произведения.

28.3 Решение систем линейных уравнений

Решение систем линейных уравнений представляет собой фундаментальную задачу, возникающую в различных приложениях. Такую систему линейных уравнений можно записать как матричное уравнение, в котором каждый элемент матрицы или вектора принадлежит некоторому полю, обычно — полю действительных

чисел **R**. В этом разделе мы рассмотрим решение систем линейных уравнений с использованием метода, называемого LUP-разложением.

Начнем с системы линейных уравнений с n неизвестными $x_1, x_2, \ldots, x_n$.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 , \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 , \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n . \end{aligned} \tag{28.16}$$

Множество значений $x_1, x_2, \ldots, x_n$, одновременно удовлетворяющее всем уравнениям в (28.16), называется ***решением*** (solution) этих уравнений. В данном разделе мы рассмотрим только случай, когда у нас имеется ровно n уравнений для n неизвестных.

Мы можем для удобства переписать уравнения (28.16) в виде матрично-векторного уравнения

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

или, вводя обозначения $A = (a_{ij})$, $x = (x_i)$ и $b = (b_i)$, в виде

$$Ax = b. \tag{28.17}$$

Если матрица A невырождена, то можно найти обратную к ней матрицу A^{-1} и найти решение системы линейных уравнений следующим образом:

$$x = A^{-1}b. \tag{28.18}$$

Можно легко доказать, что x является единственным решением уравнения (28.17). Пусть имеется два решения — x и x'. Тогда $Ax = Ax' = b$ и

$$x = \left(A^{-1}A\right)x = A^{-1}\left(A\,x\right) = A^{-1}\left(A\,x'\right) = \left(A^{-1}A\right)x' = x'.$$

В этом разделе мы будем иметь дело в основном с системами линейных уравнений, матрица A которых невырождена, или, что в соответствии с теоремой 28.1 то же, ранг матрицы A равен количеству неизвестных в системе линейных уравнений n. Имеются и другие ситуации, с которыми мы вкратце познакомимся. Если количество уравнений меньше количества неизвестных (или, говоря более общенно, ранг матрицы A меньше n), такая система линейных уравнений называется

недоопределенной (underdetermined). Недоопределенные системы линейных уравнений обычно имеют бесконечно много решений, хотя решение вообще может не существовать, если уравнения несовместимы друг с другом. Если же количество уравнений превышает количество неизвестных, такая система линейных уравнений называется ***переопределенной*** (overdetermined), и может не иметь решений. Поиск хорошего приближенного решения переопределенных систем линейных уравнений представляет собой важную задачу, которая рассматривается в разделе 28.5.

Вернемся к нашей задаче решения системы линейных уравнений $A\,x = b$ из n уравнений с n неизвестными. Одним из путей решения системы является поиск обратной матрицы A^{-1} и умножение на нее обеих частей уравнения, что даст нам $A^{-1}A\,x = A^{-1}b$, или $x = A^{-1}b$. Такой подход на практике отвергается в первую очередь из-за его численной неустойчивости. К счастью, другой способ решения — LUP-разложение — численно устойчив и быстрее работает.

Обзор LUP-разложения

Идея, лежащая в основе LUP-разложения, состоит в поиске трех матриц L, U и P размером $n \times n$, таких что

$$P\,A = L\,U, \tag{28.19}$$

где

- L — единичная нижне-треугольная матрица,
- U — верхне-треугольная матрица,
- P — матрица перестановки.

Матрицы L, U и P, удовлетворяющие уравнению (28.19), называются ***LUP-разложением*** (LUP decomposition) матрицы A. Мы докажем, что всякая невырожденная матрица A допускает такое разложение.

Преимущество вычисления LUP-разложения матрицы A основано на том, что система линейных уравнений решается гораздо легче, если ее матрица треугольна, что и выполняется в случае матриц L и U. Найдя LUP-разложение матрицы A, мы можем решить уравнение (28.17) $A\,x = b$ путем решения треугольной системы линейных уравнений, как показано далее. Умножая обе части уравнения $A\,x = b$ на P, мы получим эквивалентное уравнение $P\,A\,x = P\,b$, которое в соответствии с упражнением 28.1-5 эквивалентно перестановке уравнений из (28.16). Используя разложение (28.19), получаем

$$L\,U\,x = P\,b.$$

Теперь мы можем решить полученное уравнение, решая две треугольные системы линейных уравнений. Обозначая $y = U\,x$, где x — решение исходной системы

линейных уравнений, мы сначала решаем нижне-треугольную систему линейных уравнений

$$L\,y = P\,b, \tag{28.20}$$

находя неизвестный вектор y с помощью "прямой подстановки". После этого, имея вектор y, мы решаем верхне-треугольную систему линейных уравнений

$$U\,x = y, \tag{28.21}$$

находя x при помощи "обратной подстановки". Вектор x и есть решение исходной системы линейных уравнений $A\,x = b$, поскольку матрица перестановки P обращаема (см. упражнение 28.1-5):

$$Ax = P^{-1}L\,U\,x = P^{-1}L\,y = P^{-1}P\,b = b.$$

Теперь рассмотрим, как работают прямая и обратная подстановки, а затем приступим к задаче вычисления LUP-разложения.

Прямая и обратная подстановки

Прямая подстановка (forward substitution) позволяет решить нижне-треугольную систему линейных уравнений (28.20) для данных L, P и b за время $\Theta\left(n^2\right)$. Для удобства мы представим перестановку P в компактной форме при помощи массива $\pi\,[1..n]$. Элемент $\pi\,[i]$ при $i = 1, 2, \ldots, n$ указывает, что $P_{i,\pi[i]} = 1$ и $P_{ij} = 0$ для $j \neq \pi\,[i]$. Таким образом, в матрице PA на пересечении i-ой строки и j-го столбца находится элемент $a_{\pi[i],j}$, а i-ым элементом Pb является $b_{\pi[i]}$. Поскольку L — единичная нижне-треугольная матрица, уравнение (28.20) можно переписать следующим образом:

$$\begin{aligned}
y_1 &= b_{\pi[1]}\,,\\
l_{21}y_1 + y_2 &= b_{\pi[2]}\,,\\
l_{31}y_1 + l_{32}y_2 + y_3 &= b_{\pi[3]}\,,\\
&\vdots\\
l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \cdots + y_n &= b_{\pi[n]}\,.
\end{aligned}$$

Мы можем найти значение y_1 непосредственно, поскольку первое уравнение гласит, что $y_1 = b_{\pi[1]}$. Зная y_1, мы можем подставить его во второе уравнение и найти

$$y_2 = b_{\pi[2]} - l_{21}y_1.$$

Теперь мы можем подставить в третье уравнение два найденных значения — y_1 и y_2 и получить

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2)\,.$$

В общем случае для поиска y_i мы подставляем найденные значения $y_1, y_2, \ldots, y_{i-1}$ в i-е уравнение и находим

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j .$$

Обратная подстановка (back substitution) аналогична прямой. Для данных U и y мы сначала решаем n-ое уравнение, а затем идем в обратном направлении к первому уравнению. Так же, как и в случае прямой подстановки, время работы составляет $\Theta\left(n^2\right)$. Поскольку матрица U верхне-треугольная, мы можем переписать (28.21) в следующем виде:

$$\begin{aligned}
u_{11}x_1 + u_{12}x_2 + \cdots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n &= y_1 , \\
u_{22}x_2 + \cdots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n &= y_2 , \\
&\vdots \\
u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n &= y_{n-2} , \\
u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1} , \\
u_{n,n}x_n &= y_n .
\end{aligned}$$

Таким образом мы можем последовательно найти $x_n, x_{n-1}, \ldots, x_1$:

$$\begin{aligned}
x_n &= y_n / u_{n,n} , \\
x_{n-1} &= (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} , \\
x_{n-2} &= (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2} , \\
&\vdots
\end{aligned}$$

или, в общем виде,

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii} .$$

Процедура LUP_SOLVE для данных P, L, U и b находит x путем комбинирования прямой и обратной подстановки. В псевдокоде предполагается, что размерность n хранится в атрибуте $rows\,[L]$ и что матрица перестановки P представлена массивом π.

LUP_SOLVE(L, U, π, b)
1 $n \leftarrow rows[L]$
2 **for** $i \leftarrow 1$ **to** n
3 **do** $y_i \leftarrow b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j$
4 **for** $i \leftarrow n$ **downto** 1
5 **do** $x_i \leftarrow \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j\right) / u_{ii}$
6 **return** x

Процедура LUP_SOLVE находит y в строках 2–3 с помощью прямой подстановки, а затем вычисляет x при помощи обратной подстановки в строках 4–5. Наличие внутри каждого цикла неявного цикла суммирования приводит ко времени работы данной процедуры, равному $\Theta\left(n^2\right)$.

В качестве примера данного метода рассмотрим систему линейных уравнений

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix} ,$$

Здесь

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} , \quad b = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix} ,$$

и мы хотим найти неизвестный вектор x. LUP-разложение матрицы A имеет вид

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix} , \quad U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix} , \quad P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} .$$

(Читатель может самостоятельно убедиться в корректности разложения, проверив выполнение равенства $P\,A = L\,U$.) Используя прямую подстановку, мы находим y из уравнения $L\,y = P\,b$:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix} ,$$

что дает нам путем вычисления сначала y_1, затем y_2 и y_3 окончательное решение

$$y = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix}$$

Используя обратную подстановку, мы решаем уравнение $U\,x = y$ относительно x:

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix},$$

что дает нам требуемое решение исходной системы линейных уравнений

$$x = \begin{pmatrix} -1.4 \\ 2.2 \\ 0.6 \end{pmatrix}$$

путем вычисления сначала x_3, затем x_2 и x_1.

Вычисление LU-разложения

Мы показали, что если можно вычислить LUP-разложение невырожденной матрицы A, то для решения системы линейных уравнений $A\,x = b$ можно воспользоваться простыми прямой и обратной подстановками. Нам осталось показать, как эффективно найти LUP-разложение матрицы A. Мы начнем наше рассмотрение со случая невырожденной матрицы A размера $n \times n$ и отсутствия матрицы P (или, что эквивалентно, $P = I_n$). В этом случае мы должны найти разложение $A = L\,U$. Назовем матрицы L и U ***LU-разложением*** (LU decomposition) матрицы A.

Наш процесс построения LU-разложения называется ***методом исключения Гаусса*** (Gauss elimination) Мы начнем с удаления первой переменной из всех уравнений, кроме первого, путем вычитания из этих уравнений первого, умноженного на соответствующий коэффициент. Затем та же операция будет проделана со вторым уравнением, чтобы в третьем и последующих уравнениях отсутствовали первая и вторая переменные. Процесс будет продолжаться до тех пор, пока мы не получим верхне-треугольную матрицу — это и будет искомая матрица U. Матрица L составляется из коэффициентов, участвовавших в процессе исключения переменных.

Мы реализуем описанную стратегию при помощи рекурсивного алгоритма. Итак, мы хотим построить LU-разложение невырожденной матрицы A размером $n \times n$. Если $n = 1$, задача решена, поскольку мы можем выбрать $L = I_1$ и $U = A$. При $n > 1$ разобьем A на четыре части:

$$A = \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right) = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix},$$

где v — вектор-столбец размером $n-1$, w^{T} — вектор-строка размером $n-1$, а A' — матрица размером $(n-1)\times(n-1)$. Используя матричную алгебру (проверить полученный результат можно при помощи умножения), разложим матрицу A следующим образом:

$$A = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} . \quad (28.22)$$

Нули в первой и второй матрицах разложения представляют собой вектор-строку и вектор-столбец размера $n-1$ с нулевыми элементами. Матрица $v\,w^{\mathrm{T}}/a_{11}$, получающаяся тензорным произведением векторов v и w и делением каждого элемента получившейся в результате умножения матрицы на a_{11}, представляет собой матрицу размером $(n-1)\times(n-1)$ (тот же размер имеет и матрица A', из которой она вычитается). Получающаяся в результате матрица размером $(n-1)\times(n-1)$

$$A' - vw^{\mathrm{T}}/a_{11} \quad (28.23)$$

называется ***дополнением Шура*** (Schur complement) элемента a_{11} в матрице A.

Из требования невырожденности A вытекает невырожденность дополнения Шура. Почему? Предположим, что дополнение Шура представляет собой вырожденную матрицу размером $(n-1)\times(n-1)$. Тогда по теореме 28.1 она имеет ранг, строго меньший $n-1$. Поскольку нижние $n-1$ элементов в первом столбце матрицы

$$\begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}$$

равны 0, нижние $n-1$ строк должны иметь ранг, строго меньше $n-1$. Таким образом, ранг всей матрицы строго меньше n. Применяя результат упражнения 28.1-10 к уравнению (28.22), находим, что ранг матрицы A строго меньше n, так что согласно теореме 28.1 мы получаем противоречие с начальным условием невырожденности A.

Поскольку дополнение Шура невырождено, мы можем рекурсивно найти его LU-разложение $A' - v\,w^{\mathrm{T}}/a_{11} = L'\,U'$, где L' — нижне-треугольная, а U' — верхне-треугольная матрицы. Используя матричную алгебру, получим:

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} = \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} = \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} = LU , \end{aligned}$$

что дает нам искомое LU-разложение. (Заметим, что поскольку матрица L' — единичная нижне-треугольная, таковой же является и матрица L; поскольку матрица U' — верхне-треугольная, таковой же является и матрица U.)

Конечно, если $a_{11} = 0$, этот метод не работает, поскольку при этом должно быть выполнено деление на 0. Он также не работает, если верхний левый элемент дополнения Шура $A' - vw^{\mathrm{T}}/a_{11}$ равен нулю, поскольку в этом случае деление на ноль возникнет на следующем шаге рекурсии. Элементы, на которые в процессе LU-разложения выполняется деление, называются ***ведущими*** (pivots), и они располагаются на диагонали матрицы U. Причина, по которой в LUP-разложение включается матрица перестановок P, состоит в том, чтобы избежать деления на нулевые элементы. Использование матрицы перестановки для того, чтобы избежать деления на ноль (или на малые величины), называется ***выбором ведущего элемента*** (pivoting).

Важным классом матриц, для которых LU-разложение всегда работает корректно, является класс симметричных положительно определенных матриц. Такие матрицы не требуют выбора ведущего элемента, и описанная стратегия может быть применена без опасения столкнуться с делением на ноль. Мы докажем это в разделе 28.5.

Наш код для LU-разложения матрицы A следует рекурсивной стратегии, хотя рекурсия в нем и заменена итеративным циклом (такое преобразование является стандартной оптимизацией процедуры с оконечной рекурсией, когда последней операцией процедуры является рекурсивный вызов ее самой). В коде предполагается, что размерность матрицы A хранится в атрибуте $rows\,[A]$. Поскольку мы знаем, что в вычисляемой матрице U все элементы ниже диагонали равны 0 и процедура LUP_SOLVE к ним не обращается, наш код не заполняет их никакими значениями. Аналогично, поскольку в матрице L диагональные элементы равны 1, а элементы выше диагонали — нулевые, код процедуры не заполняет их, ограничиваясь только “значащими” элементами матриц U и L.

```
LU_DECOMPOSITION(A)
 1  n ← rows[A]
 2  for k ← 1 to n
 3       do u_kk ← a_kk
 4          for i ← k + 1 to n
 5               do l_ik ← a_ik/u_kk      ▷ l_ik содержит v_i
 6                  u_ki ← a_ki           ▷ u_ki содержит w_i^T
 7          for i ← k + 1 to n
 8               do for j ← k + 1 to n
 9                       do a_ij ← a_ij − l_ik u_kj
10  return L и U
```

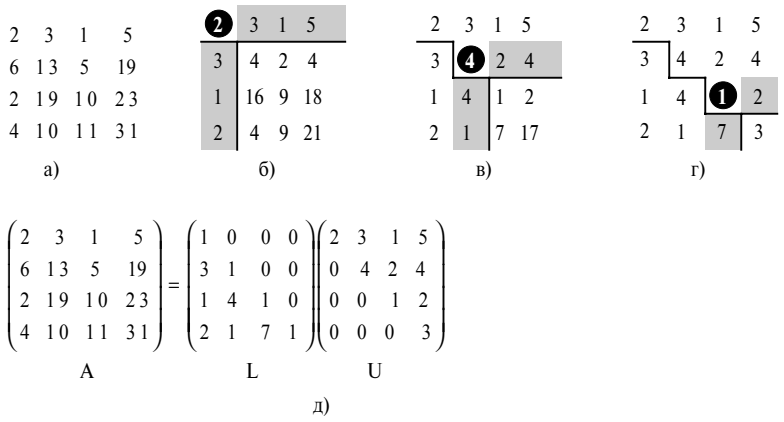


Рис. 28.1. Работа процедуры LU_DECOMPOSITION

Внешний цикл **for**, начинающийся в строке 2, выполняет каждый шаг рекурсии по одному разу. В теле этого цикла в строке 3 ведущим элементом полагается элемент a_{kk}. В цикле **for** в строках 4–6 (который не выполняется, когда k равно n) для обновления матриц U и L используются векторы v и w^{T}. Элементы вектора v определяются в строке 5 и сохраняются в l_{ik}, а элементы w^{T} определяются в строке 6, после чего каждый элемент w_i^{T} сохраняется в u_{ki}. И наконец, элементы дополнения Шура вычисляются в строках 7–9 и сохраняются в матрице A (в строке 9 нам не требуется деление на a_{kk}, так как оно уже выполнено в строке 5 при вычислении l_{ik}). В связи с тройной вложенностью в циклы строки 9 время работы процедуры LU_DECOMPOSITION равно $\Theta\left(n^3\right)$.

На рис. 28.1 показан процесс работы процедуры LU_DECOMPOSITION. На рис. 28.1*а* показана исходная матрица, на рис. 28.1*б–г* — шаги рекурсии. Ведущий элемент выделен черным цветом, а вычисляемые элементы матриц U и L — серым. На рис. 28.1*д* показан окончательный результат LU-разложения.

В приведенной реализации используется стандартная оптимизация, состоящая в том, что значащие элементы матриц U и L хранятся на месте элементов матрицы A, т.е. мы устанавливаем соответствие между каждым элементом a_{ij} и либо l_{ij} (если $i > j$), либо u_{ij} (если $i \leqslant j$) и обновляем матрицу A так, что при выходе из процедуры она будет содержать значащие элементы матриц U и L. Для этого в псевдокоде достаточно заменить все обращения к u и l на обращения к a; нетрудно убедиться, что такое преобразование кода сохраняет его корректность.

Вычисление LUP-разложения

В общем случае при решении системы линейных уравнений $A\,x = b$ мы можем оказаться должны выбирать ведущие элементы среди недиагональных элементов

матрицы A для того, чтобы избежать деления на 0. Причем нежелательно не только деление на 0, но и просто на малое число, даже если матрица A невырождена, поскольку в результате понижается численная устойчивость вычислений. В связи с этим в качестве ведущего элемента следует выбирать наибольший возможный элемент.

Математические основы LUP-разложения аналогичны LU-разложению. Нам дана невырожденная матрица A размером $n \times n$, и нам требуется найти матрицу перестановки P, единичную нижне-треугольную матрицу L и верхне-треугольную матрицу U такие, что $P\,A = L\,U$. Перед разделением матрицы A на части, как мы делали при вычислении LU-разложения, мы перемещаем ненулевой элемент, скажем, a_{k1}, откуда-то из первого столбца матрицы в позицию $(1,1)$ (если первый столбец содержит только нулевые значения, то матрица вырождена, поскольку ее определитель равен 0 (см. теоремы 28.4 и 28.5)). Для сохранения множества исходных линейных уравнений мы меняем местами строки 1 и k, что эквивалентно умножению матрицы A на матрицу перестановки Q слева (см. упражнение 28.1-5). Тогда мы можем записать QA как

$$QA = \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} ,$$

где $v = (a_{21}, a_{31}, \ldots, a_{k-1}, a_{11}, a_{k+1}, \ldots, a_{n1})^{\mathrm{T}}$ (k-й элемент заменен первым), $w^{\mathrm{T}} = (a_{k2}, a_{k3}, \ldots, a_{kn})$, а A' — матрица размером $(n-1) \times (n-1)$. Поскольку $a_{k1} \neq 0$, мы можем выполнить те же преобразования, что и при LU-разложении, не опасаясь деления на 0:

$$QA = \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} .$$

Как мы видели в LU-разложении, если матрица A невырождена, то невырождено и дополнение Шура $A' - vw^{\mathrm{T}}/a_{k1}$. Таким образом, мы можем индуктивно найти его LUP-разложение, дающее единичную нижне-треугольную матрицу L', верхне-треугольную матрицу U' и матрицу перестановки P' такие, что $P'\left(A' - vw^{\mathrm{T}}/a_{k1}\right) = L'\,U'$.

Определим матрицу

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q .$$

которая является матрицей перестановки в силу того, что она представляет собой произведение двух матриц перестановки (см. упражнение 28.1-5). Мы имеем

$$
\begin{aligned}
PA &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA = \\
&= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} = \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} = \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & P'(A' - vw^{\mathrm{T}}/a_{k1}) \end{pmatrix} = \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} = \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} = \\
&= LU\ ,
\end{aligned}
$$

что и дает нам искомое LUP-разложение. Поскольку L' — единичная нижне-треугольная матрица, такой же будет и L, и поскольку U' — верхне-треугольная матрица, такой же будет и U.

Заметим, что в отличие от LU-разложения, и вектор-столбец v/a_{k1}, и дополнение Шура $A' - vw^{\mathrm{T}}/a_{k1}$ должны умножаться на матрицу перестановки P'.

Как и ранее, в псевдокоде LUP-разложения рекурсия заменяется итерацией. В качестве усовершенствования непосредственной реализации рассмотренной рекурсии матрицу перестановки P мы представляем массивом π, где $\pi[i] = j$ означает, что i-я строка массива P содержит 1 в столбце j. Мы также реализуем код, который вычисляет L и U на месте матрицы A, т.е. по окончании работы процедуры

$$
a_{ij} = \begin{cases} l_{ij} & \text{при } i > j, \\ u_{ij} & \text{при } i \leqslant j. \end{cases}
$$

```
LUP_DECOMPOSITION(A)
 1  n ← rows[A]
 2  for i ← 1 to n
 3      do π[i] ← i
 4  for k ← 1 to n
 5      do p ← 0
 6          for i ← k to n
 7              do if |a_ik| > p
```

```
 8                then p ← |a_ik|
 9                     k' ← i
10          if p = 0
11            then error "Матрица вырождена"
12          Обмен π[k] ↔ π[k']
13          for i ← 1 to n
14              do  Обмен a_ki ↔ a_k'i
15                  for i ← k + 1 to n
16                      do a_ik ← a_ik/a_kk
17                         for j ← k + 1 to n
18                             do a_ij ← a_ij − a_ik a_kj
```

На рис. 28.2 продемонстрирована работа процедуры LUP_DECOMPOSITION по разложению матрицы. На рис. 28.2*а* показана исходная матрица с тождественной перестановкой строк (массив перестановок показан слева от матрицы). На рис. 28.2*б* показан выбор максимального ведущего элемента и соответствующий обмен строк. заштрихованные столбец и строка представляют v и w^{T}. На рис. 28.2*в* показано деление вектора v на ведущий элемент и вычисление дополнения Шура. Линии на рисунке делят матрицу на три части: элементы U над линией, элементы L слева от линии и элементы дополнения Шура в правой нижней части. На рис. 28.2*г–е* показан второй шаг вычисления LUP-разложения, а на рис. 28.2*ж–и* — третий шаг. На рис. 28.2*к* представлен окончательный результат LUP-разложения $P\,A = L\,U$.

Массив π инициализируется в строках 2–3 тождественной перестановкой. Внешний цикл **for**, начинающийся в строке 4, реализует рекурсию. При каждом выполнении тела внешнего цикла в строках 5–9 определяется элемент $a_{k'k}$ с наибольшим абсолютным значением в текущем первом столбце (столбце k) матрицы размером $(n-k+1)\times(n-k+1)$, разложение которой должно быть найдено. Если все элементы текущего первого столбца нулевые, в строках 10–11 сообщается о вырожденности матрицы. Далее мы делаем $a_{k'k}$ ведущим элементом. Для этого в строке 12 выполняется обмен элементов $\pi\,[k]$ и $\pi\,[k']$ массива π, а в строках 13–14 — обмен k-ой и k'-ой строк матрицы A. (Мы выполняем обмен строк полностью, поскольку, как говорилось в изложении метода, на P' умножается не только дополнение Шура $A' - vw^{\mathrm{T}}/a_{k1}$, но и вектор v/a_{k1}.) Наконец, в строках 15–18 вычисляется дополнение Шура (практически так же, как и в строках 4–9 процедуры LU_DECOMPOSITION, с тем отличием, что вычисленные значения заносятся в матрицу A, а не в отдельные матрицы L и U).

В силу тройной вложенности циклов время работы процедуры LUP_DECOMPOSITION равно $\Theta\left(n^3\right)$, т.е. оно точно такое же, как и в случае процедуры LU_DECOMPOSITION. Следовательно, выбор ведущего элемента приводит к увеличению времени работы не более чем на постоянный множитель.

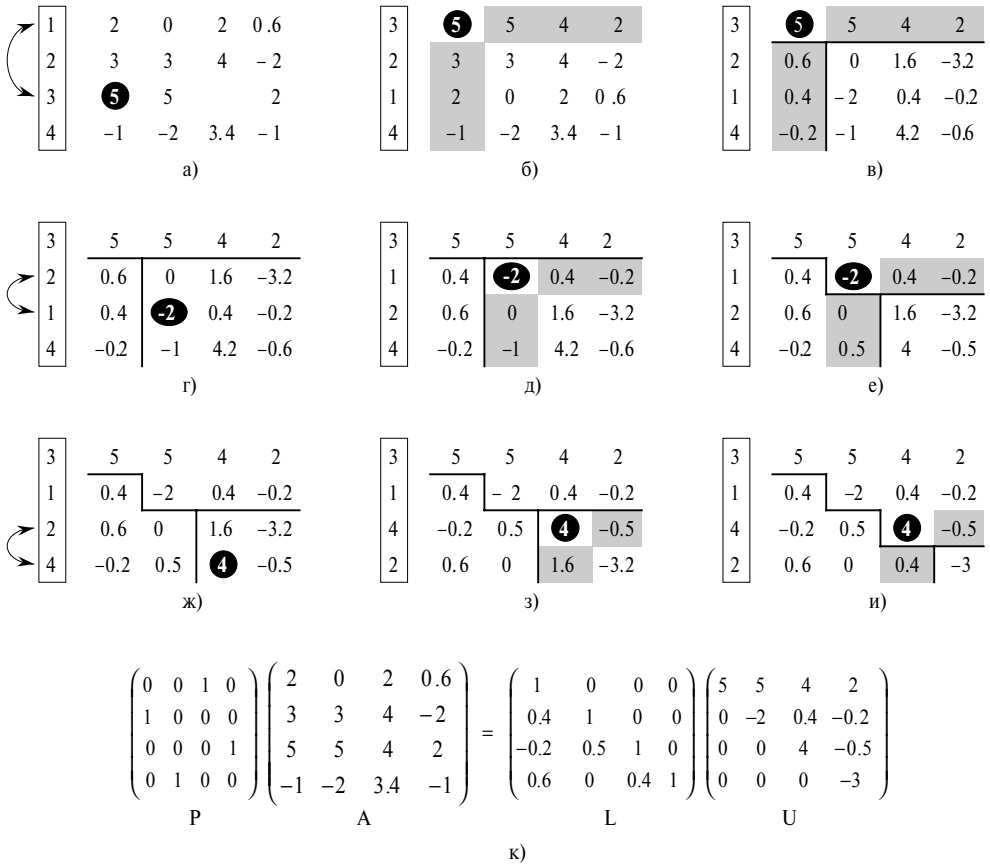


Рис. 28.2. Работа процедуры LUP_DECOMPOSITION

Упражнения

28.3-1. Решите при помощи прямой подстановки уравнение

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

28.3-2. Найдите LU-разложение матрицы

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix}.$$

28.3-3. Решите с помощью LUP-разложения систему линейных уравнений

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

28.3-4. Как выглядит LUP-разложение диагональной матрицы?

28.3-5. Как найти LUP-разложение матрицы перестановки? Докажите его единственность.

28.3-6. Покажите, что для всех $n \geqslant 1$ существует вырожденная матрица размером $n \times n$, имеющая LU-разложение.

28.3-7. Необходимо ли выполнение тела внешнего цикла **for** в процедуре LU_DECOMPOSITION при $k = n$? А в процедуре LUP_DECOMPOSITION?

28.4 Обращение матриц

На практике для решения систем линейных уравнений обращение матриц не используется, вместо этого применяются другие, численно более устойчивые методы, например, LUP-разложение. Тем не менее, задача обращения матриц имеет как теоретический, так и сугубо практический интерес. В этом разделе мы покажем, что для обращения матриц можно использовать уже рассмотренный нами метод LUP-разложения. Мы также докажем, что умножение матриц и обращение матрицы — задачи одинаковой сложности, так что для решения одной задачи мы можем использовать алгоритм для решения другой, чтобы получить одинаковое асимптотическое время работы. В частности, мы можем применить алгоритм Штрассена для обращения матриц (к слову, в своей работе Штрассен преследовал цель ускорить решение систем линейных уравнений).

Вычисление обратной матрицы из LUP-разложения

Предположим, что у нас имеется LUP-разложение матрицы A на три матрицы L, U и P такие, что $PA = LU$. Используя процедуру LUP_SOLVE, мы можем решить уравнение вида $Ax = b$ за время $\Theta\left(n^2\right)$. Поскольку LUP-разложение зависит только от A, но не от b, мы можем использовать ту же процедуру для решения другой системы линейных уравнений вида $Ax = b'$ за то же время $\Theta\left(n^2\right)$. Обобщая, имея LUP-разложение матрицы A, мы можем решить k систем линейных уравнений с одной и той же матрицей A за время $\Theta\left(kn^2\right)$.

Уравнение

$$AX = I_n \tag{28.24}$$

можно рассматривать как множество из n различных систем линейных уравнений вида $A\,x = b$. Эти уравнения позволяют найти матрицу X, обратную матрице A. Обозначим через X_i i-ый столбец X, и вспомним, что i-ым столбцом матрицы I_n является единичный вектор e_i. Мы можем найти X в уравнении (28.24), использовав LUP-разложение для решения набора уравнений

$$AX_i = e_i$$

для каждого X_i в отдельности. Поиск каждого столбца X_i требует времени $\Theta\left(n^2\right)$, так что полное время вычисления обратной матрицы X на основе LUP-разложения исходной матрицы A требует времени $\Theta\left(n^3\right)$. Поскольку LUP-разложение A также вычисляется за время $\Theta\left(n^3\right)$, задача обращения матрицы решается за время $\Theta\left(n^3\right)$.

Умножение матриц и обращение матрицы

Теперь мы покажем, каким образом можно использовать ускоренное умножение матриц для ускорения обращения матрицы (это ускорение имеет скорее теоретический интерес, чем практическое применение). В действительности мы докажем более строгое утверждение — что умножение матриц эквивалентно обращению матрицы в следующем смысле. Если обозначить через $M(n)$ время, необходимое для умножения двух матриц размером $n \times n$, то имеется способ обратить матрицу размером $n \times n$ за время $O(M(n))$. Кроме того, если обозначить через $I(n)$ время, необходимое для обращения матрицы размером $n \times n$, то имеется способ перемножит две матрицы размера $n \times n$ за время $O(I(n))$. Мы докажем эти утверждения по отдельности.

Теорема 28.7 (Умножение не сложнее обращения). Если мы можем обратить матрицу размером $n \times n$ за время $I(n)$, где $I(n) = \Omega\left(n^2\right)$ и $I(n)$ удовлетворяет условию регулярности $I(3n) = O(I(n))$, то мы можем умножить две матрицы размера $n \times n$ за время $O(I(n))$.

Доказательство. Пусть A и B — матрицы размером $n \times n$, произведение которых C мы хотим найти. Определим матрицу D размером $3n \times 3n$ следующим образом:

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix} .$$

Обратная к D матрица имеет следующий вид:

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix} ,$$

так что мы можем вычислить произведение AB как верхнюю правую подматрицу размера $n \times n$ матрицы D^{-1}.

Матрицу D мы можем построить за время $\Theta\left(n^2\right) = O\left(I\left(n\right)\right)$, а обратить ее — за время $O\left(I\left(3n\right)\right) = O\left(I\left(n\right)\right)$ в соответствии с условием регулярности $I\left(n\right)$. Таким образом, мы получаем $M\left(n\right) = O\left(I\left(n\right)\right)$. ■

Заметим, что $I\left(n\right) = \Theta\left(n^c \lg^d n\right)$ удовлетворяет условию регулярности при любых константах $c > 0$ и $d \geqslant 0$.

Доказательство того, что обращение матрицы не сложнее умножения, опирается на некоторые свойства симметричных положительно определенных матриц, которые мы докажем в разделе 28.5.

Теорема 28.8 (Обращение не сложнее умножения). Предположим, что мы можем умножить две действительные матрицы размером $n \times n$ за время $M\left(n\right)$, где $M\left(n\right) = \Omega\left(n^2\right)$ и, кроме того, удовлетворяет условию регулярности $M\left(n + k\right) = O\left(M\left(n\right)\right)$ для произвольного $0 \leqslant k \leqslant n$, а также $M\left(n/2\right) \leqslant cM\left(n\right)$ для некоторой константы $c < 1/2$. В таком случае мы можем обратить любую действительную невырожденную матрицу размером $n \times n$ за время $O\left(M\left(n\right)\right)$.

Доказательство. Мы можем считать, что n — точная степень 2, поскольку для любого $k > 0$

$$\begin{pmatrix} A & 0 \\ 0 & I_n \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_n \end{pmatrix}$$

Таким образом, выбрав k так, чтобы величина $n+k$ была степенью 2, мы увеличиваем исходную матрицу до размера, представляющего собой степень 2, а обратную матрицу A^{-1} получаем как часть обращенной матрицы большего размера. Первое условие регулярности $M\left(n\right)$ гарантирует, что такое увеличение не вызовет увеличения времени работы более чем на постоянный множитель.

Предположим теперь, что матрица A размером $n \times n$ симметрична и положительно определенная. Разделим A на четыре матрицы размером $n/2 \times n/2$:

$$A = \begin{pmatrix} B & C^{\mathrm{T}} \\ C & D \end{pmatrix} . \tag{28.25}$$

Обозначив через S дополнение Шура подматрицы B в матрице A

$$S = D - CB^{-1}C^{\mathrm{T}} \tag{28.26}$$

мы получим

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1} & -B^{-1}C^{\mathrm{T}}S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix} , \tag{28.27}$$

поскольку $A\,A^{-1} = I_n$. Приведенное выражение легко проверить непосредственным матричным умножением. Согласно леммам 28.9, 28.10 и 28.11 из раздела 28.5, если матрица A — симметричная положительно определенная, то существуют и матрицы B^{-1} и S^{-1}, поскольку и B, и S — симметричные положительно определенные матрицы. Согласно упражнению 28.1-2, $B^{-1}C^{\mathrm{T}} = \left(C\,B^{-1}\right)^{\mathrm{T}}$ и $B^{-1}\,C^{\mathrm{T}}\,S^{-1} = \left(S^{-1}\,C\,B^{-1}\right)^{\mathrm{T}}$. Уравнения (28.26) и (28.27) могут таким образом быть использованы для определения рекурсивного алгоритма, требующего четыре умножения матриц размером $n/2 \times n/2$:

$$\begin{aligned}
&C \cdot B^{-1}\ ,\\
&(CB^{-1}) \cdot C^{\mathrm{T}}\ ,\\
&S^{-1} \cdot (CB^{-1})\ ,\\
&(CB^{-1})^{\mathrm{T}} \cdot (S^{-1}CB^{-1})\ .
\end{aligned}$$

Таким образом, для обращения симметричной положительно определенной матрицы размером $n \times n$ надо инвертировать две матрицы размером $n/2 \times n/2$ (B и S), выполнить четыре указанных умножения матриц размером $n/2 \times n/2$ (это умножение можно осуществить при помощи алгоритма умножения матриц размером $n \times n$), а также выполнить дополнительные действия по извлечению подматриц из A (стоимость $O\left(n^2\right)$) и некоторое постоянное количество сложений и вычитаний матриц размером $n/2 \times n/2$. В результате мы получаем следующее рекуррентное соотношение:

$$I\,(n) \leqslant 2I\,(n/2) + 4M\,(n) + O\left(n^2\right) = 2I\,(n/2) + \Theta\,(M\,(n)) = O\,(M\,(n))\,.$$

Первое равенство в приведенном соотношении выполняется в силу того, что $M\,(n) = \Omega\left(n^2\right)$, а второе получается потому, что второе условие регулярности, накладываемое в теореме на $M\,(n)$, позволяет нам использовать случай 3 теоремы 4.1.

Нам остается доказать, что асимптотическое время умножения матриц может быть получено для обращения невырожденной матрицы A, которая не является симметричной и положительно определенной. Основная идея заключается в том, что для любой невырожденной матрицы A матрица $A^{\mathrm{T}}\,A$ симметричная (согласно упражнению 28.1-2) и положительно определенная (в соответствии с теоремой 28.6). Все, что остается — это привести задачу обращения матрицы A к задаче обращения матрицы $A^{\mathrm{T}}\,A$.

Такое приведение основано на наблюдении, что если A — невырожденная матрица размера $n \times n$, тогда

$$A^{-1} = \left(A^{\mathrm{T}}A\right)^{-1} A^{\mathrm{T}},$$

поскольку

$$\left(\left(A^{\mathrm{T}}A\right)^{-1}A^{\mathrm{T}}\right)A = \left(A^{\mathrm{T}}A\right)^{-1}\left(A^{\mathrm{T}}A\right) = I_n,$$

а обратная матрица единственна. Следовательно, мы можем вычислить A^{-1}, сначала находя A^{T} для получения симметричной положительно определенной матрицы $A^{\mathrm{T}}A$, затем обращая эту матрицу с помощью описанного выше рекурсивного алгоритма, а затем умножая полученный результат на A^{T}. Каждый из перечисленных шагов требует $O\left(M\left(n\right)\right)$ времени, так что обращение любой невырожденной матрицы с действительными элементами может быть выполнено за время $O\left(M\left(n\right)\right)$. ■

Доказательство теоремы 28.8 наводит на мысль о том, каким образом можно решать систему уравнений $A\,x = b$ с невырожденной матрицей A при помощи LU-разложения, без выбора ведущего элемента. Для этого обе части уравнения нужно умножить на A^{T}, получая уравнение $\left(A^{\mathrm{T}}A\right)x = A^{\mathrm{T}}b$. Такое преобразование не влияет на x в силу обращаемости матрицы A^{T}, а тот факт, что матрица $A^{\mathrm{T}}A$ является симметричной положительно определенной, позволяет использовать для решения метод LU-разложения. После этого применение прямой и обратной подстановки с использованием правой части уравнения, равной $A^{\mathrm{T}}b$, даст нам решение x исходного уравнения. Хотя теоретически этот метод вполне корректен, на практике процедура LUP_DECOMPOSITION работает существенно лучше. Во-первых, она требует меньшего количества арифметических операций (отличающееся постоянным множителем от количества операций при описанном методе), и во-вторых, численная устойчивость LUP_DECOMPOSITION несколько выше.

Упражнения

28.4-1. Пусть $M\left(n\right)$ — время, необходимое для умножения двух матриц размером $n \times n$, а $S\left(n\right)$ — время, необходимое для возведения матрицы размером $n \times n$ в квадрат. Покажите, что умножение матриц и возведение матрицы в квадрат имеют одну и ту же сложность: из $M\left(n\right)$-алгоритма умножения матриц вытекает $O\left(M\left(n\right)\right)$-алгоритм возведения матрицы в квадрат, а $S\left(n\right)$-алгоритм возведения матрицы в квадрат дает $O\left(S\left(n\right)\right)$-алгоритм умножения матриц.

28.4-2. Пусть $M\left(n\right)$ — время, необходимое для умножения двух матриц размером $n \times n$, а $L\left(n\right)$ — время, необходимое для LUP-разложения матрицы того же размера. Покажите, что умножение матриц и LUP-разложение имеют одинаковую сложность, т.е. что из $M\left(n\right)$-алгоритма умножения матриц вытекает $O\left(M\left(n\right)\right)$-алгоритм LUP-разложения, а $L\left(n\right)$-алгоритм LUP-разложения дает $O\left(L\left(n\right)\right)$-алгоритм умножения матриц.

28.4-3. Пусть $M(n)$ — время, необходимое для умножения двух матриц размером $n \times n$, а $D(n)$ — время, необходимое для вычисления определителя матрицы того же размера. Покажите, что умножение матриц и вычисление определителя имеют одинаковую сложность, т.е. что из $M(n)$-алгоритма умножения матриц вытекает $O(M(n))$-алгоритм вычисления определителя, а $D(n)$-алгоритм вычисления определителя дает $O(D(n))$-алгоритм умножения матриц.

28.4-4. Пусть $M(n)$ — время, необходимое для умножения двух булевых матриц размером $n \times n$, а $T(n)$ — время, необходимое для поиска транзитивного замыкания булевой матрицы размером $n \times n$ (см. раздел 25.2). Покажите, что $M(n)$-алгоритм умножения булевых матриц дает $O(M(n) \lg n)$-алгоритм поиска транзитивного замыкания, а $T(n)$-алгоритм поиска транзитивного замыкания приводит к $O(T(n))$-алгоритму умножения булевых матриц.

28.4-5. Применим ли основанный на теореме 28.8 алгоритм обращения матриц к матрицам над полем целых чисел по модулю 2? Поясните свой ответ.

⋆ 28.4-6. Обобщите алгоритм обращения матриц из теоремы 28.8 для матриц с комплексными числами, и докажите корректность вашего обобщения. (*Указание*: вместо транспонирования матрицы A воспользуйтесь ***сопряженно-транспонированной*** (conjugate transpose) матрицей A^*, которая получается из исходной путем транспонирования и замены всех элементов комплексно сопряженными числами. Вместо симметричных матриц рассмотрите ***эрмитовы*** (Hermitian) матрицы, обладающие тем свойством, что $A = A^*$.)

28.5 Симметричные положительно определенные матрицы и метод наименьших квадратов

Симметричные положительно определенные матрицы обладают рядом интересных и полезных свойств. Например, они невырождены и их LU-разложение можно выполнить, не опасаясь столкнуться с делением на 0. В этом разделе мы докажем ряд важных свойств симметричных положительно определенных матриц и покажем их применение для получения приближений методом наименьших квадратов.

Первое свойство, которое мы докажем, пожалуй, наиболее фундаментальное.

Лемма 28.9. Любая симметричная положительно определенная матрица является невырожденной.

Доказательство. Предположим, что матрица A вырождена. Тогда, согласно следствию 28.3, имеется такой ненулевой вектор x, что $Ax = 0$. Следовательно, $x^{\mathrm{T}}Ax = 0$, и A не может быть положительно определенной. ■

Доказательство того факта, что LU-разложение симметричных положительно определенных матриц можно выполнить, не опасаясь столкнуться с делением на 0, более сложное. Мы начнем с доказательства свойств некоторых определенных подматриц A. Определим ***k-ую главную подматрицу*** (leading submatrix) A как матрицу A_k, состоящую из пересечения k первых строк и k первых столбцов A.

Лемма 28.10. Если A — симметричная положительно определенная матрица, то все ее главные подматрицы симметричные и положительно определенные.

Доказательство. То, что каждая главная подматрица A_k является симметричной, очевидно. Для доказательства того, что она положительно определенная, воспользуемся методом от противного. Если A_k не является положительно определенной, то существует вектор $x_k \neq 0$ размером k такой, что $x_k^{\mathrm{T}} A_k x_k \leqslant 0$. Пусть матрица A имеет размер $n \times n$. Определим вектор $x = \begin{pmatrix} x_k^{\mathrm{T}} & 0 \end{pmatrix}^{\mathrm{T}}$ размером n, в котором после x_k следуют $n - k$ нулей. Тогда

$$\begin{aligned} x^{\mathrm{T}}Ax &= \begin{pmatrix} x_k^{\mathrm{T}} & 0 \end{pmatrix} \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \begin{pmatrix} x_k \\ 0 \end{pmatrix} = \\ &= \begin{pmatrix} x_k^{\mathrm{T}} & 0 \end{pmatrix} \begin{pmatrix} A_k x_k \\ B x_k \end{pmatrix} = \\ &= x_k^{\mathrm{T}} A_k x_k \leqslant 0, \end{aligned}$$

что противоречит условию, что матрица A положительно определенная. ■

Теперь мы рассмотрим дополнение Шура. Пусть A — симметричная положительно определенная матрица, а A_k — главная подматрица A размера $k \times k$. Разделим A на части следующим образом:

$$A = \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} . \tag{28.28}$$

Обобщим определение (28.23) для определения ***дополнения Шура матрицы A относительно подматрицы A_k*** (Schur complement of A with respect to A_k) следующим образом:

$$S = C - BA_k^{-1}B^{\mathrm{T}} . \tag{28.29}$$

(В соответствии с леммой 28.10 A_k — симметричная положительно определенная матрица; следовательно, согласно лемме 28.9, A_k^{-1} существует, и дополнение Шура S вполне определено.) Заметим, что прежнее определение (28.23) дополнения Шура согласуется с определением (28.29), если принять k равным 1.

Следующая лемма показывает, что дополнение Шура симметричной положительно определенной матрицы является симметричным положительно определенным. Этот результат используется в теореме 28.8, а следствие из него необходимо для доказательства корректности LU-разложения симметричных положительно определенных матриц.

Лемма 28.11 (Лемма о дополнении Шура). Если A — симметричная положительно определенная матрица, а A_k — главная подматрица A размером $k \times k$, то дополнение Шура к подматрице A_k матрицы A является симметричным положительно определенным.

Доказательство. Поскольку матрица A симметрична, симметрична также подматрица C. Согласно упражнению 28.1-8, произведение $BA_k^{-1}B^{\mathrm{T}}$ является симметричным, так что в соответствии с упражнением 28.1-1 дополнение Шура S также является симметричным.

Остается показать, что дополнение Шура положительно определенное. Рассмотрим разделение A (28.28). Для любого ненулевого вектора x в соответствии с тем, что A — положительно определенная матрица, выполняется соотношение $x^{\mathrm{T}}Ax > 0$. Разобьем x на два подвектора y и z, совместимые с A_k и C соответственно. В силу существования A_k^{-1} имеем:

$$\begin{aligned} x^{\mathrm{T}}Ax &= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \\ &= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k y + B^{\mathrm{T}} z \\ By + Cz \end{pmatrix} \\ &= y^{\mathrm{T}} A_k y + y^{\mathrm{T}} B^{\mathrm{T}} z + z^{\mathrm{T}} By + z^{\mathrm{T}} Cz \\ &= (y + A_k^{-1} B^{\mathrm{T}} z)^{\mathrm{T}} A_k (y + A_k^{-1} B^{\mathrm{T}} z) + z^{\mathrm{T}} (C - BA_k^{-1} B^{\mathrm{T}}) z \,, \end{aligned} \tag{28.30}$$

(Корректность приведенного выражения можно проверить непосредственным вычислением.) Последнее равенство соответствует выделению полного квадрата из квадратичной формы (см. упражнение 28.5-2).

Поскольку неравенство $x^{\mathrm{T}} A\, x > 0$ выполняется для любого ненулевого x, для любого ненулевого z выберем $y = -A_k^{-1} B^{\mathrm{T}} z$, что удаляет первое слагаемое в (28.30), оставляя только

$$z^{\mathrm{T}} \left(C - B\, A_k^{-1} B^{\mathrm{T}} \right) z = z^{\mathrm{T}} S\, z$$

в качестве значения всего выражения. Итак, для любого $z \neq 0$ мы имеем $z^{\mathrm{T}} S z = x^{\mathrm{T}} A x > 0$, так что дополнение Шура является положительно определенным. ■

Следствие 28.12. LU-разложение симметричной положительно определенной матрицы никогда не приводит к делению на 0.

Доказательство. Пусть A — симметричная положительно определенная матрица. Докажем несколько более строгое утверждение, чем формулировка данного следствия: каждый ведущий элемент строго положителен. Первым ведущим элементом является a_{11}. Этот элемент положителен по определению положительно определенной матрицы A, так как его можно получить с помощью единичного вектора e_1 следующим образом: $a_{11} = e_1^{\mathrm{T}} A e_1 > 0$. На следующем шаге рекурсии LU-разложение применяется к дополнению Шура матрицы A относительно подматрицы $A_1 = (a_{11})$, которое в соответствии с леммой 28.11 является положительно определенным, так что по индукции все ведущие элементы в процессе LU-разложения оказываются положительны. ■

Метод наименьших квадратов

Подбор кривой для множества точек представляет собой важное применение симметричных положительно определенных матриц. Предположим, что нам дано множество из m точек

$$(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m),$$

где значения y_i содержат ошибки измерений. Мы хотим найти функцию $F(x)$, такую что ошибки аппроксимации

$$\eta_i = F(x_i) - y_i \tag{28.31}$$

малы при $i = 1, 2, \ldots, m$. Вид функции F зависит от рассматриваемой задачи, и здесь мы будем считать, что она имеет вид линейной взвешенной суммы

$$F(x) = \sum_{j=1}^{n} c_j f_j(x),$$

где количество слагаемых n и набор ***базисных функций*** (basis functions) f_j выбираются на основе знаний о рассматриваемой задаче. Зачастую в качестве базисных функций выбираются $f_j(x) = x^{j-1}$, т.е. функция F представляет собой полином степени $n - 1$:

$$F(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_n x^{n-1}.$$

При $n = m$ можно найти функцию F, которая удовлетворяет соотношению (28.31) с нулевыми погрешностями. Такой выбор функции F не слишком удачен, так как помимо данных он учитывает и весь "шум", что приводит к плохим результатам при использовании F для предсказания значения y для некоторого x, измерения для которого еще не выполнялись. Обычно гораздо лучшие результаты получаются при n значительно меньшем, чем m, поскольку тогда есть надежда отфильтровать шум, вносимый ошибками измерений. Для выбора значения n имеются определенные теоретические предпосылки, но данная тема лежит за пределами нашей книги. В любом случае, когда n выбрано, мы получаем переопределенную систему линейных уравнений, приближенное решение которой хотим найти. Сейчас мы покажем, каким образом это можно сделать.

Пусть A — матрица значений базисных функций в заданных точках:

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix}$$

т.е. $a_{ij} = f_j(x_i)$, и пусть $c = (c_k)$ — искомый вектор коэффициентов размером n. Тогда

$$Ac = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix}$$

представляет собой вектор размера m "предсказанных значений" y, а вектор $\eta = Ac - y$ — вектор ***невязок*** (approximation error) размера m.

Для минимизации ошибок приближения мы будем минимизировать норму вектора ошибок η, что отражено в названии "решение по методу ***наименьших квадратов***" (least-square solution):

$$\|\eta\| = \left(\sum_{i=1}^{m} \eta_i^2 \right)^{1/2}.$$

Поскольку

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right)^2,$$

мы можем минимизировать $\|\eta\|$, дифференцируя $\|\eta\|^2$ по всем c_k и приравнивая полученные производные к 0:

$$\frac{d\,\|\eta\|^2}{dc_k} = \sum_{i=1}^{m} 2\left(\sum_{j=1}^{n} a_{ij}c_j - y_i\right) a_{ik} = 0\ . \tag{28.32}$$

n уравнений (28.32) эквивалентны одному матричному уравнению $(A\,c - y)^{\mathrm{T}}\,A = 0$, которое эквивалентно (см. упражнение 28.1-2) уравнению $A^{\mathrm{T}}\,(A\,c - y) = 0$, откуда вытекает

$$A^{\mathrm{T}}A\,c = A^{\mathrm{T}}y. \tag{28.33}$$

В математической статистике такое уравнение называется ***нормальным уравнением*** (normal equation). Согласно упражнению 28.1-2, матрица $A^{\mathrm{T}}A$ симметрична, и если A имеет полный столбцовый ранг, то по теореме 28.6 матрица $A^{\mathrm{T}}A$ положительно определенная. Следовательно, существует обратная матрица $\left(A^{\mathrm{T}}A\right)^{-1}$, и решение уравнения (28.33) имеет вид

$$c = \left(\left(A^{\mathrm{T}}A\right)^{-1}A^{\mathrm{T}}\right)y = A^{+}y, \tag{28.34}$$

где

$$A^{+} = \left(\left(A^{\mathrm{T}}A\right)^{-1}A^{\mathrm{T}}\right)$$

псевдообратная (pseudoinverse) к A матрица. Псевдообратная матрица является естественным обобщением обратной матрицы для случая, когда исходная матрица не является квадратной (сравните уравнение (28.34), дающее приближенное решение уравнения $A\,c = y$ с точным решением $A^{-1}b$ уравнения $A\,x = b$).

В качестве примера рассмотрим пять экспериментальных точек

$$\begin{aligned}
(x_1, y_1) &= (-1, 2)\,,\\
(x_2, y_2) &= (1, 1)\,,\\
(x_3, y_3) &= (2, 1)\,,\\
(x_4, y_4) &= (3, 0)\,,\\
(x_5, y_5) &= (5, 3)\,,
\end{aligned}$$

которые показаны на рис. 28.3 черным цветом. Мы хотим найти приближение экспериментальных данных квадратичным полиномом $F(x) = c_1 + c_2x + c_3x^2$.

Начнем с матрицы значений базисных функций:

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix}\ ,$$

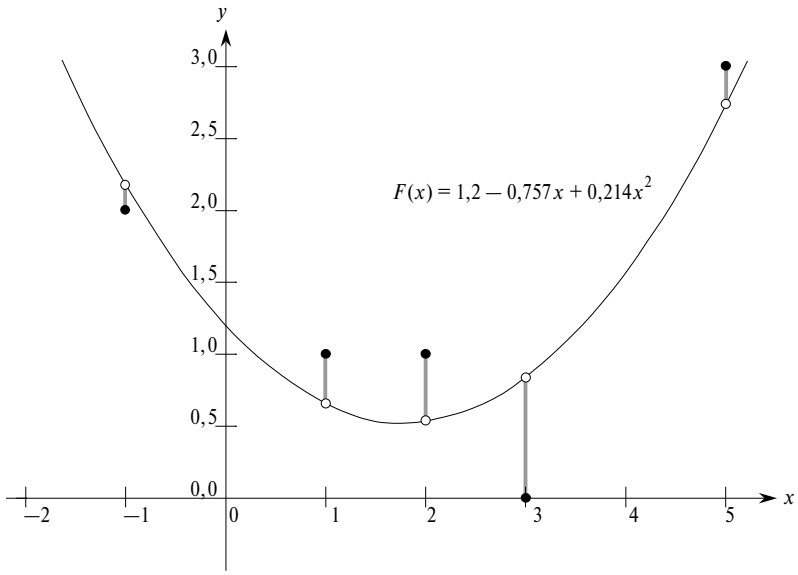


Рис. 28.3. Применение метода наименьших квадратов

Псевдообратная к ней матрица равна

$$A^+ = \begin{pmatrix} 0.500 & 0.300 & 0.200 & 0.100 & -0.100 \\ -0.388 & 0.093 & 0.190 & 0.193 & -0.088 \\ 0.060 & -0.036 & -0.048 & -0.036 & 0.060 \end{pmatrix} .$$

Умножая y на A^+, мы получаем вектор коэффициентов

$$c = \begin{pmatrix} 1.200 \\ -0.757 \\ 0.214 \end{pmatrix} ,$$

который дает нам квадратный полином

$$F(x) = 1.200 - 0.757x + 0.214x^2$$

в качестве наилучшего квадратичного приближения экспериментальных данных в смысле наименьших квадратов.

На практике нормальное уравнение (28.33) решается путем умножения $A^{\mathrm{T}}y$ с последующим поиском LU-разложения $A^{\mathrm{T}}A$. Если матрица A имеет полный ранг, то матрица $A^{\mathrm{T}}A$ гарантированно невырожденная, поскольку она симметричная и положительно определенная (см. упражнение 28.1-2 и теорему 28.6).

Упражнения

28.5-1. Докажите, что все диагональные элементы симметричной положительно определенной матрицы положительны.

28.5-2. Пусть $A = \left(\begin{smallmatrix} a & b \\ b & c \end{smallmatrix}\right)$ — симметричная положительно определенная матрица размером 2×2. Докажите, что ее определитель $ac - b^2$ положителен, выделив полный квадрат так, как это было сделано в доказательстве леммы 28.11.

28.5-3. Докажите, что максимальный элемент симметричной положительно определенной матрицы находится на ее диагонали.

28.5-4. Докажите, что определитель каждой главной подматрицы симметричной положительно определенной матрицы положителен.

28.5-5. Пусть A_k — k-я главная подматрица симметричной положительно определенной матрицы A. Докажите, что k-й ведущий элемент при LU-разложении равен $\det(A_k)/\det(A_{k-1})$ (полагаем $\det(A_0) = 1$).

28.5-6. Найдите функцию вида $F(x) = c_1 + c_2 x \lg x + c_3 e^x$, являющуюся наилучшим приближением в смысле наименьших квадратов экспериментальных данных $(1,1)$, $(2,1)$, $(3,3)$, $(4,8)$.

28.5-7. Покажите, что псевдообратная матрица A^+ обладает следующими свойствами:

$$\begin{aligned} AA^+A &= A\,, \\ A^+AA^+ &= A^+\,, \\ (AA^+)^{\mathrm{T}} &= AA^+\,, \\ (A^+A)^{\mathrm{T}} &= A^+A\,. \end{aligned}$$

Задачи

28-1. **Трехдиагональные системы линейных уравнений**

Рассмотрим трехдиагональную матрицу:

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}.$$

а) Найдите LU-разложение матрицы A.

б) Решите уравнение $Ax = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix}^{\mathrm{T}}$ с использованием прямой и обратной подстановки.

в) Найдите обратную к A матрицу.

г) Покажите, что для любой симметричной положительно определенной трехдиагональной матрицы A размером $n \times n$ и произвольного вектора b размером n уравнение $Ax = b$ можно решить при помощи LU-разложения за время $O(n)$. Покажите, что любой другой алгоритм, основанный на обращении матрицы A, имеет асимптотически большее время работы в худшем случае.

д) Покажите, что использование LUP-разложения позволяет решить уравнение $Ax = b$, где A — невырожденная трехдиагональная матрица размером $n \times n$, а b — произвольный вектор размером n, за время $O(n)$.

28-2. **Сплайны**

Один из практических методов интерполяции набора точек при помощи кривых заключается в использовании ***кубических сплайнов*** (cubic splines). Пусть задан набор $\{(x_i, y_i) : i = 0, 1, \ldots, n\}$ из $n+1$ точек, причем $x_0 < x_1 < \cdots < x_n$. Мы хотим провести через эти точки кусочно-кубическую кривую (сплайн) $f(x)$. Кривая $f(x)$ состоит из n кубических полиномов $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ ($i = 0, 1, \ldots, n-1$). Когда x находится в диапазоне $x_i \leqslant x \leqslant x_{i+1}$, значение кривой вычисляется как $f(x) = f_i(x - x_i)$. Точки x_i, в которых состыковываются кубические полиномы, называются ***узлами*** (knots). Для простоты будем считать, что $x_i = i$ при $i = 0, 1, \ldots, n-1$.

Чтобы обеспечить непрерывность $f(x)$, потребуем выполнения условий

$$\begin{aligned} f(x_i) &= f_i(0) = y_i \,, \\ f(x_{i+1}) &= f_i(1) = y_{i+1} \end{aligned}$$

для $i = 0, 1, \ldots, n-1$. Чтобы функция $f(x)$ была достаточно гладкой, мы также потребуем непрерывности первой производной в узлах:

$$f'(x_{i+1}) = f_i'(1) = f_{i+1}'(0)$$

для $i = 0, 1, \ldots, n-1$.

а) Предположим, что нам даны не только пары $\{(x_i, y_i)\}$, но и значения первых производных $D_i = f'(x_i)$ в каждой точке $i = 0, 1, \ldots, n$. Выразите коэффициенты a_i, b_i, c_i и d_i через y_i, y_{i+1}, D_i и D_{i+1} (напомним, что $x_i = i$). Сколько времени потребуется для вычисления $4n$ коэффициентов при таких условиях?

Остается вопрос о вычислении первой производной $f(x)$ в узлах. Один из методов их определения состоит в том, чтобы обеспечить непрерывность второй производной в узлах, так что

$$f''(x_{i+1}) = f_i''(1) = f_{i+1}''(0)$$

для $i = 0, 1, \ldots, n-1$. Мы полагаем, что в первом и последнем узлах $f''(x_0) = f_0''(0) = 0$ и $f''(x_n) = f_{n-1}''(1) = 0$. Это предположение дает нам ***естественный кубический сплайн*** (natural cubic spline).

б) Используя непрерывность второй производной, покажите, что для $i = 1, 2, \ldots, n-1$

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}). \tag{28.35}$$

в) Покажите, что

$$2D_0 + D_1 = 3(y_1 - y_0), \tag{28.36}$$

$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}). \tag{28.37}$$

г) Перепишите уравнения (28.35)–(28.37) в матричном виде с использованием вектора неизвестных $D = \langle D_0, D_1, \ldots, D_n \rangle$. Какими свойствами обладает матрица вашего уравнения?

д) Докажите, что множество из $n+1$ точек может быть интерполировано при помощи естественного кубического сплайна за время $O(n)$ (см. задачу 28-1).

е) Покажите, как построить естественный кубический сплайн, который интерполирует множество из $n+1$ точек (x_i, y_i), где $x_0 < x_1 < \cdots < x_n$ и x_i не обязательно равны i. Какое матричное уравнение надо для этого решить и сколько времени для этого потребуется?

Заключительные замечания

Имеется множество превосходных работ, в которых вопросы числовых и научных вычислений рассматриваются существенно более детально, чем в данной книге можем позволить себе мы. В особенности хочется порекомендовать следующие книги: Джордж (George) и Лью (Liu) [113], Голуб (Golub) и ван Лоан (Van Loan) [125], Пресс (Press), Фланнери (Flannery), Тукольски (Teukolsky) и Веттерлинг (Vetterling) [248, 249], Стренг (Strang) [285, 286].

В книге Голуба и ван Лоана [125] рассматриваются вопросы численной устойчивости. Они показали, почему $\det(A)$ не может служить хорошим показателем устойчивости матрицы A, предложив вместо него использовать величину

$\|A\|_\infty \left\|A^{-1}\right\|_\infty$, где $\|A\|_\infty = \max_{1 \leqslant i \leqslant n} \sum_{j=1}^{n} |a_{ij}|$. Кроме того, они рассмотрели также вопрос вычисления указанного значения без реального вычисления обратной матрицы A^{-1}.

Публикация алгоритма Штрассена в 1969 году [287] произвела сенсацию — до этого было трудно даже представить, что обычный алгоритм умножения матриц может быть улучшен. С того времени асимптотическая верхняя граница сложности умножения матриц была существенно улучшена. Асимптотически наиболее эффективный алгоритм умножения матриц размером $n \times n$ приведен в работе Копперсмита (Coppersmith) и Винограда (Winograd) [70] и имеет время работы $O\left(n^{2.376}\right)$. Графическое представление алгоритма Штрассена приведено в работе Патерсона (Paterson) [238].

Метод исключения неизвестных Гаусса, на котором основаны алгоритмы LU- и LUP-разложения, был первым систематическим методом решения систем линейных уравнений, и одним из первых численных алгоритмов. Хотя он был известен и ранее, его открытие обычно приписывают К. Ф. Гауссу (C. F. Gauss) (1777–1855).

В своей знаменитой работе [287] Штрассен также показал, что матрица размером $n \times n$ может быть обращена за время $O\left(n^{\lg 7}\right)$. Виноград [317] доказал, что умножение матриц не сложнее обращения, а обратная оценка была получена Ахо (Aho), Хопкрофтом (Hopcroft) и Ульманом (Ullman) [5].

Еще одним важным разложением матриц является ***сингулярное разложение*** (singular value decomposition, SVD). SVD-разложение матрицы A размером $m \times n$ представляет собой разложение $A = Q_1 \Sigma Q_2^{\mathrm{T}}$, где Σ — матрица размером $m \times n$, в которой ненулевые элементы находятся только на диагонали, Q_1 — матрица размером $m \times m$ со взаимно ортонормальными столбцами, а матрица Q_2 имеет размер $n \times n$ и также обладает свойством взаимной ортонормальности столбцов. Два вектора называются ***ортонормальными*** (orthonormal), если их скалярное произведение равно 0, а норма каждого вектора равна 1. Сингулярное разложение хорошо изложено в книгах Стренга [285, 286] и Голуба и ван Лоана [125].

Прекрасное изложение теории симметричных положительно определенных матриц (и линейной алгебры вообще) содержит книга Стренга [286].

ГЛАВА 29

Линейное программирование

Многие задачи можно сформулировать как задачи максимизации или минимизации некой цели в условиях ограниченности ресурсов и наличия конкурирующих ограничений. Если удастся задать цель в виде линейной функции нескольких переменных и сформулировать ограничения в виде равенств или неравенств, связывающих эти переменные, мы получим ***задачу линейного программирования*** (linear-programming problem). Задачи линейного программирования часто встречаются в разнообразных практических приложениях. Начнем их изучение на примере выработки предвыборной политики.

Политическая задача

Представьте себя на месте политика, стремящегося выиграть выборы. В вашем избирательном округе есть районы трех типов — городские, пригородные и сельские. В этих районах зарегистрировано, соответственно, 100 000, 200 000 и 50 000 избирателей. Чтобы добиться успеха, желательно получить большинство голосов в каждом из трех регионов. Вы — честный человек и никогда не будете давать обещания, в которые сами не верите. Однако вам известно, что определенные пункты программы могут помочь завоевать голоса определенных групп избирателей. Основными пунктами программы являются строительство дорог, контроль за использованием огнестрельного оружия, сельскохозяйственные субсидии и налог на бензин, направляемый на улучшение работы общественного транспорта. Согласно исследованиям вашего предвыборного штаба, можно оценить, сколько голосов будет приобретено или потеряно в каждой группе избирателей, если потратить 1 000 долл. на пропаганду по каждому пункту программы. Эта информация представлена в табл. 29.1. Каждый элемент данной таблицы

показывает, сколько тысяч избирателей из городских, пригородных или сельских районов удастся привлечь, потратив 1 000 долл. на агитацию в поддержку определенного пункта предвыборной программы. Отрицательные элементы отражают потерю голосов. Задача состоит в минимизации суммы, которая позволит завоевать 50 000 голосов горожан, 100 000 голосов избирателей из пригородных зон и 25 000 голосов сельских жителей.

Таблица 29.1. Влияние предвыборной политики на настроения избирателей

Пункт программы	Горожане	Жители пригорода	Сельские жители
Строительство дорог	-2	5	3
Контроль использования оружия	8	2	-5
Сельскохозяйственные субсидии	0	0	10
Налог на бензин	10	0	-2

Методом проб и ошибок можно выработать стратегию, которая позволит получить необходимое количество голосов, но затраты на такую стратегию могут оказаться не самыми низкими. Например, можно выделить 20 000 долл. на пропаганду строительства дорог, 0 долл. на агитацию в пользу контроля за использованием оружия, 4 000 долл. на пропаганду сельскохозяйственных субсидий и 9 000 долл. на агитацию за введение налога на бензин. При этом удастся привлечь $20 \cdot (-2) + 0 \cdot (8) + 4 \cdot (0) + 9 \cdot 10 = 50$ тысяч голосов горожан, $20 \cdot (5) + 0 \cdot (2) + 4 \cdot (0) + 9 \cdot 0 = 100$ тысяч голосов жителей пригородов и $20 \cdot (3) + 0 \cdot (-5) + 4 \cdot 10 + 9 \cdot (-2) = 82$ тысячи голосов сельских жителей. Таким образом, будет получено ровно необходимое количество голосов в городских и пригородных районах и большее количество голосов в сельской местности. (Получается, что в сельской местности привлечено голосов больше, чем наличное число избирателей!) Чтобы получить эти голоса, придется потратить на пропаганду $20+0+4+9 = 33$ тысячи долларов.

Возникает естественный вопрос: является ли данная стратегия наилучшей из возможных, т.е. можно ли достичь поставленных целей, потратив на пропаганду меньше денег? Ответ на этот вопрос можно получить, действуя методом проб и ошибок, однако вместо этого хотелось бы иметь некий систематический метод для ответа на подобные вопросы. Сформулируем данный вопрос математически. Введем 4 переменные:

- x_1 — сумма (в тысячах долларов), потраченная на пропаганду программы строительства дорог,
- x_2 — сумма (в тысячах долларов), потраченная на агитацию в пользу контроля за использованием оружия,
- x_3 — сумма (в тысячах долларов), потраченная на пропаганду программы сельскохозяйственных субсидий,

- x_4 — сумма (в тысячах долларов), потраченная на агитацию за введение налога на бензин.

Требование получить не менее 50 000 голосов избирателей-горожан можно записать в виде неравенства

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geqslant 50. \tag{29.1}$$

Аналогично, требования получить не менее 100 000 голосов избирателей, живущих в пригороде, и 25 000 голосов избирателей в сельской местности можно записать как неравенства

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geqslant 100 \tag{29.2}$$

и

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geqslant 25. \tag{29.3}$$

Любые значения переменных x_1, x_2, x_3, x_4, удовлетворяющие неравенствам (29.1)–(29.3), позволят получить достаточное количество голосов избирателей в каждом регионе. Чтобы сделать затраты минимально возможными, необходимо минимизировать сумму, затраченную на пропаганду, т.е. минимизировать выражение

$$x_1 + x_2 + x_3 + x_4. \tag{29.4}$$

Хотя отрицательная агитация часто встречается в политической борьбе, отрицательных затрат на пропаганду не бывает. Поэтому следует записать условия

$$x_1 \geqslant 0, x_2 \geqslant 0, x_3 \geqslant 0 \text{ и } x_4 \geqslant 0. \tag{29.5}$$

Поставив задачу минимизации выражения (29.4) при соблюдении неравенств (29.1)–(29.3) и (29.5), мы получаем так называемую “задачу линейного программирования”. Запишем ее следующим образом:

Минимизировать $x_1 + x_2 + x_3 + x_4$ (29.6)

при условиях

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \tag{29.7}$$

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \tag{29.8}$$

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \tag{29.9}$$

$$x_1, x_2, x_3, x_4 \geq 0\ . \tag{29.10}$$

Решение данной задачи линейного программирования позволит политику получить оптимальную стратегию предвыборной агитации.

Общий вид задач линейного программирования

В общем случае в задаче линейного программирования требуется оптимизировать некую линейную функцию при условии выполнения множества линейных неравенств. Для данных действительных чисел $a_1, a_2, \ldots, a_n$, и множества переменных $x_1, x_2, \ldots, x_n$ ***линейная функция*** этих переменных f определяется как

$$f(x_1, x_2, \ldots, x_n) = a_1 x_1 + a_2 x_2 + \cdots a_n x_n = \sum_{j=1}^{n} a_j x_j.$$

Если b — действительное число, а f — линейная функция, то выражение

$$f(x_1, x_2, \ldots, x_n) = b$$

называется ***линейным равенством***, а выражения

$$f(x_1, x_2, \ldots, x_n) \leqslant b$$

и

$$f(x_1, x_2, \ldots, x_n) \geqslant b$$

линейными неравенствами. Термин ***линейные ограничения*** применяется как к линейным равенствам, так и к линейным неравенствам. В линейном программировании не допускается использование строгих неравенств. Формально ***задача линейного программирования*** — это задача минимизации или максимизации линейной функции при соблюдении конечного множества линейных ограничений. Если выполняется минимизация, то такая задача называется ***задачей минимизации***, а если производится максимизация, то такая задача называется ***задачей максимизации***.

Вся оставшаяся часть данной главы будет посвящена формулированию и решению задач линейного программирования. Для решения задач линейного программирования существует несколько алгоритмов с полиномиальным временем выполнения, однако изучать их в данной главе мы не будем. Вместо этого мы рассмотрим самый старый алгоритм линейного программирования — симплекс-алгоритм. В худшем случае симплекс-алгоритм не выполняется за полиномиальное время, однако обычно он достаточно эффективен и широко используется на практике.

Краткий обзор задач линейного программирования

Чтобы описывать свойства задач линейного программирования и алгоритмы их решения, удобно договориться, в каких формах их записывать. В данной главе мы будем использовать две формы: ***стандартную*** и ***каноническую*** (slack).

Их строгое определение будет дано в разделе 29.1. Неформально, *стандартная* форма задачи линейного программирования — это задача максимизации линейной функции при соблюдении линейных *неравенств*, а *каноническая* форма — это задача максимизации линейной функции при соблюдении линейных *равенств*. Обычно мы будем использовать стандартную форму задач линейного программирования, однако при описании принципа работы симплекс-алгоритма удобней использовать каноническую форму. На данном этапе ограничимся рассмотрением задачи максимизации линейной функции n переменных при условии выполнения m линейных неравенств.

Рассмотрим следующую задачу линейного программирования с двумя переменными:

$$\text{Максимизировать } x_1 + x_2 \tag{29.11}$$

при условиях

$$4x_1 - x_2 \leqslant 8 \tag{29.12}$$

$$2x_1 + x_2 \leqslant 10 \tag{29.13}$$

$$5x_1 - 2x_2 \geqslant -2 \tag{29.14}$$

$$x_1, x_2 \geqslant 0 \tag{29.15}$$

Любой набор значений переменных x_1 и x_2, удовлетворяющий всем ограничениям (29.12)–(29.15), называется ***допустимым решением*** (feasible solution) данной задачи линейного программирования. Если изобразить эти ограничения в декартовой системе координат (x_1, x_2), как показано на рис. 29.1*a*, то множество допустимых решений (заштрихованная область на рисунке) образует выпуклую область[1] в двумерном пространстве. Эта выпуклая область называется ***допустимой областью*** (feasible region). Функция, которую мы хотим максимизировать, называется ***целевой функцией*** (objective function). Теоретически, можно было бы оценить значение целевой функции $x_1 + x_2$ в каждой точке допустимой области (значение целевой функции в определенной точке называется ***целевым значением*** (objective value)). Затем можно найти точку, в которой целевое значение максимально; она и будет оптимальным решением. В данном примере (как и в большинстве задач линейного программирования) допустимая область содержит бесконечное множество точек, поэтому хотелось бы найти способ, который позволит находить точку с максимальным целевым значением, не прибегая к вычислению значений целевой функции в каждой точке допустимой области.

В двумерном случае оптимальное решение можно найти с помощью графической процедуры. Множество точек, для которых $x_1 + x_2 = z$, при любом z

[1] Интуитивно выпуклая область определяется как область, удовлетворяющая тому требованию, чтобы для любых двух точек, принадлежащих области, все точки соединяющего их отрезка также принадлежали этой области.

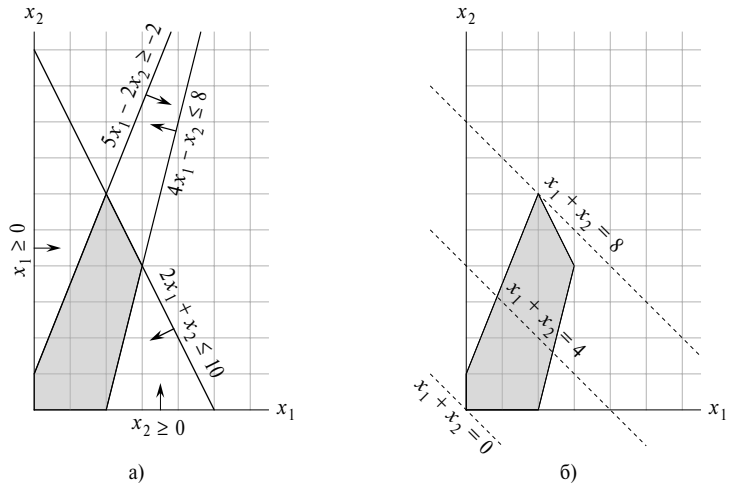


Рис. 29.1. а) Задача линейного программирования (29.12)–(29.15); б) Пунктирные линии показывают точки, в которых целевое значение равно 0, 4 и 8, соответственно

представляет собой прямую с коэффициентом наклона -1. Если мы построим график функции $x_1 + x_2 = 0$, получится прямая с коэффициентом наклона -1, проходящая через начало координат, как показано на рис. 29.1*б*. Пересечение данной прямой и допустимой области — это множество допустимых решений, целевое значение в которых равно 0. В данном случае пересечением прямой и допустимой области является точка $(0, 0)$. В общем случае для любого z пересечением прямой $x_1 + x_2 = z$ с допустимой областью является множество допустимых решений, в которых целевое значение равно z. На рис. 29.1*б* показаны прямые $x_1 + x_2 = 0$, $x_1 + x_2 = 4$ и $x_1 + x_2 = 8$. Поскольку допустимая область на рис. 29.1 ограничена, должно существовать некое максимальное значение z, для которого пересечение прямой $x_1 + x_2 = z$ и допустимой области является непустым множеством. Любая точка этого пересечения является оптимальным решением задачи линейного программирования; в данном случае такой точкой является $x_1 = 2$, $x_2 = 6$ с целевым значением 8. То, что оптимальное решение задачи линейного программирования оказалось в некоторой вершине допустимой области, не случайно. Максимальное значение z, при котором прямая $x_1 + x_2 = z$ пересекает допустимую область, должно находиться на границе допустимой области, поэтому пересечение данной прямой и границы допустимой области может быть либо вершиной, либо отрезком. Если пересечение является вершиной, то существует единственное оптимальное решение, находящееся в данной вершине. Если же пересечение является отрезком, то все точки этого отрезка имеют одинаковое целевое значение и являются оптимальными решениями; в частности,

оптимальными решениями являются оба конца отрезка. Каждый конец отрезка — это вершина, поэтому в данном случае также существует оптимальное решение в вершине допустимой области.

Несмотря на то, что для задач линейного программирования, где число переменных больше двух, простое графическое решение построить невозможно, наши интуитивные соображения остаются в силе. В случае трех переменных каждое ограничение описывается полупространством в трехмерном пространстве. Пересечение этих полупространств образует допустимую область. Множество точек, в которых целевая функция имеет значение z, представляет собой некую плоскость. Если все коэффициенты целевой функции неотрицательны и начало координат является допустимым решением рассматриваемой задачи линейного программирования, то при движении этой плоскости по направлению от начала координат получаются точки с возрастающими значениями целевой функции. (Если начало координат не является допустимым решением или некоторые коэффициенты целевой функции отрицательны, интуитивная картина будет немного сложнее.) Как и в двумерном случае, поскольку допустимая область выпукла, множество точек, в которых достигается оптимальное целевое значение, должно содержать вершину допустимой области. Аналогично, в случае n переменных каждое ограничение определяет полупространство в n-мерном пространстве. Допустимая область, образуемая пересечением этих полупространств, называется ***симплексом*** (simplex). Целевая функция в данном случае представляет собой гиперплоскость, и благодаря выпуклости допустимой области оптимальное решение находится в некой вершине симплекса.

Симплекс-алгоритм получает на входе задачу линейного программирования и возвращает оптимальное решение. Он начинает работу в некоторой вершине симплекса и выполняет последовательность итераций. В каждой итерации осуществляется переход вдоль ребра симплекса из текущей вершины в соседнюю, целевое значение в которой не меньше (как правило, больше), чем в текущей вершине. Симплекс-алгоритм завершается при достижении локального максимума, т.е. вершины, все соседние вершины которой имеют меньшее целевое значение. Поскольку допустимая область является выпуклой, а целевая функция линейна, локальный оптимум в действительности является глобальным. В разделе 29.4 мы воспользуемся понятием двойственности, чтобы показать, что решение, полученное с помощью симплекс-алгоритма, действительно оптимально.

Хотя геометрическое представление позволяет наглядно проиллюстрировать операции симплекс-алгоритма, мы не будем непосредственно обращаться к нему при подробном рассмотрении симплекс-метода в разделе 29.3. Вместо этого мы воспользуемся алгебраическим представлением. Сначала запишем задачу линейного программирования в канонической форме в виде набора линейных равенств. Эти линейные равенства выражают одни переменные, называемые *базисными*, через другие переменные, называемые *небазисными*. Переход от одной вершины

к другой осуществляется путем замены одной из базисных переменных на небазисную. Данная операция называется *замещением* и алгебраически заключается в переписывании задачи линейного программирования в эквивалентной канонической форме.

Приведенный выше пример с двумя переменными был исключительно простым. В данной главе нам предстоит рассмотреть и более сложные случаи: задачи линейного программирования, не имеющие решений; задачи, не имеющие конечного оптимального решения, и задачи, для которых начало координат не является допустимым решением.

Приложения линейного программирования

Линейное программирование имеет широкий спектр приложений. В любом учебнике по исследованию операций содержится множество примеров задач линейного программирования; линейное программирование — стандартный метод, который преподается студентам большинства школ бизнеса. Разработка сценария предвыборной борьбы — лишь один типичный пример. Приведем еще два примера, где с успехом используется линейное программирование.

- Авиакомпания составляет график работы экипажей. Федеральное авиационное агентство установило ряд ограничений, таких как ограничение времени непрерывной работы для каждого члена экипажа и требование, чтобы каждый конкретный экипаж работал только на самолетах одной модели на протяжении одного месяца. Авиакомпания хочет назначить экипажи на все рейсы, задействовав как можно меньше сотрудников (членов экипажей).
- Нефтяная компания выбирает место бурения скважины. С размещением буровой в каждом конкретном месте связаны определенные затраты и ожидаемая прибыль в виде некоторого количества баррелей добытой нефти, рассчитанная на основе проведенных геологических исследований. Средства, выделяемые на бурение новых скважин, ограничены, и компания хочет максимизировать ожидаемое количество добываемой нефти исходя из заданного бюджета.

Задачи линейного программирования также полезны при моделировании и решении задач теории графов и комбинаторных задач, таких как задачи, приведенные в данной книге. Мы уже встречались с частным случаем линейного программирования, который использовался для решения систем разностных ограничений в разделе 24.4. В разделе 29.2 мы покажем, как сформулировать в виде задач линейного программирования некоторые задачи теории графов и задачи сетевых потоков. В разделе 35.4 линейное программирование будет использоваться в качестве инструмента поиска приблизительного решения еще одной задачи теории графов.

Алгоритмы решения задач линейного программирования

В данной главе изучается симплекс-алгоритм. При правильном применении данный алгоритм на практике обычно позволяет быстро решать задачи линейного программирования общего вида. Однако при некоторых специально подобранных начальных значениях время выполнения симплекс-алгоритма может оказаться экспоненциальным. Первым алгоритмом с полиномиальным временем выполнения для решения задач линейного программирования был ***эллипсоидный алгоритм***, который на практике работает весьма медленно. Второй класс полиномиальных по времени алгоритмов содержит так называемые ***методы внутренней точки*** (interior-point methods). В отличие от симплекс-алгоритма, который движется по границе допустимой области и на каждой итерации рассматривает допустимое решение, являющееся вершиной симплекса, эти алгоритмы осуществляют движение по внутренней части допустимой области. Промежуточные решения являются допустимыми, но не обязательно находятся в вершинах симплекса, однако конечное решение является вершиной. Первый подобный алгоритм был разработан Кармаркаром (Karmarkar). Для задач большой размерности производительность алгоритмов внутренней точки может быть сравнима с производительностью симплекс-алгоритма, а иногда и превышает ее.

Если ввести в задачу линейного программирования дополнительное требование, чтобы все переменные принимали целые значения, получится задача ***целочисленного линейного программирования***. В упражнении 34.5-3 предлагается показать, что в этом случае даже задача поиска допустимого решения является NP-полной; поскольку ни для одной NP-полной задачи не известны полиномиальные алгоритмы решения, для задач целочисленного линейного программирования полиномиальные по времени алгоритмы также не известны. В отличие от них, задача линейного программирования общего вида может быть решена за полиномиальное время.

В данной главе для указания конкретного набора значений переменных в задаче линейного программирования с переменными $x = (x_1, x_2, \ldots, x_n)$ мы будем использовать обозначение $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$.

29.1 Стандартная и каноническая формы задач линейного программирования

В данном разделе описываются стандартная и каноническая формы задач линейного программирования, которые будут полезны при формулировании задач и работе с ними. В стандартной форме все ограничения являются неравенствами, а в канонической форме ограничения являются равенствами.

Стандартная форма

В стандартной форме заданы n действительных чисел $c_1, c_2, \ldots, c_n$; m действительных чисел $b_1, b_2, \ldots, b_m$; и mn действительных чисел a_{ij}, где $i = 1, 2, \ldots, m$ и $j = 1, 2, \ldots, n$. Требуется найти n действительных чисел $x_1, x_2, \ldots, x_n$, которые максимизируют

$$\sum_{j=1}^{n} c_j x_j \tag{29.16}$$

при условиях

$$\sum_{j=1}^{n} a_{ij} x_j \leqslant b_i, \quad \text{где } i = 1, 2, \ldots, m, \tag{29.17}$$

$$x_j \geqslant 0, \quad \text{где } j = 1, 2, \ldots, n. \tag{29.18}$$

Обобщая введенную для двумерной задачи линейного программирования терминологию, будем называть выражение (29.16) ***целевой функцией***, а $n + m$ неравенств (29.17) и (29.18) — ***ограничениями***; n ограничений (29.18) называются ***ограничениями неотрицательности***. Произвольная задача линейного программирования не обязательно содержит ограничения неотрицательности, но в стандартной форме они необходимы. Иногда удобно записывать задачу линейного программирования в более компактной форме. Определим $m \times n$-матрицу $A = (a_{ij})$, m-мерный вектор $b = (b_i)$, n-мерный вектор $c = (c_j)$ и n-мерный вектор $x = (x_j)$; тогда задачу линейного программирования (29.16)–(29.18) можно записать в следующем виде:

$$\text{Максимизировать } c^{\mathrm{T}} x \tag{29.19}$$

при условиях

$$Ax \leqslant b, \tag{29.20}$$

$$x \geqslant 0. \tag{29.21}$$

В выражении (29.19) $c^{\mathrm{T}} x$ — это скалярное произведение двух векторов. В выражении (19.20) Ax — произведение матрицы и вектора, а $x \geqslant 0$ в (29.21) означает, что все компоненты вектора x должны быть неотрицательными. Таким образом, задачу линейного программирования в стандартной форме можно описать с помощью тройки (A, b, c), и мы примем соглашение, что A, b, и c всегда имеют указанную выше размерность.

Теперь введем терминологию для описания различных ситуаций, возникающих в линейном программировании. Некоторые термины уже использовались в двумерной задаче. Набор значений переменных $\bar{x}$, которые удовлетворяют всем ограничениям, называется ***допустимым решением***, в то время как набор значений переменных $\bar{x}$, не удовлетворяющий хотя бы одному ограничению, называется ***недопустимым решением***. Решению $\bar{x}$ соответствует ***целевое значение*** $c^{\mathrm{T}} \bar{x}$.

Допустимое решение $\bar{x}$, целевое значение которого является максимальным среди всех допустимых решений, является ***оптимальным решением***, а его целевое значение $c^{\mathrm{T}}\bar{x}$ называется ***оптимальным целевым значением***. Если задача линейного программирования не имеет допустимых решений, она называется ***неразрешимой*** (infeasible), в противном случае она является ***разрешимой*** (feasible). Если задача линейного программирования имеет допустимые решения, но не имеет конечного оптимального целевого значения, она называется ***неограниченной*** (unbounded). В упражнении 29.1-9 предлагается показать, что задача линейного программирования может иметь конечное оптимальное целевое значение даже в том случае, когда ее допустимая область неограниченна.

Преобразование задач линейного программирования в стандартную форму

Любую задачу линейного программирования, в которой требуется минимизировать или максимизировать некую линейную функцию при наличии линейных ограничений, можно преобразовать в стандартную форму. Исходная задача может находиться не в стандартной форме по четырем причинам.

1. Целевая функция минимизируется, а не максимизируется.
2. На некоторые переменные не наложены условия неотрицательности.
3. Некоторые ограничения имеют форму равенств, т.е. имеют знак равенства вместо знака “меньше или равно”.
4. Некоторые ограничения-неравенства вместо знака “меньше или равно” имеют знак “больше или равно”.

Преобразовывая задачу линейного программирования L в другую задачу линейного программирования L', мы бы хотели, чтобы оптимальное решение задачи L' позволяло найти оптимальное решение задачи L. Будем говорить, что две задачи максимизации L и L' ***эквивалентны***, если для каждого допустимого решения $\bar{x}$ задачи L с целевым значением z существует соответствующее допустимое решение $\bar{x}'$ задачи L' с целевым значением z, а для каждого допустимого решения $\bar{x}'$ задачи L' с целевым значением z существует соответствующее допустимое решение $\bar{x}$ задачи L с целевым значением z. (Это определение не подразумевает взаимно однозначного соответствия между допустимыми решениями.) Задачи минимизации L и задача максимизации L' эквивалентны, если для каждого допустимого решения $\bar{x}$ задачи L с целевым значением z существует соответствующее допустимое решение $\bar{x}'$ задачи L' с целевым значением $-z$, а для каждого допустимого решения $\bar{x}'$ задачи L' с целевым значением z существует соответствующее допустимое решение $\bar{x}$ задачи L с целевым значением $-z$.

Теперь покажем, как поочередно избавиться от перечисленных выше проблем. После устранения каждого несоответствия стандартной форме докажем, что новая задача линейного программирования эквивалентна старой.

Чтобы превратить задачу минимизации L в эквивалентную ей задачу максимизации L', достаточно просто изменить знаки коэффициентов целевой функции на противоположные. Поскольку L и L' имеют одинаковые множества допустимых решений и для любого допустимого решения целевое значение L противоположно целевому значению L', эти две задачи линейного программирования эквивалентны. Например, пусть исходная задача имеет вид:

$$\begin{aligned} &\text{Минимизировать} && -2x_1 + 3x_2 \\ &\text{при условиях} \\ & && x_1 + \ x_2 = 7 \\ & && x_1 - 2x_2 \leqslant 4 \\ & && x_1 \qquad \geqslant 0 \end{aligned}$$

Если мы поменяем знаки коэффициентов целевой функции, то получим следующую задачу:

$$\begin{aligned} &\text{Максимизировать} && 2x_1 - 3x_2 \\ &\text{при условиях} \\ & && x_1 + \ x_2 = 7 \\ & && x_1 - 2x_2 \leqslant 4 \\ & && x_1 \qquad \geqslant 0 \end{aligned}$$

Теперь покажем, как преобразовать задачу линейного программирования, в которой на некоторые переменные не наложены ограничения неотрицательности, в задачу, где все переменные подчиняются этому условию. Предположим, что для некоторой переменной x_j ограничение неотрицательности отсутствует. Заменим все вхождения переменной x_j выражением $x'_j - x''_j$ и добавим ограничения неотрицательности $x'_j \geqslant 0$ и $x''_j \geqslant 0$. Так, если целевая функция содержит слагаемое $c_j x_j$, то оно заменяется на $c_j x'_j - c_j x''_j$, а если ограничение i содержит слагаемое $a_{ij} x_j$, оно заменяется на $a_{ij} x'_j - a_{ij} x''_j$. Любое допустимое решение $\bar{x}$ новой задачи линейного программирования соответствует допустимому решению исходной задачи с $\bar{x}_j = \bar{x}'_j - \bar{x}''_j$ и тем же самым целевым значением, следовательно, эти две задачи эквивалентны. Применив эту схему преобразования ко всем переменным, для которых нет ограничений неотрицательности, получим эквивалентную задачу линейного программирования, в которой на все переменные наложены ограничения неотрицательности.

Продолжая рассмотрение нашего примера, проверяем, для всех ли переменных есть соответствующие ограничения неотрицательности. Для переменной x_1 такое

ограничение есть, а для переменной x_2 — нет. Заменив переменную x_2 двумя переменными x_2' и x_2'' и выполнив соответствующие преобразования, получим следующую задачу:

$$\begin{aligned}&\text{Максимизировать } 2x_1 - 3x_2' + 3x_2''\\&\text{при условиях}\\&\qquad x_1 + x_2' - x_2'' = 7\\&\qquad x_1 - 2x_2' + 2x_2'' \leqslant 4\\&\qquad x_1, x_2', x_2'' \geqslant 0\end{aligned} \tag{29.22}$$

Теперь преобразуем ограничения-равенства в ограничения-неравенства. Предположим, что задача линейного программирования содержит ограничение-равенство $f(x_1, x_2, \ldots, x_n) = b$. Поскольку $x = y$ тогда и только тогда, когда справедливы оба неравенства $x \geqslant y$ и $x \leqslant y$, можно заменить данное ограничение-равенство парой ограничений-неравенств $f(x_1, x_2, \ldots, x_n) \leqslant b$ и $f(x_1, x_2, \ldots, x_n) \geqslant b$. Выполнив такое преобразование для всех ограничений-равенств, получим задачу линейного программирования, в которой все ограничения являются неравенствами.

Наконец, можно преобразовать ограничения вида "больше или равно" в ограничения вида "меньше или равно" путем умножения этих ограничений на -1. Любое ограничение вида

$$\sum_{j=1}^{n} a_{ij} x_j \geqslant b_i$$

эквивалентно ограничению

$$\sum_{j=1}^{n} -a_{ij} x_j \leqslant -b_i$$

Таким образом, заменив каждый коэффициент a_{ij} на $-a_{ij}$ и каждое значение b_j на $-b_j$, мы получим эквивалентное ограничение вида "меньше или равно".

Чтобы завершить преобразование нашего примера, заменим ограничение-равенство (29.22) на два неравенства, и получим:

$$\begin{aligned}&\text{Максимизировать } 2x_1 - 3x_2' + 3x_2''\\&\text{при условиях}\\&\qquad x_1 + x_2' - x_2'' \leqslant 7\\&\qquad x_1 + x_2' - x_2'' \geqslant 7\\&\qquad x_1 - 2x_2' + 2x_2'' \leqslant 4\\&\qquad x_1, x_2', x_2'' \geqslant 0\end{aligned} \tag{29.23}$$

Теперь изменим знак ограничения (29.23). Для единообразия имен переменных переименуем x_2' в x_2, и x_2'' в x_3. Полученная стандартная форма имеет вид:

$$\text{Максимизировать } 2x_1 - 3x_2 + 3x_3 \tag{29.24}$$

при условиях

$$x_1 + x_2 - x_3 \leqslant 7 \tag{29.25}$$

$$-x_1 - x_2 + x_3 \leqslant -7 \tag{29.26}$$

$$x_1 - 2x_2 + 2x_3 \leqslant 4 \tag{29.27}$$

$$x_1, x_2, x_3 \geqslant 0 \tag{29.28}$$

Преобразование задач линейного программирования в каноническую форму

Чтобы решать задачу линейного программирования с помощью симплекс-метода, удобно записать ее в такой форме, когда некоторые ограничения заданы в виде равенств. Говоря более точно, мы будем приводить задачу к форме, в которой только ограничения неотрицательности заданы в виде неравенств, а остальные ограничения являются равенствами. Пусть ограничение-неравенство имеет вид

$$\sum_{j=1}^{n} a_{ij} x_j \leqslant b_i. \tag{29.29}$$

Введем новую переменную s и перепишем неравенство (29.29) в виде двух ограничений:

$$s = b_i - \sum_{j=1}^{n} a_{ij} x_j, \tag{29.30}$$

$$s \geqslant 0. \tag{29.31}$$

Переменная s называется ***вспомогательной переменной*** (slack variable), она определяет ***разность*** (slack) между левой и правой частями выражения (29.29). Поскольку неравенство (29.29) верно тогда и только тогда, когда одновременно выполнены равенство (29.30) и неравенство (29.31), можно применить данное преобразование ко всем ограничениям-неравенствам задачи линейного программирования и получить эквивалентную задачу, в которой в виде неравенств записаны только условия неотрицательности. При переходе от стандартной формы к канонической мы будем использовать для связанной с i-м ограничением вспомогательной переменной обозначение x_{n+i} (вместо s). Тогда i-е ограничение будет записано

в виде равенства

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_i, \tag{29.32}$$

наряду с ограничением неотрицательности $x_{n+i} \geqslant 0$.

Применяя данное преобразование ко всем ограничениям задачи линейного программирования в стандартной форме, получаем задачу в канонической форме. Например, для задачи, заданной формулами (29.24)–(29.28), введя вспомогательные переменные x_4, x_5, x_6, получим:

$$\begin{aligned}
&\text{Максимизировать} && 2x_1 - 3x_2 + 3x_3 && (29.33)\\
&\text{при условиях} \\
& && x_4 = \quad 7 - \ x_1 - \ x_2 + \ x_3 && (29.34)\\
& && x_5 = -7 + \ x_1 + \ x_2 - \ x_3 && (29.35)\\
& && x_6 = \quad 4 - \ x_1 + 2x_2 - 2x_3 && (29.36)\\
& && x_1, x_2, x_3, x_4, x_5, x_6 \geqslant 0 && (29.37)
\end{aligned}$$

В этой задаче линейного программирования все ограничения, за исключением условий неотрицательности, являются равенствами, и все переменные подчиняются ограничениям неотрицательности. В записи каждого ограничения-равенства в левой части стоит одна переменная, а остальные переменные находятся в правой части. Более того, каждое уравнение содержит в правой части одни и те же переменные, и это только те переменные, которые входят в целевую функцию. Переменные, находящиеся в левой части равенств, называются ***базисными переменными*** (basic variables), а переменные, находящиеся в правой части, — ***небазисными переменными*** (nonbasic variables).

В задачах линейного программирования, удовлетворяющих данным условиям, мы иногда будем пропускать слова “максимизировать” и “при условии”, а также “ограничения неотрицательности”. Для обозначения значения целевой функции мы будем использовать переменную z. Полученная форма записи называется ***канонической формой*** (slack form). Каноническая форма задачи линейного программирования (29.33)–(29.37) выглядит следующим образом:

$$\begin{aligned}
z &= 2x_1 - 3x_2 + 3x_3 && (29.38)\\
x_4 &= 7 - \ x_1 - \ x_2 + \ x_3 && (29.39)\\
x_5 &= -7 + \ x_1 + \ x_2 - \ x_3 && (29.40)\\
x_6 &= 4 - \ x_1 + 2x_2 - 2x_3 && (29.41)
\end{aligned}$$

Для канонической формы, как и для стандартной, удобно иметь более короткий способ записи. Как будет показано в разделе 29.3, множества базисных и небазисных переменных будут меняться в процессе работы симплекс-алгоритма.

Обозначим множество индексов небазисных переменных N, а множество индексов базисных переменных — B. Всегда выполняются соотношения $|N| = n$, $|B| = m$ и $N \cup B = \{1, 2, \ldots, n+m\}$. Уравнения будут индексироваться элементами множества B, а переменные правых частей будут индексироваться элементами множества N. Как и в стандартной форме, мы используем b_j, c_j и a_{ij} для обозначения констант и коэффициентов. Для обозначения необязательного постоянного члена в целевой функции используется v. Таким образом, можно кратко описать каноническую форму с помощью кортежа (N, B, A, b, c, v); она служит кратким обозначением следующей канонической формы:

$$z = v + \sum_{j \in N} c_j x_j, \tag{29.42}$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j, \text{ где } i \in B. \tag{29.43}$$

При этом все переменные x подчиняются условиям неотрицательности. Поскольку сумма $\sum_{j \in N} a_{ij} x_j$ в выражении (29.43) вычитается, значения элементов a_{ij} противоположны коэффициентам, входящим в каноническую форму.

Например, для канонической формы

$$\begin{aligned}
z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\
x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\
x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\
x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2}
\end{aligned}$$

элементы шестерки имеют вид $B = \{1, 2, 4\}$, $N = \{3, 5, 6\}$, $v = 28$,

$$A = \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix},$$

$$b = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix},$$

$c = \begin{pmatrix} c_3 & c_5 & c_6 \end{pmatrix}^{\mathrm{T}} = \begin{pmatrix} -1/6 & -1/6 & -2/3 \end{pmatrix}^{\mathrm{T}}$, а индексы у A, b и c не обязательно являются множествами последовательных целых чисел; они зависят от того, какие индексы входят во множества B и N. В качестве примера противоположности элементов матрицы A коэффициентам канонической формы, заметим, что

в уравнение для x_1 входит слагаемое $x_3/6$, так что коэффициент a_{13} равен $-1/6$, а не $+1/6$.

Упражнения

29.1-1. Если записать задачу линейного программирования (29.24)–(29.28) в компактной форме (29.19)–(29.21), чему будут равны n, m, A, b и c?

29.1-2. Укажите три допустимых решения задачи линейного программирования (29.24)–(29.28). Чему равно целевое значение для каждого решения?

29.1-3. Какими будут N, B, A, b, c и v для канонической формы (29.38)–(29.41)?

29.1-4. Приведите следующую задачу линейного программирования к стандартной форме:

$$\begin{aligned}
&\text{Минимизировать} && 2x_1 + 7x_2 \\
&\text{при условиях} \\
& && x_1 = 7 \\
& && 3x_1 + x_2 \geqslant 24 \\
& && x_1 \geqslant 0 \\
& && x_2 \leqslant 0
\end{aligned}$$

29.1-5. Преобразуйте следующую задачу линейного программирования в каноническую форму:

$$\begin{aligned}
&\text{Максимизировать} && 2x_1 - 6x_3 \\
&\text{при условиях} \\
& && x_1 + x_2 - x_3 \leqslant 7 \\
& && 3x_1 - x_2 \geqslant 8 \\
& && -x_1 + 2x_2 + 2x_3 \geqslant 0 \\
& && x_1, x_2, x_3 \geqslant 0
\end{aligned}$$

Какие переменные являются базисными, а какие небазисными?

29.1-6. Покажите, что следующая задача линейного программирования является неразрешимой:

$$\begin{aligned}
&\text{Максимизировать} && 3x_1 - 2x_2 \\
&\text{при условиях} \\
& && x_1 + x_2 \leqslant 2 \\
& && -2x_1 - 2x_2 \leqslant -10 \\
& && x_1, x_2 \geqslant 0
\end{aligned}$$

29.1-7. Покажите, что следующая задача линейного программирования является неограниченной:

$$\begin{aligned} &\text{Максимизировать} && x_1 - x_2 \\ &\text{при условиях} \\ & && -2x_1 + x_2 \leqslant -1 \\ & && -x_1 - 2x_2 \leqslant -2 \\ & && x_1, x_2 \geqslant 0 \end{aligned}$$

29.1-8. Пусть имеется задача линейного программирования общего вида с n переменными и m ограничениями. Предположим, что мы преобразовали ее в стандартную форму. Укажите верхнюю границу числа переменных и ограничений в полученной задаче.

29.1-9. Приведите пример задачи линейного программирования, для которой допустимая область является неограниченной, но оптимальное целевое значение конечно.

29.2 Формулирование задач в виде задач линейного программирования

Хотя основное внимание в данной главе уделяется симплекс-алгоритму, важно также понимать, в каких случаях задачу можно сформулировать в виде задачи линейного программирования. После того как задача сформулирована в таком виде, ее можно решить за полиномиальное время с помощью эллипсоидного алгоритма, или алгоритма внутренней точки. Существует несколько пакетов прикладных программ, эффективно решающих задачи линейного программирования, а значит, если проблему удастся сформулировать в виде задачи линейного программирования, то ее можно решить на практике с помощью такого пакета.

Рассмотрим несколько конкретных примеров задач линейного программирования. Начнем с двух уже рассмотренных ранее задач: задачи поиска кратчайших путей из одной вершины (single source shortest parts problem) (см. главу 24) и задачи максимального потока (maximum-flow problem) (см. главу 26). После этого мы опишем задачу поиска потока с минимальными затратами (minimum-cost-flow). Для данной задачи существует алгоритм с полиномиальными затратами времени, не основанный на линейном программировании, однако мы не будем его рассматривать. И наконец, мы опишем задачу многопродуктового потока (multicomodity-flow problem), единственный известный полиномиальный по времени алгоритм решения которой базируется на линейном программировании.

Кратчайшие пути

Задачу поиска кратчайших путей из одной вершины-источника, описанную в главе 24, можно сформулировать в виде задачи линейного программирования. В данном разделе мы остановимся на формулировании задачи кратчайшего пути для одной пары источник–адресат, а более общий случай задачи поиска кратчайших путей из одного источника предлагается рассмотреть в качестве упражнения 29.2-3.

В задаче поиска кратчайшего пути для одной пары источник–адресат нам дан взвешенный ориентированный граф $G = (V, E)$, весовая функция $w : E \to \mathbf{R}$, ставящая в соответствие дугам графа действительные веса, исходная вершина s и конечная вершина t. Мы хотим вычислить значение $d[t]$, которое является весом кратчайшего пути из s в t. Чтобы записать эту задачу в виде задачи линейного программирования, необходимо определить множество переменных и ограничения, которые позволят определить, какой путь из s в t является кратчайшим. К счастью, алгоритм Беллмана-Форда именно для этого и предназначен. Когда алгоритм Беллмана-Форда завершает свою работу, для каждой вершины v оказывается вычисленным значение $d[v]$, такое что для каждой дуги $(u, v) \in E$ выполняется условие $d[v] \leqslant d[u] + w(u, v)$. Исходная вершина первоначально получает значение $d[s] = 0$, которое никогда не изменяется. Таким образом, мы получаем следующую задачу линейного программирования для вычисления веса кратчайшего пути из s в t:

Максимизировать $d[t]$ (29.44)

при условиях

$$d[v] \leqslant d[u] + w(u, v) \quad \text{для всех } (u, v) \in E, \tag{29.45}$$

$$d[s] = 0. \tag{29.46}$$

В этой задаче $|V|$ переменных $d[v]$, по одной для каждой вершины $v \in E$, и $|E| + 1$ ограничение, по одному для каждой дуги плюс дополнительное ограничение, состоящее в том, что исходной вершине всегда соответствует значение 0.

Максимальный поток

Задачу максимального потока также можно сформулировать в виде задачи линейного программирования. Напомним, что нам задан ориентированный граф $G = (V, E)$ в котором каждая дуга $(u, v) \in E$ имеет неотрицательную пропускную способность $c(u, v) \geqslant 0$, и две различные вершины, источник s и сток t. Согласно определению из раздела 26.1, поток — это действительная функция $f : V \times V \to \mathbf{R}$, обладающая тремя свойствами: она удовлетворяет ограничениям пропускной способности; меняет знак при изменении направления; обеспечивает сохранение потока. Максимальный поток —это поток, который удовлетворяет

данным ограничениям и максимизирует величину потока, представляющую собой суммарный поток, выходящий из источника. Таким образом, поток удовлетворяет линейным ограничениям, а величина потока является линейной функцией. Напомним также, что мы предполагаем, что $c(u,v) = 0$, если $(u,v) \notin E$. Теперь можно записать задачу максимизации потока в виде задачи линейного программирования:

$$\text{Максимизировать} \quad \sum_{v \in V} f(s,v) \tag{29.47}$$

при условиях

$$f(u,v) \leqslant c(u,v) \qquad \text{для всех } u,v \in V\ , \tag{29.48}$$

$$f(u,v) = -f(v,u) \qquad \text{для всех } u,v \in V\ , \tag{29.49}$$

$$\sum_{v \in V} f(u,v) = 0 \qquad \text{для всех } u \in V - \{s,t\}\ . \tag{29.50}$$

Эта задача линейного программирования содержит $|V|^2$ переменных, соответствующих потоку между каждой парой вершин, и $2|V|^2 + |V| - 2$ ограничений.

Обычно более удобно решать задачу линейного программирования меньшей размерности. Запись задачи (29.47)-(29.50), например, можно упростить, если учесть, что поток и пропускная способность каждой пары вершин u, v, таких что $(u,v) \notin E$, равны 0. Но более эффективно было бы переписать задачу так, чтобы в ней содержалось $O(V+E)$ ограничений, что и предлагается сделать в упражнении 29.2-5.

Поиск потока с минимальными затратами

До сих пор в данном разделе мы использовали линейное программирование для решения задач, для которых нам уже известны эффективные алгоритмы. Фактически, хороший алгоритм, разработанный специально для некой задачи, как, например, алгоритм Дейкстры (Dijkstra) для задачи поиска кратчайшего пути из одной вершины или метод проталкивания для задачи максимального потока, обычно более эффективен, чем линейное программирование, — как в теории, так и на практике.

Истинная ценность линейного программирования состоит в возможности решения новых задач. Вспомним задачу выработки предвыборной политики, описанную в начале данной главы. Задача получения необходимого количества голосов при минимальных затратах не решается ни одним из алгоритмов, уже изученных нами в данной книге, однако ее можно решить с помощью линейного программирования. О реальных задачах, которые можно решить с помощью линейного программирования, написано множество книг. Линейное программирование также чрезвычайно полезно при решении разновидностей задач, для которых еще не известны эффективные алгоритмы.

Рассмотрим, например, следующее обобщение задачи максимального потока. Предположим, что каждой дуге (u, v), помимо пропускной способности $c(u, v)$, соответствует некая стоимость $a(u, v)$, принимающая действительные значения. По техническим причинам мы требуем, чтобы в случае $a(u, v) > 0$ выполнялось соотношение $a(v, u) = 0$. Если по дуге (u, v) послано $f(u, v)$ единиц потока, это приводит к затратам $a(u, v) \cdot f(u, v)$. Нам также задано целевое значение потока d. Мы хотим переправить d единиц потока из s в t таким образом, чтобы общая стоимость, связанная с передачей потока, $\sum_{(u,v)\in E} a(u, v) \cdot f(u, v)$, была минимальной. Эта задача называется ***задачей поиска потока с минимальными затратами*** (minimum-cost-flow).

На рис. 29.2*а* представлен пример задачи поиска потока с минимальными затратами. Нам нужно переслать 4 единицы потока из s в t, минимизировав суммарные затраты (пропускная способность каждого ребра обозначена как c, а затраты передачи — как a). С любым допустимым потоком, т.е. функцией f, удовлетворяющей ограничениям (29.48)–(29.50), связаны затраты $\sum_{(u,v)\in E} a(u, v) \cdot f(u, v)$. Мы хотим найти такой поток, который минимизирует эти затраты. Оптимальное решение показано на рис. 29.2*б*; ему соответствуют суммарные затраты

$$\sum_{(u,v)\in E} a(u, v) \cdot f(u, v) = (2 \cdot 2) + (5 \cdot 2) + (3 \cdot 1) + (7 \cdot 1) + (1 \cdot 3) = 27.$$

Существуют алгоритмы с полиномиальными затратами времени, специально разработанные для задачи поиска потока с минимальными затратами, но в книге мы не будем их рассматривать. Запишем данную задачу в виде задачи линейного программирования. Эта задача линейного программирования напоминает задачу, записанную для максимизации потока, однако она содержит дополнительное ограничение, состоящее в том, что величина потока составляет ровно d единиц, и новую целевую функцию минимизации затрат:

$$\text{Минимизировать} \quad \sum_{(u,v)\in E} a(u, v) f(u, v) \tag{29.51}$$

при условиях

$$f(u, v) \leqslant c(u, v) \quad \text{для всех } u, v \in V\,, \tag{29.52}$$

$$f(u, v) = -f(v, u) \quad \text{для всех } u, v \in V\,, \tag{29.53}$$

$$\sum_{v\in V} f(u, v) = 0 \quad \text{для всех } u \in V - \{s, t\}\,, \tag{29.54}$$

$$\sum_{v\in E} f(s, v) = d\,. \tag{29.55}$$

Многопродуктовый поток

В качестве последнего примера рассмотрим еще одну задачу поиска потоков. Предположим, что компания Lucky Puck из раздела 26.1 приняла решение о расширении ассортимента продукции и отгружает не только хоккейные шайбы, но

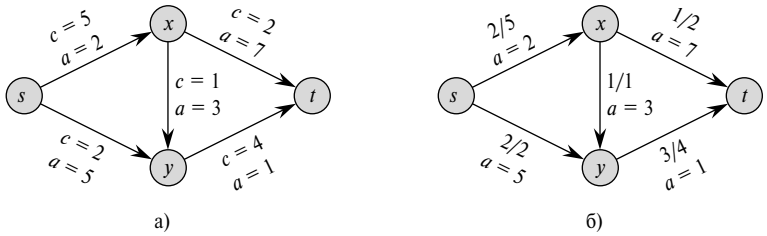


Рис. 29.2. Пример задачи поиска потока с минимальными затратами и ее решение

также клюшки и шлемы. Каждый элемент снаряжения выпускается на отдельной фабрике, хранится на отдельном складе и должен ежедневно доставляться с фабрики на склад. Клюшки производятся в Ванкувере и доставляются в Саскатун, а шлемы производятся в Эдмонтоне и доставляются в Реджину. Пропускная способность сети доставки не изменилась, и различные элементы, или ***продукты***, должны совместно использовать одну и ту же сеть.

Данный пример — образец задачи ***многопродуктового потока*** (multicommodity flow). В этой задаче снова задан ориентированный граф $G = (V, E)$, в котором каждая дуга $(u, v) \in E$ имеет неотрицательную пропускную способность $c(u, v) \geqslant 0$. Как и в задаче максимального потока, неявно подразумевается, что $c(u, v) = 0$ для $(u, v) \notin E$, а кроме того, требуется, чтобы из $c(u, v) > 0$ вытекало $c(v, u) = 0$. Кроме того, даны k различных продуктов $K_1, K_2, \ldots, K_k$, причем каждый i-й продукт характеризуется тройкой $K_i = (s_i, t_i, d_i)$, где s_i — источник продукта i, t_i — место назначения продукта i, а d_i — спрос, т.е. желаемая величина потока продукта i из s_i в t_i. По определению, поток продукта i, обозначаемый f_i (так что $f_i(u, v)$ — поток продукта i из вершины u в вершину v), — это действительная функция, удовлетворяющая ограничениям сохранения потока, смены знака при изменении направления и пропускной способности. Определим ***совокупный поток*** (aggregate flow) $f(u, v)$ как сумму потоков различных продуктов: $f(u, v) = \sum_{i=1}^{k} f_i(u, v)$. Совокупный поток по дуге (u, v) не должен превышать пропускную способность дуги (u, v). При таком способе описания задачи нам не нужно ничего минимизировать; необходимо только определить, возможно ли найти такой поток. Поэтому мы записываем задачу линейного программирования с “пустой” целевой функцией:

$$
\begin{array}{lll}
\text{Минимизировать} & 0 & \\
\text{при условиях} & \sum_{i=1}^{k} f_i(u,v) \leqslant c(u,v) & \text{для всех } u,v \in V, \\
& f_i(u,v) = -f_i(v,u) & \text{для всех } i = 1,2,\ldots,k \text{ и } u,v \in V, \\
& f_i(u,v) \leqslant c(u,v) & \text{для всех } i = 1,2,\ldots,k \text{ и } u,v \in V, \\
& \sum_{v \in V} f_i(u,v) = 0 & \text{для всех } i = 1,2,\ldots,k \\
& & \text{и } u \in V - \{s_i, t_i\}, \\
& \sum_{v \in V} f_i(s_i,v) = d_i & \text{для всех } i = 1,2,\ldots,k\,.
\end{array}
$$

Единственный известный алгоритм решения этой задачи с полиномиальным временем выполнения состоит в том, чтобы записать ее в виде задачи линейного программирования и затем решить с помощью полиномиального по времени алгоритма линейного программирования.

Упражнения

29.2-1. Приведите задачу линейного программирования поиска кратчайшего пути для одной пары вершин, заданную формулами (29.44)–(29.46), к стандартной форме.

29.2-2. Запишите в явном виде задачу линейного программирования, соответствующую задаче поиска кратчайшего пути из узла s в узел y на рис. 24.2a.

29.2-3. В задаче поиска кратчайшего пути из одной вершины мы хотим найти веса кратчайших путей из вершины s ко всем вершинам $v \in V$. Запишите задачу линейного программирования для данного графа G, решение которой обладает тем свойством, что $d[v]$ является весом кратчайшего пути из s в v для всех вершин $v \in V$.

29.2-4. Запишите задачу линейного программирования, соответствующую поиску максимального потока на рис. 26.1a.

29.2-5. Перепишите задачу линейного программирования для поиска максимального потока (29.47)–(29.50) так, чтобы в ней было только $O(V+E)$ ограничений.

29.2-6. Сформулируйте задачу линейного программирования, которая для заданного двудольного графа $G = (V,E)$ решает задачу о взвешенных паросочетаниях с максимальным весом (maximum-bipartite-matching).

29.2-7. В задаче поиска ***многопродуктового потока с минимальными затратами*** (minimum-cost multicommodity-flow problem) задан ориентированный граф $G = (V,E)$, в котором каждой дуге $(u,v) \in E$ соответствуют неотрицательная пропускная способность $c(u,v) \geqslant 0$ и затраты $a(u,v)$.

Как и в задаче многопродуктового потока, имеется k различных товаров $K_1, K_2, \ldots, K_k$, при этом каждый продукт i характеризуется тройкой $K_i = (s_i, t_i, d_i)$. Поток f_i для i-го продукта и совокупный поток $f(u, v)$ вдоль дуги (u, v) определяются так же, как и в задаче многопродуктового потока. Допустимым является такой поток, для которого совокупный поток вдоль каждой дуги (u, v) не превышает ее пропускной способности. Затраты для определенного потока определяются как $\sum_{u,v \in V} a(u, v) f(u, v)$, и цель состоит в том, чтобы найти допустимый поток с минимальными затратами. Запишите данную задачу в виде задачи линейного программирования.

29.3 Симплекс-алгоритм

Симплекс-алгоритм является классическим методом решения задач линейного программирования. В отличие от многих других приведенных в данной книге алгоритмов, время выполнения этого алгоритма в худшем случае не является полиномиальным. Однако он действительно выражает суть линейного программирования и на практике обычно бывает замечательно быстрым.

Симплекс-алгоритм имеет геометрическую интерпретацию, описанную ранее в данной главе, кроме того, можно провести определенные аналогии между ним и методом исключения Гаусса, рассмотренным в разделе 28.3. Метод исключения Гаусса применяется при поиске решения системы линейных уравнений. На каждой итерации система переписывается в эквивалентной форме, имеющей определенную структуру. После ряда итераций получается система, записанная в таком виде, что можно легко найти ее решение. Симплекс-алгоритм работает аналогичным образом, и его можно рассматривать как метод исключения Гаусса для неравенств.

Опишем основную идею, лежащую в основе итераций симплекс-алгоритма. С каждой итерацией связывается некое “базисное решение”, которое легко получить из канонической формы задачи линейного программирования: каждой небазисной переменной присваивается значение 0, и из ограничений-равенств вычисляются значения базисных переменных. Базисное решение всегда соответствует некой вершине симплекса. С алгебраической точки зрения каждая итерация преобразует одну каноническую форму в эквивалентную. Целевое значение, соответствующее новому базисному допустимому решению, должно быть не меньше (как правило, больше) целевого значения предыдущей итерации. Чтобы добиться этого увеличения, выбирается некоторая небазисная переменная, причем такая, что при увеличении ее значения целевое значение также увеличится. То, насколько можно увеличить данную переменную, определяется другими ограничениями; а именно: ее значение увеличивается до тех пор, пока какая-либо базисная переменная не станет равной 0. После этого каноническая форма переписывается так,

что эта базисная переменная и выбранная небазисная переменная меняются ролями. Хотя мы использовали определенные начальные значения переменных для описания алгоритма и будем использовать их в наших доказательствах, в алгоритме данное решение в явном виде не поддерживается. Он просто переписывает задачу линейного программирования до тех пор, пока оптимальное решение не станет “очевидным”.

Пример симплекс-алгоритма

Рассмотрим следующую задачу линейного программирования в стандартной форме:

$$\begin{aligned}
&\text{Максимизировать } 3x_1 + x_2 + 2x_3 && (29.56)\\
&\text{при условиях}\\
&\qquad x_1 + x_2 + 3x_3 \leqslant 30 && (29.57)\\
&\qquad 2x_1 + 2x_2 + 5x_3 \leqslant 24 && (29.58)\\
&\qquad 4x_1 + x_2 + 2x_3 \leqslant 36 && (29.59)\\
&\qquad x_1, x_2, x_3 \geqslant 0\ . && (29.60)
\end{aligned}$$

Чтобы можно было применить симплекс-алгоритм, необходимо преобразовать данную задачу в каноническую форму, как описано в разделе 29.1. Вспомогательные переменные — не просто элементы алгебраического преобразования, они являются содержательным алгоритмическим понятием. Напомним (см. раздел 29.1), что каждой переменной соответствует ограничение неотрицательности. Будем говорить, что ограничение-равенство является ***строгим*** (tight) при определенном наборе значений его небазисных переменных, если при этих значениях базисная переменная данного ограничения становится равной 0. Аналогично, набор значений небазисных переменных, который делает базисную переменную отрицательной, ***нарушает*** данное ограничение. Таким образом, вспомогательные переменные наглядно показывают, насколько каждое ограничение отличается от строгого, и тем самым помогают определить, насколько можно увеличить значения небазисных переменных, не нарушив ни одного ограничения.

Свяжем с ограничениями (29.57)–(29.59) вспомогательные переменные x_4, x_5 и x_6 соответственно, и приведем задачу линейного программирования к канонической форме. В результате получится следующая задача:

$$\begin{aligned}
z &= 3x_1 + x_2 + 2x_3 && (29.61)\\
x_4 &= 30 - x_1 - x_2 - 3x_3 && (29.62)\\
x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 && (29.63)\\
x_6 &= 36 - 4x_1 - x_2 - 2x_3 && (29.64)
\end{aligned}$$

Система ограничений (29.62)-(29.64) содержит 3 уравнения и 6 переменных. Любое задание значений переменных x_1, x_2 и x_3 определяет значения переменных x_4, x_5 и x_6, следовательно, существует бесконечное число решений данной системы уравнений. Решение является допустимым, если все $x_1, x_2, \ldots, x_6$ неотрицательны. Число допустимых решений также может быть бесконечным. Свойство бесконечности числа возможных решений подобной системы понадобится нам в дальнейших доказательствах. Рассмотрим ***базисное решение*** (basic solution): установим все (небазисные) переменные правой части равными 0 и вычислим значения (базисных) переменных левой части. В данном примере базисным решением является $(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_6) = (0, 0, 0, 30, 24, 36)$ и ему соответствует целевое значение $z = (3 \cdot 0) + (1 \cdot 0) + (2 \cdot 0) = 0$. Заметим, что в этом базисном решении $\bar{x}_i = b_i$ для всех $i \in B$. Итерация симплекс-алгоритма переписывает множество уравнений и целевую функцию так, что в правой части оказывается другое множество переменных. Таким образом, с переписанной задачей связано другое базисное решение. Мы подчеркиваем, что такая перезапись никоим образом не меняет лежащую в основе задачу линейного программирования; задача на каждой итерации имеет точно то же множество допустимых решений, что и задача на предыдущей итерации. Однако эта задача имеет базисное решение, отличное от базисного решения предыдущей итерации.

Если базисное решение является также допустимым, оно называется ***допустимым базисным решением***. В процессе работы симплекс-алгоритма базисное решение практически всегда будет допустимым базисным решением. Однако в разделе 29.5 мы покажем, что в нескольких первых итерациях симплекс-алгоритма базисное решение может не быть допустимым.

На каждой итерации нашей целью является переформулирование задачи линейного программирования таким образом, чтобы новое базисное решение имело большее целевое значение. Мы выбираем некоторую небазисную переменную x_e, коэффициент при которой в целевой функции положителен, и увеличиваем ее значение настолько, насколько это возможно без нарушения существующих ограничений. Переменная x_e становится базисной, а некоторая другая переменная x_l становится небазисной. Значения остальных базисных переменных и целевой функции также могут измениться.

Продолжая изучение примера, рассмотрим возможность увеличения значения x_1. При увеличении x_1 значения переменных x_4, x_5 и x_6 уменьшаются. Поскольку на каждую переменную наложено ограничение неотрицательности, ни одна из этих переменных не должна стать отрицательной. Если x_1 увеличить более, чем на 30, то x_4 станет отрицательной, а x_5 и x_6 станут отрицательными при увеличении x_1 на 12 и 9 соответственно. Третье ограничение (29.64) является самым строгим, именно оно определяет, на сколько можно увеличить x_1. Следовательно, поменяем

ролями переменные x_1 и x_6. Решим уравнение (29.64) относительно x_1 и получим

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \ . \tag{29.65}$$

Чтобы записать другие уравнения с x_6 в правой части, подставим вместо x_1 выражение из (29.65). Для уравнения (29.62) получаем

$$\begin{aligned} x_4 &= 30 - x_1 - x_2 - 3x_3 = \\ &= 30 - \left(9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}\right) - x_2 - 3x_3 = \\ &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \ . \end{aligned} \tag{29.66}$$

Аналогично поступаем с ограничением (29.63) и целевой функцией (29.61) и записываем нашу задачу линейного программирования в следующем виде:

$$z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \tag{29.67}$$

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \tag{29.68}$$

$$x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \tag{29.69}$$

$$x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2} \tag{29.70}$$

Эта операция называется ***замещением***. Как было показано выше, в процессе замещения выбираются небазисная переменная x_e, называемая ***вводимой переменной*** (entering variable), и базисная переменная x_l, называемая ***выводимой переменной*** (leaving variable), которые затем меняются ролями.

Задача, записанная уравнениями (29.67)–(29.70), эквивалентна задаче (29.61)–(29.64). В процессе работы симплекс-алгоритма производятся только операции переноса переменных из левой части уравнения в правую и наоборот, а также подстановки одного уравнения в другое. Первая операция, очевидно, создает эквивалентную задачу, то же можно сказать и о второй операции.

Чтобы продемонстрировать эквивалентность указанных задач, убедимся, что исходное базисное решение $(0, 0, 0, 30, 24, 36)$ удовлетворяет новым уравнениям (29.67)-(29.70) и имеет целевое значение $27 + (1/4) \cdot 0 + (1/2) \cdot 0 - (3/4) \cdot 36 = 0$. В базисном решении, связанном с новой задачей, новые небазисные переменные равны 0. Таким образом, оно имеет вид $(9, 0, 0, 21, 6, 0)$, а соответствующее целевое значение $z = 27$. Простые арифметические действия позволяют убедиться, что данное решение удовлетворяет уравнениям (29.62)–(29.64) и при подстановке в целевую функцию (29.61) имеет целевое значение $(3 \cdot 9) + (1 \cdot 0) + (2 \cdot 0) = 27$.

Продолжая рассмотрение примера, необходимо найти новую базисную переменную, значение которой можно увеличить. Нет смысла увеличивать x_6, поскольку при ее увеличении целевое значение уменьшается. Можно попробовать

увеличить x_2 или x_3; мы выберем x_3. Насколько можно увеличить x_3, чтобы не нарушить ни одно из ограничений? Ограничение (29.68) допускает увеличение, не превышающее 18, ограничение (29.69) — $42/5$, а ограничение (29.70) — $3/2$. Третье ограничение снова оказывается самым строгим, следовательно, мы переписываем его так, чтобы x_3 было в левой части, а x_5 — в правой части. Затем подставляем это новое уравнение в уравнения (29.67)-(29.69) и получаем новую эквивалентную задачу:

$$z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \tag{29.71}$$

$$x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \tag{29.72}$$

$$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \tag{29.73}$$

$$x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16} \tag{29.74}$$

С этой системой связано базисное решение $(33/4, 0, 3/2, 69/4, 0, 0)$ с целевым значением $111/4$. Теперь единственная возможность увеличить целевое значение — увеличить x_2. Имеющиеся ограничения задают верхние границы увеличения 132, 4 и ∞ соответственно (верхняя граница в ограничении (29.74) равна ∞, поскольку при увеличении x_2 значение базисной переменной x_4 также увеличивается. Следовательно, данное уравнение не налагает никаких ограничений на величину возможного увеличения x_2). Увеличиваем x_2 до 4 и делаем ее базисной. Затем решаем уравнение (29.73) относительно x_2, подставляем полученное выражения в другие уравнения и получаем новую задачу:

$$z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \tag{29.75}$$

$$x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \tag{29.76}$$

$$x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \tag{29.77}$$

$$x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \tag{29.78}$$

В полученной задаче все коэффициенты целевой функции отрицательны. Как будет показано далее, такая ситуация возникает только тогда, когда базисное решение переписанной задачи линейного программирования является оптимальным ее решением. Таким образом, для данной задачи решение $(8, 4, 0, 18, 0, 0)$ с целевым значением 28 является оптимальным. Теперь можно вернуться к исходной задаче линейного программирования, заданной уравнениями (29.56)–(29.60). Исходная задача содержит только переменные x_1, x_2 и x_3, поэтому оптимальное

решение имеет вид $x_1 = 8$, $x_2 = 4$, $x_3 = 0$, с целевым значением $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$. Заметим, что значения вспомогательных переменных в окончательном решении показывают, насколько велик резерв в каждом неравенстве. Вспомогательная переменная x_4 равна 18, а значение левой части в неравенстве (29.57) равно $8 + 4 + 0 = 12$, что на 18 меньше, чем значение правой части этого неравенства — 30. Вспомогательные переменные x_5 и x_6 равны 0, и действительно, в неравенствах (29.58) и (29.59) левые и правые части равны. Обратите внимание на то, что даже если коэффициенты исходной канонической формы являются целочисленными, коэффициенты последующих эквивалентных задач не обязательно целочисленны, и не обязательно целочисленны и промежуточные решения. Более того, окончательное решение задачи линейного программирования не обязательно будет целочисленным; в данном примере это просто совпадение.

Замещение

Формализуем процедуру замещения. Процедура PIVOT получает на входе каноническую форму задачи линейного программирования, заданную кортежем (N, B, A, b, c, v), индекс l выводимой переменной x_l и индекс e вводимой переменной x_e. Она возвращает кортеж $\left(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v}\right)$, описывающий новую каноническую форму. (Еще раз напомним, что элементы матриц A и $\widehat{A}$ являются числами, обратными коэффициентам канонической формы.)

PIVOT(N, B, A, b, c, v, l, e)
1 ▷ Вычисление коэффициентов уравнения
  для новой базисной переменной x_e
2 $\widehat{b}_e \leftarrow b_l / a_{le}$
3 **for** (для) всех $j \in N - \{e\}$
4 **do** $\widehat{a}_{ej} \leftarrow a_{lj} / a_{le}$
5 $\widehat{a}_{el} \leftarrow 1 / a_{le}$
6 ▷ Вычисление коэффициентов остальных ограничений
7 **for** (для) всех $i \in B - \{l\}$
8 **do** $\widehat{b}_i \leftarrow b_i - a_{ie}\widehat{b}_e$
9 **for** (для) всех $j \in N - \{e\}$
10 **do** $\widehat{a}_{ij} \leftarrow a_{ij} - a_{ie}\widehat{a}_{ej}$
11 $\widehat{a}_{il} \leftarrow -a_{ie}\widehat{a}_{el}$
12 ▷ Вычисление целевой функции
13 $\widehat{v} \leftarrow v + c_e\widehat{b}_e$
14 **for** (для) всех $j \in N - \{e\}$
15 **do** $\widehat{c}_j \leftarrow c_j - c_e\widehat{a}_{ej}$
16 $\widehat{c}_l \leftarrow -c_e\widehat{a}_{el}$

17 ▷ Вычисление новых множеств базисных
и небазисных переменных
18 $\widehat{N} \leftarrow N - \{e\} \cup \{l\}$
19 $\widehat{B} \leftarrow B - \{l\} \cup \{e\}$
20 **return** $\left(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v}\right)$

Процедура PIVOT работает следующим образом. В строках 2-5 вычисляются коэффициенты нового уравнения для x_e; для этого уравнение, в левой части которого стоит x_l, переписывается так, чтобы в левой части оказалась x_e. В строках 7–11 оставшиеся уравнения обновляются путем подстановки правой части полученного нового уравнения вместо всех вхождений x_e. В строках 13–16 такая же подстановка выполняется для целевой функции, а в строках 18 и 19 обновляются множества небазисных и базисных переменных. Строка 20 возвращает новую каноническую форму. Если $a_{le} = 0$, вызов процедуры PIVOT приведет к ошибке (деление на 0), однако, как будет показано в ходе доказательства лемм 29.2 и 29.12, данная процедура вызывается только тогда, когда $a_{le} \neq 0$.

Рассмотрим, как процедура PIVOT действует на значения переменных базисного решения.

Лемма 29.1. Рассмотрим вызов процедуры PIVOT(N, B, A, b, c, v, l, e) при $a_{le} \neq$ $\neq 0$. Пусть в результате вызова возвращаются значения $\left(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v}\right)$, и пусть $\bar{x}$ обозначает базисное решение после выполнения вызова. Тогда

1. $\bar{x}_j = 0$ для всех $j \in \widehat{N}$.
2. $\bar{x}_e = b_l / a_{le}$.
3. $\bar{x}_i = b_i - a_{ie}\widehat{b}_e$ для всех $i \in \widehat{B} - \{e\}$.

Доказательство. Первое утверждение верно, поскольку в базисном решении все небазисные переменные задаются равными 0. Если мы приравняем 0 все небазисные переменные в ограничении

$$x_i = \widehat{b}_i - \sum_{j \in \widehat{N}} \widehat{a}_{ij} x_j,$$

то получим, что $\bar{x}_i = \widehat{b}_i$ для всех $j \in \widehat{B}$. Поскольку $e \in \widehat{B}$, то, согласно строке 2 процедуры PIVOT, получаем

$$\bar{x}_e = \widehat{b}_e = b_l / a_{le},$$

что доказывает второе утверждение. Аналогично, используя строку 8 для всех $i \in \widehat{B} - \{e\}$, получаем

$$\bar{x}_i = \widehat{b}_i = b_i - a_{ie}\widehat{b}_e,$$

что доказывает третье утверждение. ■

Формальный симплекс-алгоритм

Теперь мы готовы формализовать симплекс-алгоритм, который уже продемонстрировали на примере. Этот пример был очень удачным, однако нам еще необходимо ответить на следующие вопросы.

- Как определить, что задача линейного программирования является разрешимой?
- Что делать, если задача линейного программирования является разрешимой, однако начальное базисное решение не является допустимым?
- Как определить, что задача линейного программирования является неограниченной?
- Как выбирать вводимую и выводимую переменные?

В разделе 29.5 мы покажем, как определить, является ли задача разрешимой и, в случае положительного ответа, как найти каноническую форму, в которой начальное базисное решение является допустимым. На данном этапе предположим, что у нас есть процедура INITIALIZE_SIMPLEX(A, b, c), которая получает на входе задачу линейного программирования в стандартной форме, т.е. матрицу $A = (a_{ij})$ размером $m \times n$, m-мерный вектор $b = (b_i)$ и n-мерный вектор $c = (c_i)$. Если задача неразрешима, процедура возвращает соответствующее сообщение и завершается. В противном случае она возвращает каноническую форму, начальное базисное решение которой является допустимым.

Процедура SIMPLEX получает на входе задачу линейного программирования в стандартной форме, описанной выше. Она возвращает n-мерный вектор $\bar{x} = (\bar{x}_j)$, который является оптимальным решением задачи линейного программирования, заданной формулами (29.19)–(29.21).

```
SIMPLEX(A, b, c)
 1  (N, B, A, b, c, v) ← INITIALIZE_SIMPLEX(A, b, c)
 2  while (пока) c_j > 0 для некоторого индекса j ∈ N
 3      do выбрать индекс e ∈ N, для которого c_e > 0
 4         for (для) каждого индекса i ∈ B
 5             do if a_ie > 0
 6                   then Δ_i ← b_i/a_ie
 7                   else Δ_i ← ∞
 8          Выбираем индекс l ∈ B, который минимизирует Δ_i
 9         if Δ_l = ∞
10           then return "Задача неограниченна"
11           else (N, B, A, b, c, v) ← PIVOT(N, B, A, b, c, v, l, e)
12  for i ← 1 to n
13      do if i ∈ B
14            then x̄_i ← b_i
```

```
15          else  x̄_i ← 0
16  return  (x̄_1, x̄_2, ..., x̄_n)
```

Процедура SIMPLEX работает следующим образом. В строке 1 выполняется вызов упомянутой выше процедуры INITIALIZE_SIMPLEX(A, b, c), которая или определяет, что предложенная задача неразрешима, или возвращает каноническую форму, базисное решение которой является допустимым. Главная часть алгоритма содержится в цикле **while** в строках 2–11. Если все коэффициенты целевой функции отрицательны, цикл **while** завершается. В противном случае в строке 3 мы выбираем в качестве вводимой переменной некоторую переменную x_e, коэффициент при которой в целевой функции положителен. Хотя в качестве вводимой можно выбирать любую такую переменную, предполагается, что используется некое предварительно заданное детерминистическое правило. Затем, в строках 4–8, производится проверка каждого ограничения и выбирается то, которое более всего лимитирует величину увеличения x_e, не приводящего к нарушению ограничений неотрицательности; базисная переменная, связанная с этим ограничением, выбирается в качестве x_l. Если таких переменных несколько, можно выбрать любую из них, однако предполагается, что и здесь мы используем некое предварительно заданное детерминистическое правило. Если ни одно из ограничений не лимитирует возможность увеличения вводимой переменной, алгоритм выдает сообщение “неограниченна” (строка 10). В противном случае, в строке 11 роли вводимой и выводимой переменных меняются путем вызова описанной выше процедуры PIVOT(N, B, A, b, c, v, l, e). В строках 12-15 вычисляется решение для переменных $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$ исходной задачи линейного программирования путем присваивания всем небазисным переменным значения 0, а всем базисным переменным $\bar{x}_i$ соответствующих значений b_i. В теореме 29.10 мы покажем, что это решение является оптимальным решением задачи линейного программирования. Наконец, в строке 16 возвращаются эти вычисленные значения переменных исходной задачи линейного программирования.

Чтобы показать, что процедура SIMPLEX работает корректно, сначала покажем, что если процедуре задано начальное допустимое значение и она завершилась, то при этом или возвращается допустимое решение, или сообщается, что данная задача является неограниченной. Затем покажем, что процедура SIMPLEX завершается; и наконец, в разделе 29.4 покажем, что возвращенное решение является оптимальным.

Лемма 29.2. Пусть задана задача линейного программирования (A, b, c); предположим, что вызываемая в строке 1 процедура INITIALIZE_SIMPLEX возвращает каноническую форму, базисное решение которой является допустимым. Тогда процедура SIMPLEX в строке 16 возвращает решение, которое является допустимым решением этой задачи линейного программирования. Если процедура SIMPLEX

возвращает в строке 10 сообщение “неограниченна”, данная задача линейного программирования является неограниченной.

Доказательство. Докажем следующий тройной инвариант цикла. В начале каждой итерации цикла **while** в строках 2–11

1. имеющаяся каноническая форма эквивалентна канонической форме, полученной в результате вызова процедуры INITIALIZE_SIMPLEX,
2. $b_i \geqslant 0$ для всех $i \in B$,
3. базисное решение, связанное с данной канонической формой, является допустимым.

Инициализация. Эквивалентность канонических форм для первой итерации очевидна. В формулировке леммы предполагается, что вызов процедуры INITIALIZE_SIMPLEX в строке 1 процедуры SIMPLEX возвращает каноническую форму, базисное решение которой является допустимым. Следовательно, третье утверждение справедливо. Далее, поскольку в базисном решении каждой базисной переменной x_i присваивается значение b_i, а допустимость базисного решения предполагает неотрицательность всех базисных переменных x_i, то $b_i \geqslant 0$. Таким образом, второе утверждение инварианта также справедливо.

Сохранение. Покажем, что данный инвариант цикла сохраняется при условии, что в строке 10 не выполняется оператор **return**. Случай выполнения строки 10 мы рассмотрим при обсуждении завершения цикла.

Каждая итерация цикла **while** меняет ролями некоторую базисную и небазисную переменные. При этом выполняются только операции решения уравнений и подстановки одного уравнения в другое, следовательно, новая каноническая форма эквивалентна канонической форме предыдущей итерации, которая, согласно инварианту цикла, эквивалентна исходной канонической форме.

Теперь покажем, что сохраняется вторая часть инварианта цикла. Предположим, что в начале каждой итерации цикла **while** $b_i \geqslant 0$ для всех $i \in B$, и покажем, что эти неравенства остаются верными после вызова процедуры PIVOT в строке 11. Поскольку изменения в переменные b_i и множество B вносятся только в этой строке, достаточно показать, что она сохраняет данную часть инварианта. Пусть b_i, a_{ij} и B обозначают значения перед вызовом процедуры PIVOT, а $\widehat{b}_i$ — значения, возвращаемые процедурой PIVOT.

Во-первых, заметим, что $\widehat{b}_e \geqslant 0$, поскольку $b_i \geqslant 0$ согласно инварианту цикла, $a_{le} > 0$ согласно строке 5 процедуры SIMPLEX, а $\widehat{b}_e = b_l / a_{le}$ согласно строке 2 процедуры PIVOT.

Для остальных индексов $i \in B - \{l\}$ мы имеем

$$\begin{aligned} \widehat{b}_i &= b_i - a_{ie}\widehat{b}_e = && \text{(согласно строке 8 процедуры PIVOT)} \\ &= b_i - a_{ie}\,(b_l/a_{le}) && \text{(согласно строке 2 процедуры PIVOT)}\,. \end{aligned} \tag{29.79}$$

Нам необходимо рассмотреть два случая: $a_{ie} > 0$ и $a_{ie} \leqslant 0$. Если $a_{ie} > 0$, то поскольку l выбирается так, что

$$b_l/a_{le} \leqslant b_i/a_{ie} \quad \text{для всех } i \in B, \tag{29.80}$$

получаем

$$\widehat{b}_i = b_i - a_{ie}\,(b_l/a_{le}) \geqslant b_i - a_{ie}\,(b_i/a_{ie}) = b_i - b_i = 0,$$

где первое равенство следует из уравнения (29.79), а неравенство — из неравенства (29.80). Таким образом, $\widehat{b}_i \geqslant 0$. Если же $a_{ie} \leqslant 0$, то поскольку все a_{le}, b_i и b_l неотрицательны, из уравнения (29.79) следует, что $\widehat{b}_i$ также должно быть неотрицательным.

Теперь докажем, что это базисное решение является допустимым, т.е. что все переменные имеют неотрицательные значения. Небазисные переменные устанавливаются равными 0 и, следовательно, являются неотрицательными. Каждая базисная переменная задается уравнением

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j.$$

В базисном решении $\bar{x}_i = b_i$. Используя вторую часть инварианта цикла, можно сделать вывод, что все базисные переменные $\bar{x}_i$ неотрицательны.

Завершение. Цикл **while** может завершиться одним из двух способов. Если его завершение связано с выполнением условия в строке 2, то текущее базисное решение является допустимым и это решение возвращается строкой 16. Второй способ завершения связан с возвращением сообщения “неограниченна” строкой 10. В этом случае в каждой итерации цикла **for** (строки 4–7) при выполнении строки 5 оказывается, что $a_{ie} \leqslant 0$. Пусть x — базисное решение, связанное с канонической формой в начале итерации, на которой возвращается сообщение “неограниченна”. Рассмотрим решение $\bar{x}$, определяемое как

$$\bar{x}_i = \begin{cases} \infty & \text{если } i = e, \\ 0 & \text{если } i \in N - \{e\}, \\ b_i - \sum_{j \in N} a_{ij}\bar{x}_j & \text{если } i \in B. \end{cases}$$

Покажем, что данное решение является допустимым, т.е. все переменные являются неотрицательными. Небазисные переменные, отличные от $\bar{x}_e$, равны

0, а значение $\bar{x}_e$ положительно, следовательно, все небазисные переменные неотрицательны. Все базисные переменные вычисляются по формуле

$$\bar{x}_i = b_i - \sum_{j \in N} a_{ij}\bar{x}_j = b_i - a_{ie}\bar{x}_e.$$

В соответствии с инвариантом цикла $b_i \geqslant 0$, кроме того, $a_{ie} \leqslant 0$ и $\bar{x}_e = \infty > 0$; следовательно, $\bar{x}_i \geqslant 0$.

Теперь покажем, что целевое значение этого решения $\bar{x}$ является неограниченным. Целевое значение равно

$$z = v + \sum_{j \in N} c_j\bar{x}_j = v + c_e\bar{x}_e.$$

Поскольку $c_e > 0$ (согласно строке 3) и $\bar{x}_e = \infty$, целевое значение бесконечно, следовательно, задача линейного программирования неограниченна. ■

На каждой итерации процедура SIMPLEX, помимо множеств N и B, поддерживает A, b, c и v. Хотя сохранение в явном виде значений A, b, c и v существенно для эффективной реализации симплкекс-алгоритма, оно не является необходимым. Иными словами, каноническая форма уникальным образом определяется множествами базисных и небазисных переменных. Прежде чем доказывать это утверждение, докажем полезную алгебраическую лемму.

Лемма 29.3. Пусть I — множество индексов. Пусть для каждого $i \in I$ α_i и β_i — действительные числа, а x_i — действительная переменная. Пусть также γ — некоторое действительное число. Предположим, что для любых x_i выполняется следующее условие:

$$\sum_{i \in I} \alpha_i x_i = \gamma + \sum_{i \in I} \beta_i x_i \ . \tag{29.81}$$

Тогда $\alpha_i = \beta_i$ для всех $i \in I$, а $\gamma = 0$.

Доказательство. Поскольку уравнение (29.81) выполняется для любых значений x_i, можно выбрать для них определенные значения, чтобы сделать заключения об α, β и γ. Выбрав $x_i = 0$ для всех $i \in I$, можно сделать вывод, что $\gamma = 0$. Теперь выберем произвольный индекс $i \in I$ и зададим $x_i = 1$ и $x_k = 0$ для всех $k \neq i$. В таком случае должно выполняться равенство $\alpha_i = \beta_i$. Поскольку индекс i выбирался из множества I произвольным образом, можно заключить, что $\alpha_i = \beta_i$ для всех $i \in I$. ■

Теперь покажем, что каноническая форма любой задачи линейного программирования уникальным образом определяется множеством базисных переменных.

Лемма 29.4. Пусть (A, b, c) — задача линейного программирования в стандартной форме. Задание множества базисных переменных B однозначно определяет соответствующую каноническую форму.

Доказательство. Будем проводить доказательство от противного. Предположим, что существуют две канонические формы с одинаковым множеством базисных переменных B. Эти канонические формы должны также иметь одинаковые множества небазисных переменных $N = \{1, 2, \ldots, n+m\} - B$. Запишем первую каноническую форму как

$$z = v + \sum_{j \in N} c_j x_j, \tag{29.82}$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{для } i \in B, \tag{29.83}$$

а вторую — как

$$z = v' + \sum_{j \in N} c'_j x_j, \tag{29.84}$$

$$x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{для } i \in B. \tag{29.85}$$

Рассмотрим систему уравнений, образованную путем вычитания каждого уравнения строки (29.85) из соответствующего уравнения строки (29.83). Полученная система имеет вид

$$0 = \left(b_i - b'_i\right) - \sum_{j \in N} \left(a_{ij} - a'_{ij}\right) x_j \quad \text{для } i \in B,$$

или, что эквивалентно,

$$\sum_{j \in N} a_{ij} x_j = \left(b_i - b'_i\right) + \sum_{j \in N} a'_{ij} x_j \quad \text{для } i \in B.$$

Теперь для всех $i \in B$ применим лемму 29.3, где $\alpha_i = a_{ij}$, $\beta_i = a'_{ij}$ и $\gamma = b_i - b'_i$. Поскольку $\alpha_i = \beta_i$, то $a_{ij} = a'_{ij}$ для всех $j \in N$, и поскольку $\gamma = 0$, то $b_i = b'_i$. Таким образом, в этих двух канонических формах матрицы A и A' и векторы b и b' идентичны. С помощью аналогичных рассуждений в упражнении 29.3-1 показывается, что в этом случае также $c = c'$ и $v = v'$; следовательно, рассматриваемые канонические формы идентичны. ■

Осталось показать, что процедура SIMPLEX заканчивается и, когда она завершается, возвращаемое решение является оптимальным. Вопрос оптимальности рассматривается в разделе 29.4. Обсудим завершение процедуры.

Завершение

В приведенном в начале данного раздела примере каждая итерация симплекс-алгоритма увеличивала целевое значение, связанное с базисным решением. В упражнении 29.3-2 предлагается показать, что ни одна итерация процедуры SIMPLEX не может уменьшить целевое значение, связанное с базисным решением. К сожалению, может оказаться, что итерация оставляет целевое значение неизменным. Это явление называется ***вырожденностью*** (degeneracy), и мы сейчас рассмотрим его более подробно.

Целевое значение изменяется в строке 13 процедуры PIVOT в результате присваивания $\widehat{v} \leftarrow v + c_e\widehat{b}_e$. Поскольку вызов процедуры PIVOT в процедуре SIMPLEX происходит только при $c_e > 0$, целевое значение может остаться неизменным (т.е., $\widehat{v} = v$) только в том случае, когда $\widehat{b}_e$ будет равно 0. Это значение вычисляется как $\widehat{b}_e \leftarrow b_l/a_{le}$ в строке 2 процедуры PIVOT. Поскольку процедура PIVOT вызывается только при $a_{le} \neq 0$, то для того, чтобы $\widehat{b}_e$ было равно 0 и, как следствие, целевое значение осталось неизменным, должно выполняться условие $b_l = 0$.

Такая ситуация действительно может возникнуть. Рассмотрим следующую задачу линейного программирования:

$$\begin{aligned} z &= x_1 + x_2 + x_3 \\ x_4 &= 8 - x_1 - x_2 \\ x_5 &= x_2 - x_3 \end{aligned}$$

Предположим, что в качестве вводимой переменной выбрана переменная x_1, а в качестве выводимой x_4. После замещения получим следующую задачу:

$$\begin{aligned} z &= 8 + x_3 - x_4 \\ x_1 &= 8 - x_2 - x_4 \\ x_5 &= x_2 - x_3 \end{aligned}$$

В данном случае единственная возможность замещения — когда вводимой переменной является x_3, а выводимой переменной x_5. Поскольку $b_5 = 0$, после замещения целевое значение 8 останется неизменным:

$$\begin{aligned} z &= 8 + x_2 - x_4 - x_5 \\ x_1 &= 8 - x_2 - x_4 \\ x_3 &= x_2 - x_5 \end{aligned}$$

Целевое значение осталось неизменным, но представление задачи изменилось. К счастью, если мы продолжим замещение, выбрав в качестве вводимой переменной x_2, а в качестве выводимой x_1, целевое значение увеличится и симплекс-алгоритм может продолжить свою работу.

Покажем, что вырожденность является единственным возможным препятствием для окончания симплекс-алгоритма. Вспомним о наших предположениях, что процедура SIMPLEX выбирает индексы e и l (в строках 3 и 8 соответственно) согласно некому детерминистическому правилу. Мы говорим, что процедура SIMPLEX ***зациклилась***, если канонические формы на двух различных итерациях идентичны; в этом случае, поскольку данная процедура является детерминистическим алгоритмом, она бесконечно будет перебирать одну и ту же последовательность канонических форм.

Лемма 29.5. Если процедура SIMPLEX не может завершиться более чем за $\binom{n+m}{m}$ итераций, она зацикливается.

Доказательство. Согласно лемме 29.4, множество базисных переменных B однозначным образом определяет каноническую форму. Всего имеется $n+m$ переменных, а $|B| = m$, так что существует $\binom{n+m}{m}$ способов выбрать B. Следовательно, всего имеется $\binom{n+m}{m}$ различных канонических форм. Поэтому, если процедура SIMPLEX совершает более чем $\binom{n+m}{m}$ итераций, она должна зациклиться. ■

Зацикливание теоретически возможно, но встречается чрезвычайно редко. Его можно избежать путем несколько более аккуратного выбора вводимых и выводимых переменных. Один из способов состоит в подаче на вход слабого возмущения, что приводит к невозможности получить два решения с одинаковым целевым значением. Второй способ состоит в лексикографическом разрыве связей, а третий — в разрыве связей путем выбора переменной с наименьшим индексом. Последняя стратегия известна как ***правило Бленда*** (Bland's rule). Мы не будем приводить доказательств, что эти стратегии позволяют избежать зацикливания.

Лемма 29.6. Если в строках 3 и 8 процедуры SIMPLEX выполняется выбор переменной с наименьшим индексом, процедура SIMPLEX должна завершиться. ■

Данный раздел мы закончим следующей леммой.

Лемма 29.7. Если процедура INITIALIZE_SIMPLEX возвращает каноническую форму, базисное решение которой является допустимым, то процедура SIMPLEX или выдает сообщение о неограниченности решения задачи линейного программирования, или завершается, предоставляя допустимое решение, не более чем за $\binom{n+m}{m}$ итераций.

Доказательство. В леммах 29.2 и 29.6 показано, что если процедура INITIALIZE_SIMPLEX возвращает каноническую форму, базисное решение которой является допустимым, то процедура SIMPLEX или выдает сообщение о неограниченности решения задачи линейного программирования, или завершается, предоставляя допустимое решение. Используя лемму 29.5, методом от противного приходим

к заключению, что если процедура SIMPLEX завершается предоставлением допустимого решения, то это происходит не более чем за $\binom{n+m}{m}$ итераций. ■

Упражнения

29.3-1. Завершите доказательство леммы 29.4, показав, что $c = c'$ и $v = v'$.

29.3-2. Покажите, что значение v никогда не уменьшается в результате вызова процедуры PIVOT в строке 11 процедуры SIMPLEX.

29.3-3. Предположим, что мы преобразовали задачу линейного программирования (A, b, c) из стандартной формы в каноническую. Покажите, что базисное решение является допустимым тогда и только тогда, когда $b_i \geqslant 0$ для $i = 1, 2, \ldots, m$.

29.3-4. Решите следующую задачу линейного программирования, используя процедуру SIMPLEX:

$$\begin{aligned} &\text{Максимизировать} && 18x_1 + 12.5x_2 \\ &\text{при условиях} \\ & && x_1 + x_2 \leqslant 20 \\ & && x_1 \leqslant 12 \\ & && x_2 \leqslant 16 \\ & && x_1, x_2 \geqslant 0 \end{aligned}$$

29.3-5. Решите следующую задачу линейного программирования, используя процедуру SIMPLEX:

$$\begin{aligned} &\text{Максимизировать} && -5x_1 - 3x_2 \\ &\text{при условиях} \\ & && x_1 - x_2 \leqslant 1 \\ & && 2x_1 + x_2 \leqslant 2 \\ & && x_1, x_2 \geqslant 0 \end{aligned}$$

29.3-6. Решите следующую задачу линейного программирования, используя процедуру SIMPLEX:

$$\begin{aligned} &\text{Минимизировать} && x_1 + x_2 + x_3 \\ &\text{при условиях} \\ & && 2x_1 + 7.5x_2 + 3x_3 \geqslant 10000 \\ & && 20x_1 + 5x_2 + 10x_3 \geqslant 30000 \\ & && x_1, x_2, x_3 \geqslant 0 \end{aligned}$$

29.4 Двойственность

Мы доказали, что при определенных предположениях процедура SIMPLEX завершается. Однако еще не доказано, что она действительно находит оптимальное решение задачи линейного программирования. Чтобы сделать это, введем новое мощное понятие — ***двойственность (дуальность) задач линейного программирования*** (linear-programming duality).

Двойственность — очень важное свойство. В задачах оптимизации определение двойственной задачи практически всегда сопровождается открытием алгоритма с полиномиальным временем выполнения. Двойственность также является очень мощным средством при доказательстве того, что решение действительно является оптимальным.

Предположим, например, что в некой задаче максимального потока мы нашли поток f величиной $|f|$. Как выяснить, является ли f максимальным потоком? Согласно теореме 26.7, если мы сможем найти разрез, значение которого также равно $|f|$, то f действительно является максимальным потоком. Это пример двойственности: для задачи максимизации определяется связанная с ней задача минимизации, причем такая, что эти две задачи имеют одно и то же оптимальное значение.

Опишем, как для заданной задачи линейного программирования, в которой требуется максимизировать целевую функцию, сформулировать ***двойственную*** (dual) задачу линейного программирования, в которой целевую функцию требуется минимизировать и оптимальное значение которой идентично оптимальному значению исходной задачи. При работе с двойственными задачами исходная задача называется ***прямой*** (primal).

Для задачи линейного программирования в стандартной форме, такой как (29.16)–(29.18), определим двойственную задачу следующим образом:

$$\text{Минимизировать} \quad \sum_{i=1}^{m} b_i y_i \tag{29.86}$$

$$\text{при условиях} \quad \sum_{i=1}^{m} a_{ij} y_i \geqslant c_j \quad \text{для } j = 1, 2, \ldots, n, \tag{29.87}$$

$$y_i \geqslant 0 \quad \text{для } i = 1, 2, \ldots, m. \tag{29.88}$$

Чтобы получить двойственную задачу, мы изменили максимизацию на минимизацию, поменяли роли коэффициентов правых частей и целевой функции и заменили неравенства “меньше или равно” на “больше или равно”. С каждым из m ограничений прямой задачи в двойственной задаче связана переменная y_i, а с каждым из n ограничений двойственной задачи связана переменная прямой задачи x_j. Например, рассмотрим задачу линейного программирования, задан-

ную уравнениями (29.56)–(29.60). Двойственная ей задача выглядит следующим образом:

$$\begin{aligned}
&\text{Минимизировать} \quad 30y_1 + 24y_2 + 36y_3 && (29.89)\\
&\text{при условиях}\\
&\qquad y_1 + \ 2y_2 + \ 4y_3 \geqslant 3 && (29.90)\\
&\qquad y_1 + \ 2y_2 + \ \ y_3 \geqslant 1 && (29.91)\\
&\qquad 3y_1 + \ 5y_2 + \ 2y_3 \geqslant 2 && (29.92)\\
&\qquad y_1, y_2, y_3 \geqslant 0 && (29.93)
\end{aligned}$$

В теореме 29.10 мы покажем, что оптимальное значение двойственной задачи линейного программирования всегда равно оптимальному значению прямой задачи. Более того, симплекс-алгоритм неявно решает одновременно обе задачи, прямую и двойственную, тем самым обеспечивая доказательство оптимальности.

Начнем с доказательства ***слабой двойственности*** (weak duality), которая состоит в утверждении, что любое допустимое решение прямой задачи линейного программирования имеет целевое значение, не превышающее целевого значения любого допустимого решения двойственной задачи линейного программирования.

Лемма 29.8 (Слабая двойственность задач линейного программирования). Пусть $\bar{x}$ — некое допустимое решение прямой задачи линейного программирования (29.16)–(29.18), а $\bar{y}$ — некое допустимое решение двойственной задачи (29.86)–(29.88). Тогда

$$\sum_{j=1}^{n} c_j \bar{x}_j \leqslant \sum_{i=1}^{m} b_i \bar{y}_i.$$

Доказательство.

$$\sum_{j=1}^{n} c_j \bar{x}_j \leqslant \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x_j} = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} \bar{x}_j \right) y_i \leqslant \sum_{i=1}^{m} b_i \bar{y}_i,$$

где первое неравенство следует из (29.87), а второе — из (29.17). ■

Следствие 29.9. Пусть $\bar{x}$ — некоторое допустимое решение прямой задачи линейного программирования (A, b, c), и пусть $\bar{y}$ — некоторое допустимое решение соответствующей двойственной задачи. Если

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i,$$

то $\bar{x}$ и $\bar{y}$ являются оптимальными решениями прямой и двойственной задач соответственно.

Доказательство. Согласно лемме 29.8, целевое значение допустимого решения прямой задачи не превышает целевого значения допустимого решения двойственной задачи. Прямая задача является задачей максимизации, а двойственная — задачей минимизации. Поэтому если допустимые решения $\bar{x}$ и $\bar{y}$ имеют одинаковое целевое значение, ни одно из них невозможно улучшить. ■

Прежде чем доказать, что всегда существует решение двойственной задачи, целевое значение которого равно целевому значению оптимального решения прямой задачи, покажем, как найти такое решение. При решении задачи линейного программирования (29.56)–(29.60) с помощью симплекс-алгоритма в результате последней итерации была получена каноническая форма (29.75)–(29.78), в которой $B = \{1, 2, 4\}$, $N = \{3, 5, 6\}$. Как будет показано ниже, базисное решение, связанное с этой последней канонической формой, является оптимальным решением задачи линейного программирования; таким образом, оптимальное решение задачи (29.56)–(29.60) — $(\bar{x}_1, \bar{x}_2, \bar{x}_3) = (8, 4, 0)$ с целевым значением 28. Мы также покажем, что можно легко получить оптимальное решение двойственной задачи: оптимальные значения двойственных переменных противоположны коэффициентам целевой функции прямой задачи. Более строго, предположим, что последняя каноническая форма прямой задачи имеет вид:

$$z = v' + \sum_{j \in N} c'_j x_j$$
$$x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{для } i \in B.$$

Тогда оптимальное решение двойственной задачи можно найти следующим образом:

$$\bar{y}_i = \begin{cases} -c'_{n+i} & \text{если } (n+i) \in N, \\ 0 & \text{в противном случае.} \end{cases} \tag{29.94}$$

Таким образом, оптимальным решением двойственной задачи линейного программирования (29.89)–(29.93) является $\bar{y}_1 = 0$ (поскольку $n+1 = 4 \in B$), $\bar{y}_2 = -c'_5 = = 1/6$ и $\bar{y}_3 = -c'_6 = 2/3$. Вычисляя значение целевой функции двойственной задачи (29.89), получаем целевое значение $(30 \cdot 0) + (24 \cdot (1/6)) + (36 \cdot (2/3)) = 28$; это подтверждает, что целевое значение прямой задачи действительно равно целевому значению двойственной задачи. Используя эти вычисления и лемму 29.8, получаем доказательство, что оптимальное целевое значение прямой задачи линейного программирования равно 28. Теперь покажем, что в общем случае оптимальное решение двойственной задачи и доказательство оптимальности решения прямой задачи можно получить таким способом.

Теорема 29.10 (Двойственность задач линейного программирования). Предположим, что процедура SIMPLEX возвращает значения $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$ для прямой задачи линейного программирования (A, b, c). Пусть N и B — множества небазисных и базисных переменных окончательной канонической формы, c' — ее коэффициенты, а $\bar{y} = (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$ определяется уравнением (29.94). Тогда $\bar{x}$ — оптимальное решение прямой задачи линейного программирования, $\bar{y}$ — оптимальное решение двойственной задачи и

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i. \tag{29.95}$$

Доказательство. Согласно следствию 29.9, если нам удастся найти допустимые решения $\bar{x}$ и $\bar{y}$, которые удовлетворяют уравнению (29.95), то $\bar{x}$ и $\bar{y}$ должны быть оптимальными решениями прямой и двойственной задач. Покажем, что решения $\bar{x}$ и $\bar{y}$, описанные в формулировке теоремы, удовлетворяют уравнению (29.95).

Предположим, что мы решаем прямую задачу линейного программирования (29.16)–(29.18) с помощью процедуры SIMPLEX. В ходе работы алгоритма строится последовательность канонических форм, пока он не завершится, предоставив окончательную каноническую форму с целевой функцией

$$z = v' + \sum_{j \in N} c'_j x_j. \tag{29.96}$$

Поскольку процедура SIMPLEX завершается с предоставлением решения, то, согласно условию в строке 2, мы знаем, что

$$c'_j \leqslant 0 \text{ для всех } j \in N. \tag{29.97}$$

Если мы определим

$$c'_j = 0 \text{ для всех } j \in B, \tag{29.98}$$

то уравнение (29.96) можно переписать так:

$$\begin{aligned} z &= v' + \sum_{j \in N} c'_j x_j = \\ &= v' + \sum_{j \in N} c'_j x_j + \sum_{j \in B} c'_j x_j = \quad \text{(поскольку } c'_j = 0 \text{ для } j \in B\text{)} \\ &= v' + \sum_{j=1}^{n+m} c'_j x_j \qquad\qquad \text{(поскольку } N \cup B = \{1, 2, \ldots, n+m\}\text{)} \end{aligned} \tag{29.99}$$

В базисном решении $\bar{x}$, связанном с конечной канонической формой, $\bar{x}_j = 0$ для всех $j \in N$, и $z = v'$. Поскольку все канонические формы эквивалентны,

при вычислении значения исходной целевой функции для решения $\bar{x}$ мы должны получить то же самое целевое значение, т.е.

$$\sum_{j=1}^{n} c_j \bar{x}_j = v' + \sum_{j=1}^{n+m} c'_j \bar{x}_j = \tag{29.100}$$

$$= v' + \sum_{j \in N} c'_j \bar{x}_j + \sum_{j \in B} c'_j \bar{x}_j =$$

$$= v' + \sum_{j \in N} \left(c'_j \cdot 0\right) + \sum_{j \in B} \left(0 \cdot \bar{x}_j\right) = \tag{29.101}$$

$$= v'.$$

Теперь мы покажем, что решение $\bar{y}$, заданное формулой (29.94), является допустимым решением двойственной задачи и его целевое значение $\sum_{i=1}^{m} b_i \bar{y}_i$ равно $\sum_{j=1}^{n} c_j \bar{x}_j$. Из уравнения (29.100) следует, что значения целевых функций первой и последней канонических форм, вычисленные для $\bar{x}$, равны. В общем случае, эквивалентность всех канонических форм подразумевает, что для *любого* набора значений $x = (x_1, x_2, \ldots, x_n)$ справедливо равенство

$$\sum_{j=1}^{n} c_j x_j = v' + \sum_{j=1}^{n+m} c'_j x_j.$$

Следовательно, для любого набора значений x мы имеем:

$$\sum_{j=1}^{n} c_j x_j =$$

$$= v' + \sum_{j=1}^{n+m} c'_j x_j =$$

$$= v' + \sum_{j=1}^{n} c'_j x_j + \sum_{j=n+1}^{n+m} c'_j x_j =$$

$$= v' + \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} c'_{n+i} x_{n+i} =$$

$$= v' + \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} \left(-\bar{y}_i\right) x_{n+i} = \quad \text{(в соответствии с (29.94))}$$

$$= v' + \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} \left(-\bar{y}_i\right) \left(b_i - \sum_{j=1}^{n} a_{ij} x_j\right) = \quad \text{(в соответствии с (29.32))}$$

$$
\begin{aligned}
&= v' + \sum_{j=1}^{n} c'_j x_j - \sum_{i=1}^{m} b_i \bar{y}_i + \sum_{i=1}^{m} \sum_{j=1}^{n} (a_{ij} x_j)\, \bar{y}_i = \\
&= v' + \sum_{j=1}^{n} c'_j x_j - \sum_{i=1}^{m} b_i \bar{y}_i + \sum_{j=1}^{n} \sum_{i=1}^{m} (a_{ij} \bar{y}_i)\, x_j = \\
&= \left(v' - \sum_{i=1}^{m} b_i \bar{y}_i \right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i \right) x_j \ ,
\end{aligned}
$$

так что

$$
\sum_{j=1}^{n} c_j x_j = \left(v' - \sum_{i=1}^{m} b_i \bar{y}_i \right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i \right) x_j \ . \tag{29.102}
$$

Применяя лемму 29.3 к уравнению (29.102), получаем

$$
v' - \sum_{i=1}^{m} b_i \bar{y}_i = 0, \tag{29.103}
$$

$$
c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i = c_j \quad \text{для } j = 1, 2, \ldots, n. \tag{29.104}
$$

Из уравнения (29.103) следует, что $\sum_{i=1}^{m} b_i \bar{y}_i = v'$, поэтому целевое значение двойственной задачи ($\sum_{i=1}^{m} b_i \bar{y}_i$) равно целевому значению прямой задачи (v'). Осталось показать, что $\bar{y}$ является допустимым решением двойственной задачи. Из (29.97) и (29.98) следует, что $c'_j \leqslant 0$ для всех $j = 1, 2, \ldots, n+m$. Поэтому для любого $i = 1, 2, \ldots, m$ из (29.104) вытекает, что

$$
c_j = c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i \leqslant \sum_{i=1}^{m} a_{ij} \bar{y}_i,
$$

что удовлетворяет ограничениям (29.87) двойственной задачи. Наконец, поскольку $c'_j \leqslant 0$ для всех $j \in N \cup B$, то, задав $\bar{y}$ в соответствии с уравнением (29.94), мы получим, что все $\bar{y}_i \geqslant 0$, так что ограничения неотрицательности также удовлетворены. ■

Итак, мы показали, что если данная задача линейного программирования разрешима, процедура INITIALIZE_SIMPLEX возвращает допустимое решение и процедура SIMPLEX завершается, не выдав сообщение “неограниченна”, то возвращенное решение является оптимальным. Мы также показали, как строится оптимальное решение двойственной задачи линейного программирования.

Упражнения

29.4-1. Сформулируйте двойственную задачу для задачи, приведенной в упражнении 29.3-4.

29.4-2. Предположим, у нас есть задача линейного программирования, не приведенная к стандартной форме. Можно получать двойственную задачу в два этапа: сначала привести исходную задачу к стандартной форме, а затем формулировать двойственную. Однако было бы удобно иметь возможность сразу формулировать двойственную задачу. Объясните, каким образом, имея произвольную задачу линейного программирования, можно получить двойственную задачу непосредственно.

29.4-3. Запишите двойственную задачу для задачи максимального потока, заданной уравнениями (29.47)–(29.50). Объясните, как эта формулировка интерпретируется в качестве задачи минимального разреза.

29.4-4. Запишите двойственную задачу для задачи потока с минимальными затратами, заданной уравнениями (29.51)–(29.55).Объясните, как интерпретировать полученную задачу с помощью графов и потоков.

29.4-5. Покажите, что задача, двойственная к двойственной задаче линейного программирования, является прямой задачей.

29.4-6. Какой результат из главы 26 можно интерпретировать как слабую двойственность для задачи максимального потока?

29.5 Начальное базисное допустимое решение

В данном разделе мы сначала покажем, как проверить, является ли задача линейного программирования разрешимой, и как в случае положительного ответа получить каноническую форму, базисное решение которой является допустимым. В заключение мы докажем основную теорему линейного программирования, в которой утверждается, что процедура SIMPLEX всегда дает правильный результат.

Поиск начального решения

В разделе 29.3 предполагалось, что у нас есть некая процедура INITIALIZE_SIMPLEX, которая определяет, имеет ли задача линейного программирования допустимые решения, и в случае положительного ответа возвращает каноническую форму, имеющую допустимое базисное решение. Опишем данную процедуру.

Даже если задача линейного программирования является разрешимой, начальное базисное решение может оказаться недопустимым. Рассмотрим, например,

следующую задачу:

$$\text{Максимизировать } 2x_1 - x_2 \tag{29.105}$$
$$\text{при условиях}$$
$$2x_1 - x_2 \leqslant 2 \tag{29.106}$$
$$x_1 - 5x_2 \leqslant -4 \tag{29.107}$$
$$x_1, x_2 \geqslant 0 \tag{29.108}$$

Если преобразовать эту задачу в каноническую форму, базисным решением будет $x_1 = 0$, $x_2 = 0$. Это решение нарушает ограничение (29.107) и, следовательно, не является допустимым. Таким образом, процедура INITIALIZE_SIMPLEX не может сразу возвратить эту очевидную каноническую форму. На первый взгляд, вообще неясно, имеет ли данная задача допустимые решения. Чтобы определить, существуют ли они, можно сформулировать ***вспомогательную задачу линейного программирования*** (auxiliary linear program). Для этой вспомогательной задачи мы сможем (причем достаточно легко) найти каноническую форму, базисное решение которой является допустимым. Более того, решение этой вспомогательной задачи линейного программирования позволит определить, является ли исходная задача разрешимой, и если да, то обеспечит допустимое решение, которым можно инициализировать процедуру SIMPLEX.

Лемма 29.11. Пусть L — задача линейного программирования в стандартной форме, заданная уравнениями (29.16)-(29.18), а L_{aux} — следующая задача линейного программирования с $n + 1$ переменными:

$$\text{Максимизировать } -x_0 \tag{29.109}$$
$$\text{при условиях}$$
$$\sum_{j=1}^{n} a_{ij}x_j - x_0 \leqslant b_i \qquad \text{для } i = 1, 2, \ldots, m, \tag{29.110}$$
$$x_j \geqslant 0 \qquad \text{для } j = 0, 1, \ldots, n. \tag{29.111}$$

Задача L разрешима тогда и только тогда, когда оптимальное целевое значение задачи L_{aux} равно 0.

Доказательство. Предположим, что L имеет допустимое решение $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$. Тогда решение $\bar{x}_0 = 0$ в комбинации с $\bar{x}$ является допустимым решением задачи L_{aux} с целевым значением, равным 0. Поскольку в L_{aux} присутствует ограничение $x_0 \geqslant 0$, а цель состоит в максимизации $-x_0$, это решение должно быть оптимальным решением L_{aux}.

И обратно, предположим, что оптимальное целевое значение задачи L_{aux} равно 0. Тогда $\bar{x}_0 = 0$ и значения остальных переменных $\bar{x}$ удовлетворяют ограничениям L. ■

Теперь опишем стратегию поиска начального базисного допустимого решения задачи линейного программирования L, заданной в стандартной форме:

INITIALIZE_SIMPLEX(A, b, c)
1 Пусть k — индекс минимального коэффициента b_i
2 **if** $b_k \geqslant 0$ ▷ Допустимо ли начальное базисное решение?
3 **then return** $(\{1, 2, \ldots, n\}, \{n+1, n+2, \ldots, n+m\}, A, b, c, 0)$
4 Образуем L_{aux} путем добавления $-x_0$ к левой части каждого уравнения и задаем целевую функцию равной $-x_0$
5 Пусть (N, B, A, b, c, v) — результирующая каноническая форма задачи L_{aux}
$l \leftarrow n + k$
6 ▷ L_{aux} содержит $n + 1$ небазисную переменную
▷ и m базисных переменных
7 $(N, B, A, b, c, v) \leftarrow$ PIVOT$(N, B, A, b, c, v, l, 0)$
8 ▷ Данное базисное решение является допустимым для L_{aux}
9 Проводим итерации цикла **while**, описанного в строках 2–11 процедуры SIMPLEX, пока не будет найдено оптимальное решение L_{aux}
10 **if** в базисном решении $\bar{x}_0 = 0$
11 **then return** конечную каноническую форму без переменной x_0 и с исходной целевой функцией
12 **else return** "задача неразрешима"

Процедура INITIALIZE_SIMPLEX работает следующим образом. В строках 1–3 неявно проверяется базисное решение исходной канонической формы задачи L, в которой $N = \{1, 2, \ldots, n\}$,$B = \{n+1, n+2, \ldots, n+m\}$, $\bar{x}_i = b_i$ для всех $i \in B$ и $x_j = 0$ для всех $j \in N$. (Создание данной канонической формы не требует дополнительных усилий, поскольку значения A, b и c в стандартной и канонической формах одинаковы.) Если это базисное решение допустимо, т.е. $\bar{x}_i \geqslant 0$ для всех $i \in N \cup B$, то возвращается данная каноническая форма. В противном случае в строке 4 формируется вспомогательная задача линейного программирования L_{aux} так, как это описано в лемме 29.11. В строке 5 мы также присваиваем l индекс наибольшего по модулю отрицательного значения b_i, но переиндексация выполняется после преобразования к каноническому виду. Поскольку начальное базисное решение задачи L недопустимо, начальное базисное решение канонической формы L_{aux} также не будет допустимым. Поэтому в строке 7 выполняется вызов процедуры PIVOT, в котором вводимой переменной является x_0, а выводимой x_l. Немного позже мы покажем, что базисное решение, полученное в результате этого вызова процедуры PIVOT, будет допустимым. Имея каноническую форму, базисное решение которой является допустимым, мы можем (строка 9) повторно вызывать процедуру PIVOT до тех пор, пока не будет найдено окон-

чательное решение вспомогательной задачи линейного программирования. Если проверка в строке 10 показывает, что найдено оптимальное решение задачи L_{aux} с равным 0 целевым значением, тогда в строке 11 для задачи L создается каноническая форма, базисное решение которой является допустимым. Для этого из ограничений удаляются все члены с x_0 и восстанавливается исходная целевая функция задачи L. Исходная целевая функция может содержать как базисные, так и небазисные переменные. Поэтому все вхождения базисных переменных в целевой функции нужно заменить правыми частями соответствующих ограничений. Если же в строке 10 обнаруживается, что исходная задача линейного программирования неразрешима, то сообщение об этом возвращается в строке 12.

Теперь покажем, как работает процедура INITIALIZE_SIMPLEX на примере задачи линейного программирования (29.105)–(29.108). Эта задача будет разрешимой, если нам удастся найти неотрицательные значения x_1 и x_2, удовлетворяющие неравенствам (29.106) и (29.107). Используя лемму 29.11, сформулируем вспомогательную задачу:

$$\begin{aligned}
&\text{Максимизировать} && -x_0 && && (29.112)\\
&\text{при условиях} && && && \\
& && 2x_1 - x_2 - x_0 \leqslant && 2 && (29.113)\\
& && x_1 - 5x_2 - x_0 \leqslant && -4 && (29.114)\\
& && x_1, x_2, x_0 \geqslant && 0 &&
\end{aligned}$$

Согласно лемме 29.11, если оптимальное целевое значение этой вспомогательной задачи равно 0, то исходная задача имеет допустимое решение. Если же оптимальное целевое значение вспомогательной задачи отрицательно, то исходная задача не имеет допустимых решений.

Запишем вспомогательную задачу в канонической форме:

$$\begin{aligned}
z &= -x_0\\
x_3 &= 2 - 2x_1 + x_2 + x_0\\
x_4 &= -4 - x_1 + 5x_2 + x_0
\end{aligned}$$

Пока что базисное решение, в котором $x_4 = -4$, не является допустимым решением данной вспомогательной задачи. Однако с помощью единственного вызова процедуры PIVOT мы можем превратить эту каноническую форму в такую, базисное решение которой будет допустимым. Согласно строке 7, мы выбираем в качестве вводимой переменной x_0. Согласно строке 1, в качестве выводимой переменной выбирается x_4, базисная переменная, которая в базисном решении имеет наибольшее по абсолютной величине отрицательное значение. После заме-

щения получаем следующую каноническую форму:

$$\begin{aligned} z &= -4 - x_1 + 5x_2 - x_4 \\ x_0 &= \quad 4 + x_1 - 5x_2 + x_4 \\ x_3 &= \quad 6 - x_1 - 4x_2 + x_4 \end{aligned}$$

Связанное с ней базисное решение $(x_0, x_1, x_2, x_3, x_4) = (4, 0, 0, 6, 0)$ является допустимым. Теперь повторно вызываем процедуру PIVOT до тех пор, пока не получим оптимальное решение задачи L_{aux}. В данном случае после одного вызова PIVOT с вводимой переменной x_2 и выводимой переменной x_0 получаем:

$$\begin{aligned} z &= -x_0 \\ x_2 &= \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \end{aligned}$$

Эта каноническая форма является окончательным решением вспомогательной задачи. Поскольку в данном решении $x_0 = 0$, можно сделать вывод, что исходная задача разрешима. Более того, поскольку $x_0 = 0$, ее можно просто удалить из множества ограничений. Затем можно использовать исходную целевую функцию, в которой сделаны соответствующие подстановки, чтобы она содержала только небазисные переменные. В нашем примере целевая функция имеет вид

$$2x_1 - x_2 = 2x_1 - \left(\frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5}\right).$$

После подстановки $x_0 = 0$ и соответствующих упрощений получаем целевую функцию, записанную как

$$-\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5},$$

и каноническую форму:

$$\begin{aligned} z &= -\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \\ x_2 &= \quad \frac{4}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \quad \frac{14}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \end{aligned}$$

Эта каноническая форма имеет допустимое базисное решение, и ее можно возвратить процедуре SIMPLEX.

Докажем формально корректность процедуры INITIALIZE_SIMPLEX.

Лемма 29.12. Если задача линейного программирования L не имеет допустимых решений, то процедура INITIALIZE_SIMPLEX возвращает сообщение "неразрешима". В противном случае процедура возвращает корректную каноническую форму, базисное решение которой является допустимым.

Доказательство. Сначала предположим, что задача линейного программирования L не имеет допустимых решений. Тогда, согласно лемме 29.11, оптимальное целевое значение задачи L_{aux}, заданной формулами (29.109)–(29.111), является ненулевым, а поскольку x_0 подчиняется ограничению неотрицательности, оптимальное решение должно иметь отрицательное целевое значение. Более того, это целевое значение должно быть конечным, так как решение, в котором $x_i = 0$ для $i = 1, 2, \ldots, n$ и $x_0 = |\min_{i=1}^{m} \{b_i\}|$, является допустимым и имеет целевое значение $-|\min_{i=1}^{m} \{b_i\}|$. Следовательно, в строке 9 процедуры INITIALIZE_SIMPLEX будет найдено решение с отрицательным целевым значением. Пусть $\bar{x}$ — базисное решение, связанное с окончательной канонической формой. Мы не можем получить $\bar{x}_0 = 0$, поскольку тогда задача L_{aux} имела бы целевое значение 0, а это противоречит факту, что целевое значение отрицательно. Таким образом, проверка в строке 10 приведет к тому, что в строке 12 будет возвращено сообщение "неразрешима".

Теперь предположим, что задача линейного программирования L имеет допустимое решение. Из упражнения 29.3-3 мы знаем, что если $b_i \geqslant 0$ для $i = = 1, 2, \ldots, m$, то базисное решение, связанное с начальной канонической формой, является допустимым. В этом случае в строках 2–3 будет возвращена каноническая форма, связанная с исходной задачей. (Для преобразования стандартной формы в каноническую не требуется много усилий, поскольку A, b и c одинаковы и в той, и в другой форме.)

Для завершения доказательства рассмотрим случай, когда задача разрешима, но в строке 3 каноническая форма не возвращается. Докажем, что в этом случае в строках 4–9 выполняется поиск допустимого решения задачи L_{aux} с целевым значением, равным 0. Согласно строкам 1–2, в данном случае

$$b_k < 0,$$

и

$$b_k \leqslant b_i \text{ для всех } i \in B. \tag{29.115}$$

В строке 7 выполняется одна операция замещения, в процессе которой из базиса выводится переменная x_l (вспомним, что $l = n + k$), стоящая в левой части уравнения с минимальным b_i, и вводится дополнительная переменная x_0. Покажем, что после такого замещения все элементы b являются неотрицательными, и, следовательно, данное базисное решение задачи L_{aux} является допустимым.

Обозначим через $\bar{x}$ базисное решение после вызова процедуры PIVOT, и пусть $\widehat{b}$ и $\widehat{B}$ — значения, возвращенные процедурой PIVOT. Из леммы 29.1 следует, что

$$\bar{x}_i = \begin{cases} b_i - a_{ie}\widehat{b}_e & \text{если } i \in \widehat{B} - \{e\}, \\ b_l / a_{le} & \text{если } i = e. \end{cases} \tag{29.116}$$

При вызове процедуры PIVOT в строке 7 $e = 0$ и, в соответствии с (29.110),

$$a_{i0} = a_{ie} = -1 \text{ для всех } i \in B. \tag{29.117}$$

(Заметим, что a_{i0} — это коэффициент при x_0 в выражении (29.110), а не противоположное ему значение, поскольку задача L_{aux} находится в стандартной, а не канонической форме.) Поскольку $l \in B$, то $a_{le} = -1$. Таким образом, $b_l / a_{le} > 0$ и, следовательно, $\bar{x}_e > 0$. Для остальных базисных переменных получаем:

$$\begin{aligned} \bar{x}_i &= b_i - a_{ie}\widehat{b}_e = && \text{(согласно (29.116))} \\ &= b_i - a_{ie}\,(b_l / a_{le}) = && \text{(в соответствии со строкой 2 процедуры PIVOT)} \\ &= b_i - b_l \geqslant && \text{(из (29.117) и } a_{le} = -1) \\ &\geqslant 0 && \text{(из неравенства (29.115))} \,. \end{aligned}$$

Это означает, что теперь все базисные переменные неотрицательны. Следовательно, базисное решение, полученное в результате вызова процедуры PIVOT в строке 7, является допустимым. Следующей выполняется строка 9 и решается задача L_{aux}. Поскольку мы предположили, что задача L имеет допустимое решение, согласно лемме 29.11, задача L_{aux} имеет оптимальное решение с равным 0 целевым значением. Так как все канонические формы эквивалентны, в конечном базисном решении задачи L_{aux} $\bar{x}_0 = 0$, и после удаления из данной задачи переменной x_0 мы получим каноническую форму, базисное решение которой допустимо в задаче L. Эта каноническая форма возвращается в строке 10. ■

Основная теорема линейного программирования

Мы завершим данную главу демонстрацией корректности работы процедуры SIMPLEX. Произвольная задача линейного программирования или неразрешима, или неограниченна, или имеет оптимальное решение с конечным целевым значением, и в каждом случае процедура SIMPLEX будет работать правильно.

Теорема 29.13 (Основная теорема линейного программирования). Для любой задачи линейного программирования L, заданной в стандартной форме, возможен только один из следующих вариантов:

1. задача имеет оптимальное решение с конечным целевым значением;
2. задача является неразрешимой;
3. задача является неограниченной.

Если задача L является неразрешимой, процедура SIMPLEX возвращает сообщение "неразрешима". Если задача L является неограниченной, процедура SIMPLEX возвращает сообщение "неограниченна". В противном случае процедура SIMPLEX возвращает оптимальное решение с конечным целевым значением.

Доказательство. Согласно лемме 29.12, если задача линейного программирования L неразрешима, процедура SIMPLEX возвращает сообщение "неразрешима". Теперь предположим, что задача L разрешима. В соответствии с леммой 29.12, процедура INITIALIZE_SIMPLEX возвращает каноническую форму, базисное решение которой допустимо. Тогда, согласно лемме 29.7, процедура SIMPLEX или возвращает сообщение "неограниченна", или завершается возвращением допустимого решения. Если она завершается и возвращает конечное решение, то это решение является оптимальным согласно теореме 29.10. Если же процедура SIMPLEX возвращает сообщение "неограниченна", то задача L является неограниченной согласно лемме 29.2. Поскольку процедура SIMPLEX всегда завершается одним из перечисленных способов, доказательство завершено. ■

Упражнения

29.5-1. Напишите подробный псевдокод программы для реализации строк 5 и 11 процедуры INITIALIZE_SIMPLEX.

29.5-2. Покажите, что при выполнении процедурой INITIALIZE_SIMPLEX основного цикла процедуры SIMPLEX никогда не возвращается сообщение "неограниченна".

29.5-3. Предположим, что нам дана задача линейного программирования в стандартной форме L, и для обеих задач, прямой и двойственной, базисные решения, связанные с начальными каноническими формами, являются допустимыми. Покажите, что оптимальное целевое значение задачи L равно 0.

29.5-4. Предположим, что в задачу линейного программирования разрешено включать строгие неравенства. Покажите, что в этом случае основная теорема линейного программирования не выполняется.

29.5-5. Решите следующую задачу линейного программирования с помощью процедуры SIMPLEX:

$$\begin{aligned} &\text{Максимизировать} && x_1 + 3x_2 \\ &\text{при условиях} \\ & && -x_1 + x_2 \leqslant -1 \\ & && -2x_1 - 2x_2 \leqslant -6 \\ & && -x_1 + 4x_2 \leqslant 2 \\ & && x_1, x_2 \geqslant 0 \end{aligned}$$

29.5-6. Решите задачу линейного программирования (29.6)–(29.10).

29.5-7. Рассмотрим задачу линейного программирования с одной переменной, которую назовем P:

$$\begin{aligned} &\text{Максимизировать} && tx \\ &\text{при условиях} \\ & && rx \leqslant s \\ & && x \geqslant 0 \end{aligned}$$

где r, s, t — произвольные действительные числа. Пусть D — задача, двойственная P. При каких значениях r, s, и t можно утверждать, что:

а) обе задачи P и D имеют оптимальные решения с конечными целевыми значениями;

б) является разрешимой, а D — неразрешимой;

в) является разрешимой, а P — неразрешимой;

г) ни одна из задач P и D не является разрешимой.

Задачи

29-1. **Разрешимость линейных неравенств**

Пусть задано множество m линейных неравенств с n переменными $x_1, x_2, \ldots, x_n$. В ***задаче о разрешимости линейных неравенств*** требуется ответить, существует ли набор значений переменных, удовлетворяющий одновременно всем неравенствам.

а) Покажите, что для решения задачи о разрешимости системы линейных неравенств можно использовать алгоритм решения задачи линейного программирования. Число переменных и ограничений, используемых в задаче линейного программирования, должно полиномиально зависеть от n и m.

б) Покажите, что алгоритм решения задачи о разрешимости системы линейных неравенств можно использовать для решения задачи линейного программирования. Число переменных и линейных неравенств, используемых в задаче о разрешимости системы линейных неравенств, должно полиномиально зависеть от числа переменных n и числа ограничений m задачи линейного программирования.

29-2. **Дополняющая нежесткость (complementary slackness)**

Дополняющая нежесткость характеризует связь между значениями переменных прямой задачи и ограничениями двойственной задачи, а также между значениями переменных двойственной задачи и ограничениями прямой задачи. Пусть $\bar{x}$ — допустимое решение прямой задачи линейного программирования (29.16)–(29.18), а $\bar{y}$ — допустимое решение двойственной задачи (29.86)–(29.88). Принцип дополняющей нежесткости гласит, что следующие условия являются необходимыми и достаточными условиями оптимальности $\bar{x}$ и $\bar{y}$:

$$\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j \text{ или } \bar{x}_j = 0 \quad \text{для } j = 1, 2, \ldots, n$$

и

$$\sum_{j=1}^{n} a_{ij}\bar{x}_j = b_i \text{ или } \bar{y}_i = 0 \quad \text{для } i = 1, 2, \ldots, m.$$

а) Убедитесь, что принцип дополняющей нежесткости справедлив для задачи линейного программирования (29.56)–(29.60).

б) Докажите, что принцип дополняющей нежесткости справедлив для любой прямой и соответствующей ей двойственной задачи.

в) Докажите, что допустимое решение $\bar{x}$ прямой задачи линейного программирования (29.16)–(29.18) является оптимальным тогда и только тогда, когда существуют значения $\bar{y} = (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$, такие что:

1) $\bar{y}$ является допустимым решением двойственной задачи линейного программирования (29.86)–(29.88),
2) $\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j$, когда $\bar{x}_j > 0$, и
3) $\bar{y}_i = 0$, когда $\sum_{j=1}^{n} a_{ij}\bar{x}_j < b_i$.

29-3. **Целочисленное линейное программирование**

Задача целочисленного линейного программирования — это задача линейного программирования с дополнительным ограничением, состоящим в том, что переменные x должны принимать целые значения. В упражнении 34.5-3 показывается, что даже определение того, имеет ли задача

целочисленного линейного программирования допустимые решения, является NP-полной задачей; это значит, что вряд ли существует полиномиальный по времени алгоритм решения данной задачи.

а) Покажите, что для задач целочисленного линейного программирования характерна слабая двойственность (лемма 29.8).

б) Покажите, что задачи целочисленного линейного программирования не всегда характеризуются двойственностью (описанной в теореме 29.10).

в) Пусть задана прямая задача линейного программирования в стандартной форме, и пусть P — оптимальное целевое значение прямой задачи, D — оптимальное целевое значение ее двойственной задачи, IP —оптимальное целевое значение целочисленной версии прямой задачи (т.е. прямой задачи с тем дополнительным ограничением, что переменные принимают целые значения), а ID — оптимальное целевое значение целочисленной версии двойственной задачи. Исходя из предположения, что и прямая, и двойственная целочисленные задачи разрешимы и ограниченны, покажите, что $IP \leqslant P = D \leqslant ID$.

29-4. **Лемма Фаркаша**

Пусть A — матрица размером $m \times n$, а c — n-мерный вектор. Лемма Фаркаша (Farkas) гласит, что разрешима только одна из систем

$$\begin{aligned} Ax &\leqslant 0, \\ c^{\mathrm{T}}x &> 0 \end{aligned}$$

и

$$\begin{aligned} A^{\mathrm{T}}y &= c, \\ y &\geqslant 0, \end{aligned}$$

где x — n-мерный вектор, а y — m-мерный вектор. Докажите эту лемму.

Заключительные замечания

В данной главе мы только приступили к изучению обширной области линейного программирования. Множество книг посвящено исключительно линейному программированию. Среди них Чватал (Chvatal) [62], Гасс (Gass) [111], Карлов (Karloff) [171], Шрайвер (Schrijver) [266] и Вандербей (Vanderbei) [304]. Во многих других книгах подробно линейное программирование рассматривается наряду с другими вопросами (см., например, Пападимитриу (Papadimitriou), Штейглиц (Steiglitz) [237] и Ахуя (Ahuja), Магнанти (Magnanti), Орлин (Orlin) [7]). Изложение в данной главе построено на подходе, предложенном Чваталом (Chvatal).

Симплекс-алгоритм для задач линейного программирования был открыт Данцигом (G. Dantzig) в 1947 году. Немного позже было обнаружено, что множество задач из различных областей деятельности можно сформулировать в виде задач линейного программирования и решить с помощью симплекс-алгоритма. Осознание этого факта привело к стремительному росту использования линейного программирования и его алгоритмов. Различные варианты симплекс-алгоритма остаются наиболее популярными методами решения задач линейного программирования. Эта история описана во многих работах, например, в примечаниях к [62] и [171].

Эллипсоидный алгоритм, предложенный Л. Г. Хачияном в 1979 году, явился первым алгоритмом решения задач линейного программирования с полиномиальным временем выполнения. Он был основан на более ранних работах Н.З. Шора, Д.Б. Юдина и А.С. Немировского. Использование эллипсоидного алгоритма для решения разнообразных задач комбинаторной оптимизации описано в работе Грётшеля (Grötschel), Ловаса (Lovasz) и Шрайвера [134]. Однако на сегодняшний день эллипсоидный алгоритм в практическом применении не может конкурировать с симплекс-алгоритмом.

В работе Кармаркара (Karmarkar) [172] содержится описание предложенного им алгоритма внутренней точки. Многие его последователи разработали свои алгоритмы внутренней точки. Хороший обзор этих алгоритмов можно найти в статье Гольдфарба (Goldfarb) и Тодда (Todd) [122] и в книге Е (Ye) [319].

Анализ симплекс-алгоритма относится к сфере активных исследований. Кли (Klee) и Минти (Minty) построили пример, в котором симплекс-алгоритм выполняет $2^n - 1$ итераций. На практике симплекс-алгоритм работает очень эффективно, и многие исследователи пытались найти теоретическое обоснование этому эмпирическому наблюдению. Исследования, начатые Боргвардтом (Borgwardt) и продолженные многими другими, показывают, что при определенных вероятностных предположениях об исходных данных симплекс-алгоритм сходится за ожидаемое полиномиальное время. Последних результатов в этой области добились Спилмен (Spielman) и Тенг (Teng) [284], которые ввели понятие “сглаженного анализа алгоритмов” и применили его к симплекс-алгоритму.

Известно, что симплекс-алгоритм работает более эффективно в определенных специальных случаях. В частности, заслуживает внимания сетевой симплекс-алгоритм — разновидность симплекс-алгоритма, приспособленная для решения задач сетевых потоков. Для некоторых сетевых задач, включая задачи поиска кратчайшего пути, максимального потока и потока с минимальными затратами, варианты сетевого симплекс-алгоритма достигают результата за полиномиальное время. Обратитесь, например, к статье Орлина [234] и ее ссылкам.

ГЛАВА 30

Полиномы и быстрое преобразование Фурье

Для выполнения непосредственного сложения двух полиномов степени n требуется время $\Theta(n)$, однако для их непосредственного умножения требуется время $\Theta(n^2)$. В данной главе мы покажем, как с помощью быстрого преобразования Фурье (БПФ) (Fast Fourier Transform, FFT) можно сократить время умножения полиномов до $\Theta(n \lg n)$.

Наиболее часто преобразования Фурье (а следовательно, и БПФ) используются в обработке сигналов. Сигнал задается во ***временной области*** (time domain) как функция, отображающая время в амплитуду. Анализ Фурье позволяет выразить сигнал как взвешенную сумму сдвинутых по фазе синусоид различных частот. Веса и фазы связаны с частотными характеристиками сигнала в ***частотной области*** (frequency domain). Обработка сигналов — обширная область исследований, которой посвящено несколько отличных книг; в примечаниях к данной главе приводятся ссылки на некоторые из них.

Полиномы

Полиномом (polynomial) относительно переменной x над алгебраическим полем F называется представление функции $A(x)$ в виде суммы

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \ .$$

Значения $a_1, a_2, \ldots, a_{n-1}$ называются ***коэффициентами*** данного полинома. Коэффициенты принадлежат некоторому полю F, как правило, это множество комплексных чисел **C**. Говорят, что полином $A(x)$ имеет ***степень*** k, если его старшим ненулевым коэффициентом является a_k. Любое целое число, строго большее степени полинома, называется ***границей степени*** (degree-bound) данного полинома. Следовательно, степенью полинома с границей степени n может быть любое целое число от 0 до $n-1$ включительно.

Для полиномов можно определить множество разнообразных операций. Например, ***сложение полиномов*** (polynomial addition): если $A(x)$ и $B(x)$ — полиномы степени не выше n, то их ***суммой*** (sum) является полином $C(x)$ степени не выше n, такой что $C(x) = A(x) + B(x)$ для всех x из соответствующего поля. Таким образом, если

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

и

$$B(x) = \sum_{j=0}^{n-1} b_j x^j \ ,$$

то

$$C(x) = \sum_{j=0}^{n-1} c_j x^j$$

где $c_j = a_j + b_j$ для всех $j = 0, 1, \ldots, n-1$. Например, если $A(x) = 6x^3 + 7x^2 - 10x + 9$ и $B(x) = -2x^3 + 4x - 5$, то $C(x) = 4x^3 + 7x^2 - 6x + 4$.

Умножение полиномов (polynomial multiplication): если $A(x)$ и $B(x)$ — полиномы степени не выше n, их ***произведением*** (product) $C(x)$ является полином степени не выше $2n-1$, такой что $C(x) = A(x)B(x)$ для всех x из соответствующего поля. Вероятно, вам уже доводилось перемножать полиномы, умножая каждый член полинома $A(x)$ на каждый член полинома $B(x)$ и выполняя приведение членов с одинаковыми степенями. Например, умножение полиномов $A(x) = 6x^3 + 7x^2 - 10x + 9$ и $B(x) = -2x^3 + 4x - 5$ можно выполнить следующим образом:

$$\begin{array}{rrrrrrr}
 & & & 6x^3 & + 7x^2 & - 10x & + 9 \\
 & & & - 2x^3 & & + 4x & - 5 \\
\hline
 & & & - 30x^3 & - 35x^2 & + 50x & - 45 \\
 & & 24x^4 & + 28x^3 & - 40x^2 & + 36x & \\
 & - 12x^6 & - 14x^5 & + 20x^4 & - 18x^3 & & \\
\hline
 & - 12x^6 & - 14x^5 & + 44x^4 & - 20x^3 & - 75x^2 & + 86x - 45
\end{array}$$

По-другому произведение $C(x)$ можно записать как

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j \ , \tag{30.1}$$

где

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} \ . \tag{30.2}$$

Заметим, что $\text{degree}(C) = \text{degree}(A) + \text{degree}(B)$, откуда вытекает, что если A — полином степени не выше n_A, а B — полином степени не выше n_B, то C — полином степени не выше $n_A + n_B - 1$. Тем не менее, поскольку полином степени не выше k является также полиномом степени не выше $k+1$, мы будем говорить, что произведение C является полиномом степени не выше $n_A + n_B$.

Краткое содержание главы

В разделе 30.1 описаны два способа представления полиномов: представление, основанное на коэффициентах, и представление, основанное на значениях в точках. Для непосредственного умножения полиномов — уравнения (30.1) и (30.2) — требуется время $\Theta\left(n^2\right)$, если полиномы представлены при помощи коэффициентов, и только $\Theta(n)$, когда они представлены в форме, основанной на значениях в точках. Однако с помощью преобразования представлений можно умножить заданные с помощью коэффициентов полиномы за время $\Theta(n \lg n)$. Чтобы понять, как это происходит, необходимо сначала изучить свойства комплексных корней из единицы, что и предлагается сделать в разделе 30.2. Затем также описанные в разделе 30.2 прямое и обратное быстрое преобразование Фурье используются для выполнения указанных преобразований представлений. В разделе 30.3 показано, как быстро реализовать БПФ в последовательных и параллельных моделях.

В данной главе широко используются комплексные числа, поэтому символ i будет использоваться исключительно для обозначения $\sqrt{-1}$.

30.1 Представление полиномов

Представления полиномов в форме коэффициентов и в форме значений в точках в определенном смысле эквивалентны: полиному, заданному в форме точек-значений, соответствует единственный полином в коэффициентной форме. В данном разделе мы познакомимся с обоими представлениями и покажем, как их можно скомбинировать, чтобы выполнить умножение двух полиномов степени не выше n за время $\Theta(n \lg n)$.

Представление, основанное на коэффициентах

Основанным на коэффициентах представлением (coefficient representation) полинома $A(x) = \sum_{j=0}^{n-1} a_j x^j$ степени не выше n является вектор коэффициентов $a = (a_0, a_1, \ldots, a_{n-1})$. В матричных уравнениях данной главы мы будем считать векторы векторами-столбцами.

Основанное на коэффициентах представление удобно при выполнении определенных операций над полиномами. Например, операция ***вычисления*** (evaluating) полинома $A(x)$ в некой заданной точке x_0 заключается в вычислении значения $A(x_0)$. Если использовать ***схему Горнера*** (Horner's rule), вычисление требует $\Theta(n)$ времени:

$$A(x_0) = ((\cdots((a_{n-1})x_0 + a_{n-2})x_0 + \cdots + a_2)x_0 + a_1)x_0 + a_0 \ .$$

Аналогично, сложение двух полиномов, представленных векторами коэффициентов $a = (a_0, a_1, \ldots, a_{n-1})$ и $b = (b_0, b_1, \ldots, b_{n-1})$, занимает время $\Theta(n)$: мы просто создаем вектор коэффициентов $c = (c_0, c_1, \ldots, c_{n-1})$, где $c_j = a_j + b_j$, $j = 0, 1, \ldots, n-1$.

Рассмотрим теперь умножение двух полиномов степени не выше n, $A(x)$ и $B(x)$, представленных в коэффициентной форме. Если использовать метод, описанный уравнениями (30.1) и (30.2), умножение данных полиномов займет время $\Theta(n^2)$, поскольку каждый коэффициент из вектора a необходимо умножить на каждый коэффициент из вектора b. Операция умножения полиномов в коэффициентной форме оказывается гораздо более сложной, чем операции вычисления полинома или сложения двух полиномов. Результирующий вектор коэффициентов c, заданный формулой (30.2), также называется ***сверткой*** (convolution) исходных векторов a и b, и обозначается он как $c = a \otimes b$. Поскольку умножение полиномов и вычисление сверток являются фундаментальными вычислительными задачами, имеющими важное практическое значение, данная глава посвящена эффективным алгоритмам их решения.

Представление, основанное на значениях в точках

Основанным на значениях в точках представлением (point-value representation) полинома $A(x)$ степени не выше n является множество, состоящее из n ***пар точка–значение*** (point-value pairs):

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\} \ ,$$

таких что все x_k различны и

$$y_k = A(x_k) \tag{30.3}$$

для $k = 0, 1, \ldots, n-1$. Каждый полином имеет множество различных представлений, основанных на значениях в точках, поскольку в качестве базиса такого представления можно использовать любое множество n различных точек $x_0, x_1, \ldots, x_{n-1}$.

Получить основанное на значениях в точках представление полинома, заданного с помощью коэффициентов, достаточно просто: для этого достаточно выбрать n различных точек $x_0, x_1, \ldots, x_{n-1}$ и вычислить $A(x_k)$ при $k = 0, 1, \ldots, n-1$. С помощью схемы Горнера такое вычисление можно выполнить за время $\Theta(n^2)$. Далее мы покажем, что при разумном выборе x_k данное вычисление можно ускорить, и оно будет выполняться за время $\Theta(n \lg n)$.

Обратная процедура — определение коэффициентов полинома, заданного в форме значений в точках, — называется ***интерполяцией*** (interpolation). В следующей теореме утверждается, что интерполяция является вполне определенной, когда граница степени искомого интерполяционного полинома равна числу заданных пар точка–значение.

Теорема 30.1 (Единственность интерполяционного полинома). Для любого множества, состоящего из n пар точка–значение $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$, таких что все значения x_k различны, существует единственный полином $A(x)$ с границей степени n, такой что $y_k = A(x_k)$ для $k = 0, 1, \ldots, n-1$.

Доказательство. Данное доказательство основано на существовании матрицы, обратной заданной. Уравнение (30.3) эквивалентно матричному уравнению

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} . \tag{30.4}$$

Матрица в левой части обозначается $V(x_0, x_1, \ldots, x_{n-1})$ и называется матрицей Вандермонда (Vandermonde). Согласно упражнению 28.1-11, определитель данной матрицы равен

$$\prod_{0 \leqslant j < k \leqslant n-1} (x_k - x_j) ,$$

следовательно, по теореме 28.5, она является обратимой (т.е. невырожденной), если все x_k различны. Таким образом, для заданного представления в виде значений в точках можно однозначно вычислить коэффициенты a_j:

$$a = V(x_0, x_1, \ldots, x_{n-1})^{-1} y .$$

■

В доказательстве теоремы 30.1 предлагается алгоритм интерполяции, основанный на решении системы линейных уравнений (30.4). Используя описанные в главе 28 алгоритмы LU-декомпозиции, эти уравнения можно решить за время $O\left(n^3\right)$.

Более быстрый алгоритм n-точечной интерполяции основан на ***формуле Лагранжа*** (Lagrange's formula):

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)} \ . \tag{30.5}$$

Можете самостоятельно убедиться, что правая часть уравнения (30.5) является полиномом степени не выше n, удовлетворяющим условию $y_k = A\left(x_k\right)$ для всех k. В упражнении 30.1-5 предлагается показать, как с помощью формулы Лагранжа вычислить коэффициенты полинома A за время $\Theta\left(n^2\right)$.

Таким образом, вычисление и интерполяция по n точкам являются вполне определенными обратимыми операциями, которые позволяют преобразовать коэффициентное представление полинома в представление в виде точек-значений и наоборот[1]. Решение этих задач с помощью описанных выше алгоритмов занимает время $\Theta\left(n^2\right)$.

Представление в виде значений в точках весьма удобно при выполнении многих операций над полиномами. При сложении, если $C\left(x\right) = A\left(x\right) + B\left(x\right)$, то $C\left(x_k\right) = A\left(x_k\right) + B\left(x_k\right)$ для любой точки x_k. Говоря более строго, если у нас есть основанное на значениях в точках представление полиномов A

$$\left\{\left(x_0, y_0\right), \left(x_1, y_1\right), \ldots, \left(x_{n-1}, y_{n-1}\right)\right\}$$

и B

$$\left\{\left(x_0, y'_0\right), \left(x_1, y'_1\right), \ldots, \left(x_{n-1}, y'_{n-1}\right)\right\}$$

(обратите внимание, что A и B вычисляются в *одних и тех же* n точках), то основанное на значениях в точках представление полинома C имеет вид

$$\left\{\left(x_0, y_0 + y'_0\right), \left(x_1, y_1 + y'_1\right), \ldots, \left(x_{n-1}, y_{n-1} + y'_{n-1}\right)\right\} \ .$$

Таким образом, время, необходимое для сложения двух полиномов степени не выше n, заданных в форме значений в точках, составляет $\Theta\left(n\right)$.

[1] Известно, что интерполяция является сложной задачей с точки зрения численной устойчивости. Хотя описанные здесь подходы математически корректны, небольшие различия во вводимых величинах или ошибки округления в ходе вычислений могут привести к значительным различиям в результатах.

Аналогично, представление в виде точек–значений удобно при выполнении умножения полиномов. Если $C(x) = A(x)B(x)$, то $C(x_k) = A(x_k)B(x_k)$ для любой точки x_k, и можно поточечно умножить представление A на соответствующее представление B и получить основанное на значениях в точках представление C. Однако возникает следующая проблема: $\text{degree}(C) = \text{degree}(A) + \text{degree}(B)$; если A и B имеют границу степени n, то граница степени результирующего полинома составляет $2n$. Стандартное представление каждого из полиномов A и B в виде значений в точках содержит n пар точка–значение. В результате их перемножения получится n пар точка–значение, однако для однозначной интерполяции полинома C с границей степени $2n$ требуется $2n$ пар (см. упражнение 30.1-4.) Следовательно, необходимо использовать "расширенные" представления полиномов A и B, которые содержат по $2n$ пар точка–значение каждое. Если задано расширенное представление точки–значения полинома A,

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\}$$

и соответствующее расширенное представление точки–значения полинома B

$$\{(x_0, y_0'), (x_1, y_1'), \ldots, (x_{2n-1}, y_{2n-1}')\} \ ,$$

то основанное на значениях в точках представление полинома C выглядит следующим образом:

$$\{(x_0, y_0 y_0'), (x_1, y_1 y_1'), \ldots, (x_{2n-1}, y_{2n-1} y_{2n-1}')\} \ .$$

Если два исходных полинома заданы в расширенной форме точки–значения, то время их умножения для получения результата в той же форме составляет $\Theta(n)$, что значительно меньше, чем время, необходимое для умножения полиномов, заданных в коэффициентной форме.

Наконец, рассмотрим, как вычислить значение полинома, заданного в виде значений в точках, в некоторой новой точке. По-видимому, для этой задачи не существует более простого подхода, чем преобразовать полином в коэффициентную форму, а затем вычислить его значение в новой точке.

Быстрое умножение полиномов, заданных в коэффициентной форме

Можно ли использовать метод умножения полиномов, заданных в виде значений в точках, время выполнения которого линейно зависит от n, для ускорения умножения полиномов, заданных в коэффициентной форме? Ответ зависит от способности быстро выполнять преобразование полинома из коэффициентной формы в форму точки–значения (вычисление) и обратно (интерполяция).

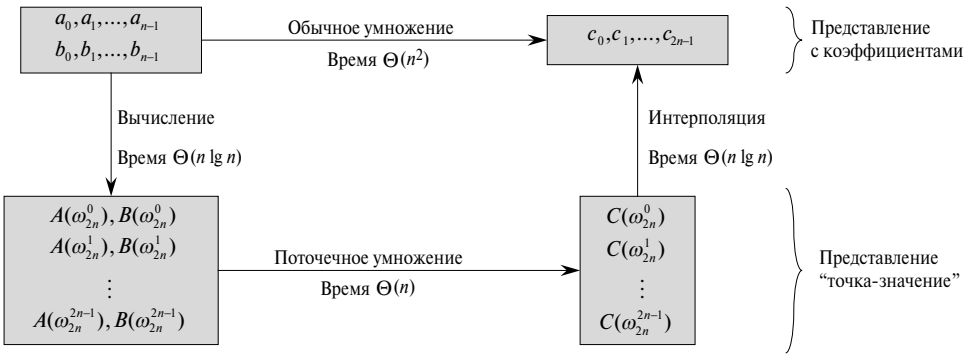


Рис. 30.1. Графическое представление эффективной процедуры умножения полиномов. Вверху приводится представление в коэффициентной форме, а внизу — в форме точек-значений. Идущие слева направо стрелки соответствуют операции умножения. Члены ω_{2n} являются комплексными корнями степени $2n$ из единицы

В качестве точек вычисления можно использовать любые точки, однако тщательный выбор точек дает возможность выполнять переход от одного представления к другому за время $\Theta(n \lg n)$. Как будет показано в разделе 30.2, если в качестве точек вычисления выбрать комплексные корни из единицы, то представление точки–значения можно получить с помощью дискретного преобразования Фурье (ДПФ) (Discrete Fourier Transform, DFT) вектора коэффициентов. Обратную операцию, интерполяцию, можно выполнить путем применения обратного дискретного преобразования Фурье к парам точки–значения, в результате чего получается вектор коэффициентов. В разделе 30.2 будет показано, что БПФ позволяет выполнять прямое и обратное ДПФ за время $\Theta(n \lg n)$.

На рис. 30.1 данная стратегия представлена графически. Небольшая сложность связана с границами степеней. Произведение двух полиномов степени не выше n является полиномом степени не выше $2n$. Поэтому прежде чем вычислять исходные полиномы A и B, их границы степени удваиваются и доводятся до $2n$ путем добавления n старших коэффициентов, равных 0. Поскольку теперь вектора коэффициентов содержат по $2n$ элементов, мы используем "комплексные корни $2n$-й степени из единицы", которые обозначены на рис. 30.1 как ω_{2n}.

Можно предложить следующую основанную на БПФ процедуру умножения двух полиномов $A(x)$ и $B(x)$ степени не выше n, в которой исходные полиномы и результат представлены в коэффициентной форме, а время выполнения составляет $\Theta(n \lg n)$. Предполагается, что n является степенью 2; это требование всегда можно удовлетворить, добавив равные нулю старшие коэффициенты.

1. *Удвоение границы степени*. Создаются коэффициентные представления полиномов $A(x)$ и $B(x)$ в виде полиномов с границей степени $2n$ путем добавления n нулевых старших коэффициентов.

2. *Вычисление*. Определяются представления полиномов $A(x)$ и $B(x)$ в форме точки–значения длины $2n$ путем двукратного применения БПФ порядка $2n$. Эти представления содержат значения двух заданных полиномов в точках, являющихся комплексными корнями степени $2n$ из единицы.
3. *Поточечное умножение*. Вычисляется представление точки–значения полинома $C(x) = A(x)B(x)$ путем поточечного умножения соответствующих значений. Это представление содержит значения полинома $C(x)$ в каждом корне степени $2n$ из единицы.
4. *Интерполяция*. Создается коэффициентное представление полинома $C(x)$ с помощью однократного применения БПФ к $2n$ парам точка–значение для вычисления обратного ДПФ.

Этапы (1) и (3) занимают время $\Theta(n)$, а этапы (2) и (4) занимают время $\Theta(n \lg n)$. Таким образом, показав, как использовать БПФ, мы докажем следующую теорему.

Теорема 30.2. Произведение двух полиномов степени не выше n в случае, когда исходные полиномы и результат находятся в коэффициентной форме, можно вычислить за время $\Theta(n \lg n)$. ■

Упражнения

30.1-1. Выполните умножение полиномов $A(x) = 7x^3 - x^2 + x - 10$ и $B(x) = 8x^3 - 6x + 3$ с помощью уравнений (30.1) и (30.2).

30.1-2. Вычисление полинома $A(x)$ степени не выше n в заданной точке x_0 можно также производить путем деления $A(x)$ на полином $(x - x_0)$, в результате чего получается полином-частное $q(x)$ степени не выше $n-1$ и остаток r, так что

$$A(x) = q(x)(x - x_0) + r\ .$$

Очевидно, что $A(x_0) = r$. Покажите, как вычислить остаток r и коэффициенты $q(x)$ за время $\Theta(n)$, если заданы точка x_0 и коэффициенты полинома A.

30.1-3. Найдите представление в виде точек-значений полинома $A^{\mathrm{rev}}(x) = \sum_{j=0}^{n-1} a_{n-1-j}x^j$, если известно представление в виде точек-значений полинома $A(x) = \sum_{j=0}^{n-1} a_j x^j$; предполагается, что все точки ненулевые.

30.1-4. Докажите, что для однозначного определения полинома степени не выше n необходимо задать n различных пар точка–значение, т.е. задание меньшего количества различных пар точка–значение не позволит определить единственный полином с границей степени n. (*Указание*: используя теорему 30.1, что можно сказать о множестве из $n-1$ пары точка–значение,

к которому добавляется еще одна произвольно выбранная пара точка–значение?)

30.1-5. Покажите, как с помощью уравнения (30.5) выполнить интерполяцию за время $\Theta\left(n^2\right)$. (*Указание*: сначала следует вычислить коэффициентное представление полинома $\prod_j (x - x_j)$, а затем разделить на $(x - x_k)$ для получения числителя каждого члена; см. упражнение 30.1-2. Каждый из n знаменателей можно вычислить за время $O(n)$.)

30.1-6. Объясните, что неправильно в "очевидном" подходе к делению представленных в виде значений в точках полиномов, когда значения y одного полинома делятся на соответствующие значения y второго полинома. Рассмотрите отдельно случаи, когда деление полиномов осуществляется без остатка и когда имеется остаток.

30.1-7. Рассмотрим два множества A и B, каждое из которых содержит n целых чисел, заключенных в пределах от 0 до $10n$. Мы хотим вычислить ***декартову сумму*** (Cartesian sum) A и B, определенную следующим образом:

$$C = \{x + y : x \in A \text{ и } y \in B\} \ .$$

Заметим, что целые числа из множества C заключены в пределах от 0 до $20n$. Требуется найти элементы C и указать, сколько раз каждый элемент C выступает в роли суммы элементов A и B. Покажите, что эту задачу можно решить за время $O(n \lg n)$. (*Указание*: представьте A и B в виде полиномов степени не выше $10n$.)

30.2 ДПФ и БПФ

В разделе 30.1 утверждалось, что используя в качестве точек комплексные корни из единицы, можно выполнять вычисление и интерполяцию полиномов за время $\Theta(n \lg n)$. В данном разделе мы дадим определение комплексных корней из единицы и изучим их свойства, определим дискретное преобразование Фурье (ДПФ), а затем покажем, как с помощью БПФ можно вычислять ДПФ и обратное ему преобразование за время $\Theta(n \lg n)$.

Комплексные корни из единицы

Комплексным корнем n-й степени из единицы (complex nth root of unity) является комплексное число ω, такое что $\omega^n = 1$.

Существует ровно n комплексных корней n-й степени из единицы: $e^{2\pi ik/n}$, где $k = 0, 1, \ldots, n-1$. Для интерпретации данной формулы воспользуемся следующим определением экспоненты комплексного числа:

$$e^{iu} = \cos(u) + i \sin(u) \ .$$

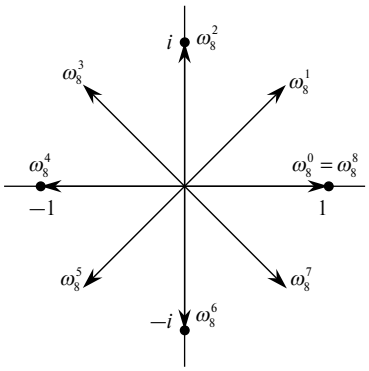


Рис. 30.2. Расположение значений $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ на комплексной плоскости, где $\omega_8 = e^{2\pi i/8}$ — главное значение корня восьмой степени из единицы

На рис. 30.2 показано, что n комплексных корней из единицы равномерно распределены по окружности единичного радиуса с центром в начале координат комплексной плоскости. Значение

$$\omega_n = e^{2\pi i/n} \tag{30.6}$$

называется ***главным значением корня n-й степени из единицы*** (the principal nth root of unity); все остальные комплексные корни n-й степени из единицы являются его степенями[2].

Указанные n комплексных корней n-й степени из единицы

$$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$$

образуют группу относительно операции умножения (см. раздел 31.1). Эта группа имеет ту же структуру, что и аддитивная группа $(\mathbf{Z}_n, +)$ по модулю n, поскольку из $\omega_n^n = \omega_n^0 = 1$ следует, что $\omega_n^j \omega_n^k = \omega_n^{(j+k) \bmod n}$. Аналогично, $\omega_n^{-1} = \omega_n^{n-1}$. Основные свойства комплексных корней n-й степени из единицы приведены в следующих леммах.

Лемма 30.3 (Лемма о сокращении). Для любых целых чисел $n \geqslant 0$, $k \geqslant 0$ и $d > 0$

$$\omega_{dn}^{dk} = \omega_n^k \,. \tag{30.7}$$

[2]Многие авторы определяют ω_n иначе: $\omega_n = e^{-2\pi i/n}$. Это альтернативное определение обычно используется в обработке сигналов. Лежащие в основе математические концепции в основном одинаковы для обоих определений.

Доказательство. Доказательство непосредственно вытекает из уравнения (30.6), поскольку

$$\omega_{dn}^{dk} = \left(e^{2\pi i/dn}\right)^{dk} = \left(e^{2\pi i/n}\right)^{k} = \omega_n^k \ .$$

■

Следствие 30.4. Для любого четного целого $n > 0$

$$\omega_n^{n/2} = \omega_2 = -1 \ .$$

Доказательство. Доказательство предлагается провести самостоятельно, выполнив упражнение 30.2-1. ■

Лемма 30.5 (Лемма о делении пополам). Если $n > 0$ четное, то квадраты n комплексных корней n-й степени из единицы представляют собой $n/2$ корней $n/2$-ой степени из единицы.

Доказательство. Согласно лемме о сокращении, для любого неотрицательного целого k справедливо $\left(\omega_n^k\right)^2 = \omega_{n/2}^k$. Заметим, что если возвести в квадрат все комплексные корни n-й степени из единицы, то каждый корень $n/2$-й степени из единицы будет получен в точности дважды, поскольку

$$\left(\omega_n^{k+n/2}\right)^2 = \omega_n^{2k+n} = \omega_n^{2k}\omega_n^n = \omega_n^{2k} = \left(\omega_n^k\right)^2 \ .$$

Таким образом, квадраты ω_n^k и $\omega_n^{k+n/2}$ одинаковы. Это свойство можно также доказать с помощью следствия 30.4: из $\omega_n^{n/2} = -1$ вытекает $\omega_n^{kn/2} = -\omega_n^k$, следовательно, $\left(\omega_n^{k+n/2}\right)^2 = \left(\omega_n^k\right)^2$. ■

Как мы увидим далее, лемма о делении пополам играет исключительно важную роль в применении декомпозиции для преобразования полиномов из коэффициентного представления в форму точки–значения и обратно, поскольку она гарантирует, что рекурсивные подзадачи имеют половинную размерность.

Лемма 30.6 (Лемма о суммировании). Для любого целого $n \geqslant 1$ и ненулевого целого k, не кратного n,

$$\sum_{j=0}^{n-1} \left(\omega_n^k\right)^j = 0 \ .$$

Доказательство. Уравнение (A.5) применимо как к действительным, так и к комплексным значениям, поэтому

$$\sum_{j=0}^{n-1} \left(\omega_n^k\right)^j = \frac{\left(\omega_n^k\right)^n - 1}{\omega_n^k - 1} = \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} = \frac{(1)^k - 1}{\omega_n^k - 1} = 0 \ .$$

Требование, что k не делится на n, гарантирует, что знаменатель не равен 0, поскольку $\omega_n^k = 1$ только тогда, когда k делится на n. ■

Дискретное преобразование Фурье

Напомним, что мы хотим вычислить полином

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \ ,$$

степень которого не выше n, в точках $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ (которые представляют собой n комплексных корней n-й степени из единицы)[3]. Без потери общности можно предположить, что n является степенью 2, поскольку заданную границу степени всегда можно увеличить, добавив, если необходимо, старшие нулевые коэффициенты[4]. Предположим, что полином A задан в коэффициентной форме: $a = (a_0, a_1, \ldots, a_{n-1})$. Определим конечные результаты y_k, $k = 0, 1, \ldots, n-1$, с помощью формулы

$$y_k = A\left(\omega_n^k\right) = \sum_{j=0}^{n-1} a_j \omega_n^{kj} \ . \tag{30.8}$$

Вектор $y = (y_0, y_1, \ldots, y_{n-1})$ представляет собой ***дискретное преобразование Фурье (ДПФ)*** (Discrete Fourier Transform, DFT) вектора коэффициентов $a = (a_0, a_1, \ldots, a_{n-1})$. Это можно также записать как $y = \mathrm{DFT}_n(a)$.

Быстрое преобразование Фурье

С помощью метода ***быстрого преобразования Фурье (БПФ)*** (Fast Fourier Transform, FFT), основанного на использовании специальных свойств комплексных корней из единицы, $\mathrm{DFT}_n(a)$ можно вычислять за время $\Theta(n \lg n)$, в отличие от метода непосредственного преобразования.

В методе БПФ применяется стратегия декомпозиции, в которой отдельно используются коэффициенты полинома $A(x)$ с четными и нечетными индексами,

[3] Здесь n на самом деле представляет собой величину, которая в разделе 30.1 обозначалась как $2n$, поскольку прежде чем выполнять вычисление, пределы степени заданных полиномов были удвоены. Таким образом, при умножении полиномов речь в действительности идет о комплексных корнях из единицы $2n$-й степени.

[4] При использовании БПФ в обработке сигналов добавлять нулевые коэффициенты в целях получения степеней 2 не рекомендуется, так как это приводит к возникновению высокочастотных артефактов. Одним из методов получения размерности, равной степени 2, в обработке сигналов является **отражение** (mirroring). Пусть n' — наименьшая целая степень 2, превышающая n; тогда одним из способов отражения является задание $a_{n+j} = a_{n-j-2}$ для $j = 0, 1, \ldots, n' - n - 1$.

чтобы определить два новых полинома $A^{[0]}(x)$ и $A^{[1]}(x)$ степени не выше $n/2$:

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} ,$$
$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} .$$

Заметим, что $A^{[0]}$ содержит все коэффициенты $A(x)$ с четными индексами (двоичное представление этих индексов заканчивается цифрой 0), а $A^{[1]}$ содержит все коэффициенты с нечетными индексами (двоичное представление которых заканчивается цифрой 1). Для определенных таким способом полиномов справедливо равенство

$$A(x) = A^{[0]}\left(x^2\right) + xA^{[1]}\left(x^2\right) , \tag{30.9}$$

так что задача вычисления $A(x)$ в точках $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ сводится к следующим задачам:

1. вычислить два полинома $A^{[0]}(x)$ и $A^{[1]}(x)$ степени не выше $n/2$ в точках

$$\left(\omega_n^0\right)^2, \left(\omega_n^1\right)^2, \ldots, \left(\omega_n^{n-1}\right)^2 , \tag{30.10}$$

   а затем

2. объединить результаты с использованием формулы (30.9).

Согласно лемме о делении пополам, список значений (30.10) сдержит уже не n различных значений, а только $n/2$ комплексных корней степени $(n/2)$ из единицы, причем каждый корень встречается в списке ровно два раза. Следовательно, полиномы $A^{[0]}$ и $A^{[1]}$ с границей степени $n/2$ рекурсивно вычисляются в $n/2$ комплексных корнях $n/2$-й степени из единицы. Эти подзадачи имеют точно такой же вид, как и исходная задача, но их размерность вдвое меньше. Таким образом, мы свели вычисление n-элементного DFT_n к вычислению двух $n/2$-элементных $\mathrm{DFT}_{n/2}$. Такая декомпозиция является основой следующего рекурсивного алгоритма БПФ, который вычисляет ДПФ n-элементного вектора $a = (a_0, a_1, \ldots, a_{n-1})$, где n является степенью 2.

```
RECURSIVE_FFT(a)
 1  n ← length[a]        ▷ n является степенью 2
 2  if n = 1
 3     then return a
 4  ω_n ← e^{2πi/n}
 5  ω ← 1
 6  a^[0] ← (a_0, a_2, ..., a_{n-2})
 7  a^[1] ← (a_1, a_3, ..., a_{n-1})
 8  y^[0] ← RECURSIVE_FFT(a^[0])
 9  y^[1] ← RECURSIVE_FFT(a^[1])
```

```
10  for k ← 0 to n/2 − 1
11      do y_k ← y_k^[0] + ω y_k^[1]
12         y_{k+(n/2)} ← y_k^[0] − ω y_k^[1]
13         ω ← ω ω_n
14  return y                ▷ Предполагается, что y — вектор-столбец
```

Процедура RECURSIVE_FFT работает следующим образом. Строки 2–3 описывают базис данной рекурсии: ДПФ одного элемента является самим этим элементом, следовательно, в данном случае

$$y_0 = a_0\omega_1^0 = a_0 \cdot 1 = a_0 \ .$$

Строки 6–7 определяют векторы коэффициентов полиномов $A^{[0]}$ и $A^{[1]}$. Строки 4, 5 и 13 гарантируют, что ω обновляется надлежащим образом, т.е. всякий раз, когда выполняются строки 11–12, $\omega = \omega_n^k$. (Сохранение текущего значения ω от итерации к итерации позволяет экономить время по сравнению с многократным вычислением ω_n^k с нуля с помощью цикла **for**.) В строках 8–9 выполняются рекурсивные вычисления $\mathrm{DFT}_{n/2}$, при этом получаются следующие значения ($k = 0, 1, \ldots, n/2 - 1$):

$$y_k^{[0]} = A^{[0]}\left(\omega_{n/2}^k\right) \ ,$$
$$y_k^{[1]} = A^{[1]}\left(\omega_{n/2}^k\right) \ ,$$

или, поскольку согласно лемме о сокращении, $\omega_{n/2}^k = \omega_n^{2k}$,

$$y_k^{[0]} = A^{[0]}\left(\omega_n^{2k}\right) \ ,$$
$$y_k^{[1]} = A^{[1]}\left(\omega_n^{2k}\right) \ .$$

В строках 11–12 комбинируются результаты рекурсивных вычислений $\mathrm{DFT}_{n/2}$. Для значений $y_0, y_1, \ldots, y_{n/2-1}$ строка 11 дает:

$$y_k = y_k^{[0]} + \omega_n^k y_k^{[1]} = A^{[0]}\left(\omega_n^{2k}\right) + \omega_n^k A^{[1]}\left(\omega_n^{2k}\right) = A\left(\omega_n^k\right) \ ,$$

где последнее равенство следует из уравнения (30.9).

Значения $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$ вычисляются в строке 12. Для значений $k = 0, 1, \ldots, n/2 - 1$ получаем:

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} = y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} = \\ &= A^{[0]}\left(\omega_n^{2k}\right) + \omega_n^{k+(n/2)} A^{[1]}\left(\omega_n^{2k}\right) = \\ &= A^{[0]}\left(\omega_n^{2k+n}\right) + \omega_n^{k+(n/2)} A^{[1]}\left(\omega_n^{2k+n}\right) = \\ &= A\left(\omega_n^{k+(n/2)}\right) \ , \end{aligned}$$

где второе равенство следует из $\omega_n^{k+(n/2)} = -\omega_n^k$, четвертое — поскольку $\omega_n^{2k+n} = \omega_n^{2k}$, а последнее — из уравнения (30.9). Таким образом, возвращаемый процедурой RECURSIVE_FFT вектор y действительно является ДПФ исходного вектора a.

Внутри цикла **for** в строках 10–13 каждое значение $y_k^{[1]}$ умножается на ω_n^k для всех $k = 0, 1, \ldots, n/2 - 1$. Полученное произведение прибавляется к $y_k^{[0]}$ и вычитается из него. Поскольку каждый множитель ω_n^k используется как с положительным, так и с отрицательным знаком, множители ω_n^k называются ***поворачивающими множителями*** (twiddle factors).

Чтобы определить время работы процедуры RECURSIVE_FFT, заметим, что, за исключением рекурсивных вызовов, каждый вызов занимает время $\Theta(n)$, где n — длина исходного вектора. Таким образом, рекуррентное соотношение для времени выполнения выглядит следующим образом:

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n) \ .$$

Итак, с помощью быстрого преобразования Фурье можно вычислить полином степени не выше n в точках, являющихся комплексными корнями n-й степени из единицы, за время $\Theta(n \lg n)$.

Интерполяция в точках, являющихся комплексными корнями из единицы

Чтобы завершить рассмотрение схемы умножения полиномов, покажем, как выполняется интерполяция комплексных корней из единицы некоторым полиномом, что позволяет перейти от формы значений в точках обратно к коэффициентной форме. Для осуществления интерполяции ДПФ записывается в виде матричного уравнения, после чего выполняется поиск обратной матрицы.

Из уравнения (30.4) можно записать ДПФ как произведение матриц $y = V_n a$, где V_n — матрица Вандермонда, содержащая соответствующие степени ω_n:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} .$$

Для выполнения обратной операции, которая записывается как $a = \mathrm{DFT}_n^{-1}(y)$, необходимо умножить y на матрицу V_n^{-1}, обратную V_n.

Теорема 30.7. Для $j, k = 0, 1, \ldots, n-1$, (j,k)-й элемент матрицы V_n^{-1} равен ω_n^{-kj}/n.

Доказательство. Покажем, что $V_n^{-1}V_n = I_n$, единичной матрице размера $n \times n$. Рассмотрим (j, j')-й элемент матрицы $V_n^{-1}V_n$:

$$\left[V_n^{-1}V_n\right]_{jj'} = \sum_{k=0}^{n-1} \left(\omega_n^{-kj}/n\right)\left(\omega_n^{kj'}\right) = \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n \ .$$

Согласно лемме о суммировании (лемма 30.6), данная сумма равна 1, если $j' = j$, и 0 в противном случае. Заметим, что поскольку $-(n-1) \leqslant j' - j \leqslant n-1$, $j' - j$ не кратно n, так что лемма применима. ■

Если дана обратная матрица V_n^{-1}, обратное дискретное преобразование Фурье $\mathrm{DFT}_n^{-1}(y)$ вычисляется по формуле

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{30.11}$$

для $j = 0, 1, \ldots, n-1$. Сравнивая уравнения (30.8) и (30.11), мы видим, что если модифицировать алгоритм БПФ — поменять ролями a и y, заменить ω_n на ω_n^{-1} и разделить каждый элемент результата на n — получится обратное ДПФ (см. упражнение 30.2-4). Таким образом, DFT_n^{-1} также можно вычислить за время $\Theta(n \lg n)$.

Таким образом, с помощью прямого и обратного БПФ можно преобразовывать полином степени не выше n из коэффициентной формы в форму значений в точках и обратно за время $\Theta(n \lg n)$. Применительно к умножению полиномов, мы показали следующее.

Теорема 30.8 (Теорема о свертке). Для любых двух векторов a и b длины n, где n является степенью 2, справедливо соотношение

$$a \otimes b = \mathrm{DFT}_{2n}^{-1}\left(\mathrm{DFT}_{2n}(a) \cdot \mathrm{DFT}_{2n}(b)\right) \ ,$$

где векторы a и b дополняются нулями до длины $2n$, а "$\cdot$" обозначает покомпонентное произведение двух $2n$-элементных векторов. ■

Упражнения

30.2-1. Докажите следствие 30.4.

30.2-2. Вычислите ДПФ вектора $(0, 1, 2, 3)$.

30.2-3. Выполните упражнение 30.1-1, используя схему с временем выполнения $\Theta(n \lg n)$.

30.2-4. Напишите псевдокод для вычисления DFT_n^{-1} за время $\Theta(n \lg n)$.

30.2-5. Опишите обобщение процедуры БПФ для случая, когда n является степенью 3. Приведите рекуррентное соотношение для времени выполнения и решите его.

⋆ 30.2-6. Предположим, что вместо выполнения n-элементного БПФ над полем комплексных чисел (где n четно), мы используем кольцо $\mathbf{Z}_m$ целых чисел по модулю $m = 2^{tn/2+1}$, где t — произвольное положительное целое число. Используйте $\omega = 2^t$ вместо ω_n в качестве главного значения корня n-й степени из единицы по модулю m. Докажите, что прямое и обратное ДПФ в этой системе является вполне определенным.

30.2-7. Покажите, как для заданного списка значений $z_0, z_1, \ldots, z_{n-1}$ (возможно, с повторениями) найти коэффициенты полинома $P(x)$ степени не выше $n+1$, который имеет нули только в точках $z_0, z_1, \ldots, z_{n-1}$ (возможно, с повторениями). Процедура должна выполняться за время $O\left(n \lg^2 n\right)$. (*Указание*: полином $P(x)$ имеет нулевое значение в точке z_j тогда и только тогда, когда $P(x)$ кратен $(x - z_j)$.)

⋆ 30.2-8. ***Чирп-преобразованием*** (chirp transform) некоторого вектора $a = (a_0, a_1, \ldots, a_{n-1})$ является вектор $y = (y_0, y_1, \ldots, y_{n-1})$, где $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$, а z — некоторое комплексное число. Таким образом, ДПФ является частным случаем чирп-преобразования, полученного для $z = \omega_n$. Докажите, что для любого комплексного числа z чирп-преобразование можно вычислить за время $O(n \lg n)$. (*Указание*: воспользуйтесь уравнением

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} \left(a_j z^{j^2/2}\right) \left(z^{-(k-j)^2/2}\right),$$

чтобы рассматривать чирп-преобразование как свертку.)

30.3 Эффективные реализации БПФ

Поскольку практические приложения ДПФ, такие как обработка сигналов, требуют максимальной скорости, в данном разделе мы рассмотрим две наиболее эффективные реализации БПФ. Сначала будет описана итеративная версия алгоритма БПФ, время выполнения которой составляет $\Theta(n \lg n)$, однако при этом в Θ скрыта меньшая константа, чем в случае рекурсивной реализации, предложенной в разделе 30.2. После этого мы вернемся к интуитивным соображениям, приведшим нас к итеративной реализации, для разработки эффективной параллельной схемы БПФ.

Итеративная реализация БПФ

Прежде всего заметим, что в цикле **for** в строках 10–13 процедуры RECURSIVE_FFT значение $\omega_n^k y_k^{[1]}$ вычисляется дважды. В терминологии компиляторов такое значение называется ***общим подвыражением*** (common subexpression). Можно изменить цикл так, чтобы вычислять это значение только один раз, сохраняя его во временной переменной t:

for $k \leftarrow 0$ **to** $n/2 - 1$
  do $t \leftarrow \omega\, y_k^{[1]}$
    $y_k \leftarrow y_k^{[0]} + t$
    $y_{k+(n/2)} \leftarrow y_k^{[0]} - t$
    $\omega \leftarrow \omega\, \omega_n$

Выполняемая в данном цикле операция (умножение $\omega_n^k y_k^{[1]}$, сохранение произведения в переменной t, прибавление и вычитание t из $y_k^{[0]}$), известна как ***преобразование бабочки*** (butterfly operation), схематически представленное на рис. 30.3.

Теперь покажем, как сделать структуру алгоритма БПФ итеративной, а не рекурсивной. На рис. 30.4 мы организовали входные векторы рекурсивных вызовов в обращении к процедуре RECURSIVE_FFT в древовидную структуру, в которой первый вызов производится для $n = 8$. Данное дерево содержит по одному узлу для каждого вызова процедуры, помеченному соответствующим входным вектором. Каждое обращение к процедуре RECURSIVE_FFT производит два рекурсивных вызова, пока не получит 1-элементный вектор. Мы сделали первый вызов левым дочерним узлом, а второй вызов — правым.

Посмотрев на дерево, можно заметить, что если удастся организовать элементы исходного вектора a в том порядке, в котором они появляются в листьях дерева, то выполнение процедуры RECURSIVE_FFT можно будет представить следующим образом. Берутся пары элементов, с помощью одного преобразования бабочки вычисляется ДПФ каждой пары, и пара заменяется своим ДПФ. В результате получается вектор, содержащий $n/2$ 2-элементных ДПФ. Затем эти ДПФ

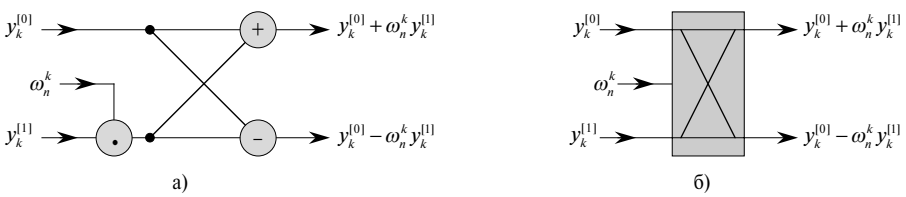


Рис. 30.3. Преобразование бабочки. а) Слева поступают два вводимых значения, ω_n^k умножается на $y_k^{[1]}$, соответствующие сумма и разность выводятся справа. б) Упрощенная схема преобразования бабочки, используемая в параллельной схеме БПФ

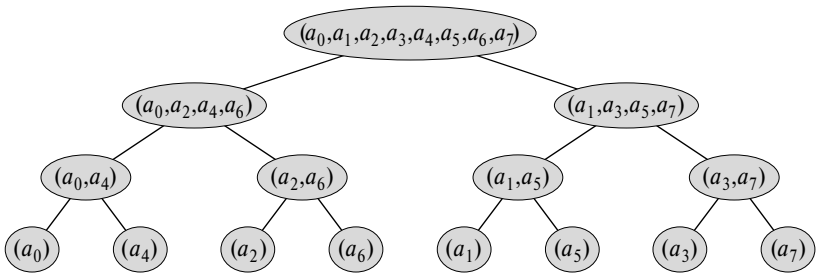


Рис. 30.4. Дерево входных векторов рекурсивных вызовов процедуры RECURSIVE_FFT. Начальный вызов производится для $n = 8$

объединяются в пары, с помощью двух преобразований бабочки вычисляются ДПФ для каждых четырех элементов вектора, объединенных в четверки, после чего два 2-элементных ДПФ заменяются одним 4-элементным ДПФ. Теперь вектор содержит $n/4$ 4-элементных ДПФ. Процесс продолжается до тех пор, пока не получится два $n/2$-элементных ДПФ, которые с помощью $n/2$ преобразований бабочки можно объединить в конечное n-элементное ДПФ.

Чтобы превратить описанную последовательность действий в код, используем массив $A\,[0..n-1]$, который первоначально содержит элементы исходного вектора a в том порядке, в котором они перечислены в листовых узлах дерева на рис. 30.4. (Позже мы покажем, как определить этот порядок, который называется ***поразрядно обратной перестановкой*** (bit-reversal permutation).) Поскольку объединение в пары необходимо выполнять на каждом уровне дерева, введем в качестве счетчика уровней переменную s, которая изменяется от 1 (на нижнем уровне, где составляются пары для вычисления 2-элементных ДПФ) до $\lg n$ (в вершине, где объединяются два $n/2$-элементных ДПФ для получения окончательного результата). Таким образом, алгоритм имеет следующую структуру:

```
1  for s ← 1 to lg n
2      do for k ← 0 to n − 1 by 2^s
3             do Объединяем два 2^(s−1)-элементных ДПФ из
                    A[k..k + 2^(s−1) − 1] и A[k + 2^(s−1)..k + 2^s − 1]
                    в одно 2^s-элементное ДПФ в A[k..k + 2^s − 1]
```

Тело цикла (строка 3) можно более точно описать с помощью псевдокода. Копируем цикл **for** из процедуры RECURSIVE_FFT, отождествив $y^{[0]}$ с $A[k..k + 2^{s-1} - - 1]$ и $y^{[1]}$ с $A\left[k + 2^{s-1}..k + 2^s - 1\right]$. Поворачивающий множитель, используемый в каждом преобразовании бабочки, зависит от значения s; это степень ω_m, где $m = 2^s$. (Мы ввели переменную m исключительно для того, чтобы формула была удобочитаемой.) Введем еще одну временную переменную u, которая позволит нам выполнять преобразование бабочки на месте, без привлечения до-

полнительной памяти. Заменив строку 3 в общей схеме телом цикла, получим следующий псевдокод, образующий базис параллельной реализации, которая будет представлена далее. В данном коде сначала вызывается вспомогательная процедура BIT_REVERSE_COPY(a, A), копирующая вектор a в массив A в том порядке, в котором нам нужны эти значения.

```
ITERATIVE_FFT(a)
 1  BIT_REVERSE_COPY(a, A)
 2  n ← length[a]            ▷ n является степенью 2.
 3  for s ← 1 to lg n
 4       do m ← 2^s
 5          ω_m ← e^{2πi/m}
 6          for k ← 0 to n − 1 by m
 7               do ω ← 1
 8                  for j ← 0 to m/2 − 1
 9                       do t ← ω A[k + j + m/2]
10                          u ← A[k + j]
11                          A[k + j] ← u + t
12                          A[k + j + m/2] ← u − t
13                          ω ← ω ω_m
```

Каким образом процедура BIT_REVERSE_COPY размещает элементы входного вектора a в массиве A в требуемом порядке? Порядок следования листов на рис. 30.4 является ***обратной перестановкой***, или ***реверсом битов*** (bit-reversal permutation), т.е. если $\mathrm{rev}\,(k) - n$-битовое целое число, образованное путем размещения битов исходного двоичного представления k в обратном порядке ("задом наперед"), то элемент a_k следует поместить в массиве на место $A\,[\mathrm{rev}\,(k)]$. На рис. 30.4, например, листовые узлы следуют в таком порядке: 0, 4, 2, 6, 1, 5, 3, 7; в двоичном представлении эта последовательность записывается как 000, 100, 010, 110, 001, 101, 011, 111; после записи битов каждого значения в обратном порядке получается обычная числовая последовательность: 000, 001, 010, 011, 100, 101, 110, 111. Чтобы убедиться, что нам нужна именно обратная перестановка битов, заметим, что на верхнем уровне дерева индексы, младший бит которых равен 0, помещаются в левое поддерево, а индексы, младший бит которых равен 1, помещаются в правое поддерево. Исключая на каждом уровне младший бит, мы продолжаем процесс вниз по дереву, пока не получим в листовых узлах порядок, заданный обратной перестановкой битов.

Поскольку функция $\mathrm{rev}\,(k)$ легко вычисляется, процедуру BIT_REVERSE_COPY можно записать следующим образом:

```
BIT_REVERSE_COPY(a, A)
1  n ← length[a]
2  for k ← 0 to n − 1
3        do A[rev(k)] ← a_k
```

Итеративная реализация БПФ выполняется за время $\Theta(n \lg n)$. Обращение к процедуре BIT_REVERSE_COPY(a,A) выполняется за время $O(n \lg n)$, поскольку проводится n итераций, а целое число от 0 до $n-1$, содержащее $\lg n$ битов, можно преобразовать к обратному порядку за время $O(\lg n)$[5]. (На практике мы обычно заранее знаем начальное значение n, поэтому можно составить таблицу, отображающую k в $\mathrm{rev}(k)$, в результате процедура BIT_REVERSE_COPY будет выполняться за время $\Theta(n)$ с малой скрытой константой. В качестве альтернативного подхода можно использовать схему амортизированного обратного бинарного битового счетчика, описанную в задаче 17-1.) Чтобы завершить доказательство того факта, что процедура ITERATIVE_FFT выполняется за время $\Theta(n \lg n)$, покажем, что число $L(n)$ выполнений тела самого внутреннего цикла (строки 8–13) составляет $\Theta(n \lg n)$. Цикл **for** (строки 6–13) повторяется $n/m = n/2^s$ раз для каждого значения s, а внутренний цикл (строки 8–13) повторяется $m/2 = 2^{s-1}$ раз. Отсюда

$$L(n) = \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} = \sum_{s=1}^{\lg n} \frac{n}{2} = \Theta(n \lg n) \ .$$

Параллельная схема БПФ

Чтобы получить эффективный параллельный алгоритм БПФ, можно использовать многие свойства, которые позволили нам реализовать эффективный итеративный алгоритм БПФ. Мы будем представлять параллельный алгоритм БПФ в виде схемы, которая похожа на схемы сравнения, описанные в главе 27. Вместо компараторов в схеме БПФ используются преобразования бабочки, показанные на рис. 30.3*б*. В данном случае также применимо понятие глубины, введенное нами в главе 27. Схема процедуры PARALLEL_FFT, вычисляющей БПФ для n введенных значений при $n = 8$, показана на рис. 30.5. Она начинается с поразрядной обратной перестановки исходных значений, затем следует $\lg n$ этапов, на каждом из которых параллельно производится $n/2$ преобразований бабочки. Таким образом, глубина схемы составляет $\Theta(\lg n)$.

На рисунке каждое преобразование бабочки получает в качестве исходных значения, поступающие по двум проводам, вместе с поворачивающим множителем, и передает полученные значения на два выходящих провода. Различные этапы каскада преобразований бабочки нумеруются в соответствии с итерациями

[5]Интересные реализации реверса битов рассматриваются в разделе 7.1 книги Г. Уоррен. *Алгоритмические трюки для программистов*. — М.: Издательский дом "Вильямс", 2003. — Прим. ред.

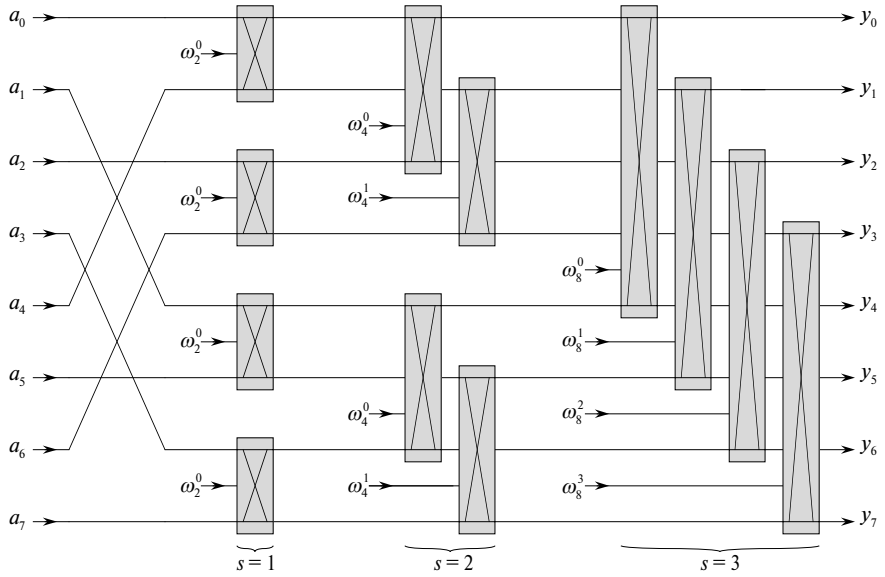


Рис. 30.5. Схема процедуры PARALLEL_FFT, вычисляющей БПФ при $n = 8$.

самого внешнего цикла процедуры ITERATIVE_FFT. Только верхний и нижний провода, проходя через "бабочку", участвуют в вычислениях; провода, проходящие через середину "бабочки", никак не взаимодействуют с ней, и их значения также не меняются данной "бабочкой". Например, верхняя бабочка на этапе 2 ничего не делает с проводом 1 (которому соответствует переменная вывода, обозначенная y_1); ее переменные ввода и вывода находятся на проводах 0 и 2 (обозначенных y_0 и y_2, соответственно). Для вычисления БПФ для n переменных ввода требуется схема глубиной $\Theta(\lg n)$, содержащая $\Theta(n \lg n)$ преобразований бабочки.

Левая часть схемы PARALLEL_FFT выполняет поразрядно обратную перестановку, а остальная часть представляет собой итеративную процедуру ITERATIVE_FFT. Поскольку при каждом повторении внешнего цикла **for** выполняется $n/2$ независимых преобразований бабочки, в данной схеме они выполняются параллельно. Значение s в каждой итерации ITERATIVE_FFT соответствует каскаду преобразований бабочки, показанному на рис. 30.5. В пределах этапа $s = 1, 2, \ldots, \lg n$ имеется $n/2^s$ групп преобразований бабочки (соответствующих различным значениям k процедуры ITERATIVE_FFT), в каждой группе выполняется 2^{s-1} операций (соответствующих различным значениям j процедуры ITERATIVE_FFT). Преобразования бабочки, показанные на рис. 30.5, соответствуют преобразованиям во внутреннем цикле (строки 9–12 процедуры ITERATIVE_FFT). Заметим также, что используемые в "бабочках" поворачивающие множители соответствуют

поворачивающим множителям, используемым в процедуре ITERATIVE_FFT: на этапе s используются значения $\omega_m^0, \omega_m^1, \ldots, \omega_m^{m/2-1}$, где $m = 2^s$.

Упражнения

30.3-1. Покажите, как процедура ITERATIVE_FFT вычисляет ДПФ исходного вектора $(0, 2, 3, -1, 4, 5, 7, 9)$.

30.3-2. Покажите, как реализовать алгоритм БПФ, в котором обратная перестановка битов выполняется в конце, а не в начале процесса вычислений. (*Указание*: рассмотрите обратное ДПФ.)

30.3-3. Сколько раз процедура ITERATIVE_FFT вычисляет поворачивающие множители на каждом этапе? Перепишите процедуру ITERATIVE_FFT так, чтобы поворачивающие множители на этапе s вычислялись только 2^{s-1} раз.

⋆ 30.3-4. Предположим, что сумматоры в преобразованиях бабочки схемы БПФ иногда дают сбои, приводящие к нулевому результату независимо от подаваемых на вход значений. Предположим, что сбой произошел в точности в одном сумматоре, однако не известно, в каком именно. Опишите, как можно быстро выявить неисправный сумматор путем тестирования всей БПФ-схемы с помощью различных вводов и изучения выводов. Насколько эффективен этот метод?

Задачи

30-1. **Умножение посредством декомпозиции**

а) Покажите, как умножить два линейных полинома $ax + b$ и $cx + d$, используя только три операции умножения. (*Указание*: одно из умножений $(a + b) \cdot (c + d)$.)

б) Приведите два алгоритма декомпозиции для умножения полиномов степени не выше n, время выполнения которых $\Theta\left(n^{\lg 3}\right)$. В первом алгоритме следует разделить коэффициенты исходного полинома на старшие и младшие, а во втором — на четные и нечетные.

в) Покажите, что два n-битовых целых числа можно умножить за $O\left(n^{\lg 3}\right)$ шагов, где каждый шаг оперирует с однобитовыми значениями, количество которых не превышает некую константу.

30-2. **Матрицы Теплица**

Матрицей Теплица (Toeplitz matrix) называется матрица $A = (a_{ij})$ размером $n \times n$, в которой $a_{ij} = a_{i-1,j-1}$ для $i, j = 2, 3, \ldots, n$.

а) Всегда ли сумма двух матриц Теплица является матрицей Теплица? Что можно сказать об их произведении?

б) Каким должно быть представление матриц Теплица, чтобы сложение двух матриц Теплица размером $n \times n$ можно было выполнить за время $O(n)$?

в) Предложите алгоритм умножения матрицы Теплица размером $n \times$ $\times n$ на вектор длины n со временем работы $O(n \lg n)$. Воспользуйтесь представлением, полученным при решении предыдущего пункта данной задачи.

г) Предложите эффективный алгоритм умножения двух матриц Теплица размером $n \times n$ и оцените время его выполнения.

30-3. **Многомерное быстрое преобразование Фурье**

Можно обобщить одномерное дискретное преобразование Фурье, определенное уравнением (30.8), на d-мерный случай. Пусть на вход подается d-мерный массив $A = (a_{i_1,i_2,\ldots,i_d})$, размерности $n_1, n_2, \ldots, n_d$ которого удовлетворяют соотношению $n_1 n_2 \cdots n_d = n$. Определим d-мерное дискретное преобразование Фурье следующим образом:

$$y_{k_1,k_2,\ldots,k_d} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1,j_2,\ldots,j_d} \omega_{n_1}^{j_1 k_1} \omega_{n_2}^{j_2 k_2} \cdots \omega_{n_d}^{j_d k_d}$$

для $0 \leqslant k_1 < n_1$, $0 \leqslant k_2 < n_2$, $\ldots$, $0 \leqslant k_d < n_d$.

а) Покажите, что можно вычислить d-мерное дискретное преобразование Фурье путем поочередного вычисления одномерных ДПФ. Таким образом, сначала вычисляется n/n_1 отдельных одномерных ДПФ вдоль первого измерения. Затем, используя результат ДПФ вдоль первого измерения в качестве ввода, вычисляется n/n_2 одномерных ДПФ вдоль второго измерения. Используя этот результат в качестве ввода, вычисляется n/n_3 отдельных ДПФ вдоль третьего измерения и т.д., до измерения d.

б) Покажите, что порядок следования измерений не имеет значения, т.е. можно вычислять d-мерное ДПФ путем вычисления одномерных ДПФ для каждого из d измерений в произвольном порядке.

в) Покажите, что если каждое одномерное ДПФ вычислять с помощью быстрого преобразования Фурье, то суммарное время вычисления d-мерного ДПФ составляет $O(n \lg n)$ независимо от d.

30-4. **Вычисление всех производных полинома в определенной точке**

Пусть задан полином $A(x)$ степени не выше n; t-я производная этого полинома определяется формулой

$$A^{(t)}(x) = \begin{cases} A(x) & \text{если } t = 0, \\ \frac{d}{dx} A^{(t-1)}(x) & \text{если } 1 \leqslant t \leqslant n-1, \\ 0 & \text{если } t \geqslant n. \end{cases}$$

Пусть заданы коэффициентное представление $(a_0, a_1, \ldots, a_{n-1})$ полинома $A(x)$ и некая точка x_0. Требуется найти $A^{(t)}(x_0)$ для $t = 0, 1, \ldots n-1$.

а) Пусть заданы коэффициенты $b_0, b_1, \ldots, b_{n-1}$, такие что

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j .$$

Покажите, как вычислить $A^{(t)}(x_0)$ для $t = 0, 1, \ldots n-1$ за время $O(n)$.

б) Объясните, как найти $b_0, b_1, \ldots, b_{n-1}$ за время $O(n \lg n)$, если даны значения $A\left(x_0 + \omega_n^k\right)$ для $k = 0, 1, \ldots n-1$.

в) Докажите, что

$$A\left(x + \omega_n^k\right) = \sum_{r=0}^{n-1} \left(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j)\, g(r-j) \right) ,$$

где $f(j) = a_j \cdot j!$, а

$$g(l) = \begin{cases} x_0^{-l}/(-l)! & \text{если } -(n-1) \leqslant l \leqslant 0, \\ 0 & \text{если } 1 \leqslant l \leqslant (n-1). \end{cases}$$

г) Объясните, как вычислить $A\left(x_0 + \omega_n^k\right)$ для $k = 0, 1, \ldots n-1$ за время $O(n \lg n)$. Обоснуйте, что все нетривиальные производные полинома $A(x)$ в точке x_0 можно вычислить за время $O(n \lg n)$.

30-5. **Вычисление полинома в нескольких точках**

Как отмечалось, задачу вычисления полинома степени не выше n в единственной точке с помощью схемы Горнера можно решить за время $O(n)$. Мы также показали, что с помощью БПФ такой полином можно вычислить во всех n комплексных корнях единицы за время $O(n \lg n)$. Теперь покажем, как вычислить полином степени не выше n в произвольных n точках за время $O\left(n \lg^2 n\right)$.

Для этого мы воспользуемся тем фактом, что при делении одного такого полинома на другой остаток можно вычислить за время $O(n \lg n)$ (этот результат мы принимаем без доказательства). Например, остаток при делении $3x^3 + x^2 - 3x + 1$ на $x^2 + x + 2$ равен

$$\left(3x^3 + x^2 - 3x + 1\right) \bmod \left(x^2 + x + 2\right) = -7x + 5 .$$

Пусть заданы коэффициентное представление полинома $A(x) = \sum_{k=0}^{n-1} a_k x^k$ и n точек $x_0, x_1, \ldots, x_{n-1}$, и нам необходимо вычислить n значений $A(x_0), A(x_1), \ldots, A(x_{n-1})$. Для $0 \leqslant i \leqslant j \leqslant n-1$ определим полиномы $P_{ij}(x) = \prod_{k=i}^{j} (x - x_k)$ и $Q_{ij} = A(x) \bmod P_{ij}(x)$. Заметим, что при этом $Q_{ij}(x)$ имеет степень не выше $j - i$.

а) Докажите, что $A(x) \bmod (x - z) = A(z)$ в любой точке z.

б) Докажите, что $Q_{kk}(x) = A(x_k)$, а $Q_{0,n-1}(x) = A(x)$.

в) Докажите, что для $i \leqslant k \leqslant j$ справедливы соотношения $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$ и $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.

г) Предложите алгоритм вычисления $A(x_0), A(x_1), \ldots, A(x_{n-1})$ со временем работы $O\left(n \lg^2 n\right)$.

30-6. **БПФ с использованием модульной арифметики**

Согласно определению, при вычислении дискретного преобразования Фурье требуется использовать комплексные числа, что может привести к потере точности из-за ошибок округления. Для некоторых задач известно, что ответы содержат только целые значения, поэтому желательно использовать вариант БПФ, основанный на модульной арифметике, чтобы гарантировать, что ответ вычисляется точно. Примером такой задачи является умножение двух полиномов с целыми коэффициентами. В упражнении 30.2-6 предлагался подход, в котором используются модули длиной $\Omega(n)$ битов для вычисления ДПФ по n точкам. В данной задаче предлагается иной подход, в котором используются модули более подходящей длины $O(\lg n)$; для его понимания следует ознакомиться с материалами главы 31. Предполагается, что n является степенью 2.

а) Пусть требуется найти наименьшее значение k, такое что $p = kn + 1$ является простым числом. Предложите простое эвристическое доказательство того, что k примерно равно $\lg n$. (Значение k может быть больше или меньше, но в среднем нам придется рассмотреть $O(\lg n)$ возможных значений k.) Как ожидаемая длина p соотносится с длиной n?

Предположим, что g является генератором $\mathbf{Z}_p^*$, и пусть $w = g^k \bmod p$.

б) Докажите, что ДПФ и обратное ДПФ являются вполне определенными обратными операциями по модулю p, если в качестве w используется главное значение корня n-й степени из единицы.

в) Докажите, что БПФ и обратное ему преобразование по модулю p могут выполняться за время $O(n \lg n)$, если при этом операции со словами длиной $O(\lg n)$ выполняются за единичное время. (Значения p и w считаются заданными.)

г) Вычислите ДПФ по модулю $p = 17$ для вектора $(0, 5, 3, 7, 7, 2, 1, 6)$. Учтите, что генератором $\mathbf{Z}_{17}^*$ является $g = 3$.

Заключительные замечания

Подробное рассмотрение быстрого преобразования Фурье содержится в книге Ван Лона (VanLoan) [303]. В работах Пресса (Press), Фланнери (Flannery), Тукольски (Teukolsky) и Веттерлинга (Vetterling) [248, 249] предлагается хорошее описание быстрого преобразования Фурье и его приложений. Прекрасное введение в обработку сигналов, область широкого применения БПФ, предлагается в работах Оппенгейма (Oppenheim) и Шафера (Schafer) [232], а также Оппенгейма и Уиллски (Willsky) [233]. В книге Оппенгейма и Шафера также описаны действия в случае, когда n не является целой степенью 2.

Анализ Фурье не ограничивается одномерными данными. Он широко используется в обработке изображений для анализа данных в двух и более измерениях. Многомерные преобразования Фурье и их применение в обработке изображений обсуждаются в книгах Гонзалеса (Gonzalez) и Вудса (Woods) [127] и Пратта (Pratt) [246], а в книгах Толимьери (Tolimieri), Эн (An) и Лу (Lu) [300] и Ван Лона (Van Loan) [303] обсуждаются математические аспекты многомерных быстрых преобразований Фурье.

Изобретение БПФ в середине 1960-х часто связывают с работой Кули (Coolet) и Таки (Tukey) [68]. На самом деле БПФ неоднократно изобреталось до этого, но важность его в полной мере не осознавалась до появления современных компьютеров. Хотя Пресс, Фланнери, Тукольски и Веттерлинг приписывают открытие данного метода Рунге (Runge) и Кёнигу (König) в 1924 году, Хейдеман (Heideman), Джонсон (Johnson) и Баррус (Burrus) [141] в своей статье утверждают, что БПФ открыл еще Гаусс (Gauss) в 1805 г.

ГЛАВА 31

Теоретико-числовые алгоритмы

Раньше теория чисел рассматривалась как элегантная, но почти бесполезная область чистой математики. В наши дни теоретико-числовые алгоритмы нашли широкое применение. В определенной степени это произошло благодаря изобретению криптографических схем, основанных на больших простых числах. Применимость этих схем базируется на том, что имеется возможность легко находить большие простые числа, а их безопасность — на отсутствии простой возможности разложения на множители произведения больших простых чисел. В этой главе изложены некоторые вопросы теории чисел и связанные с ними алгоритмы, на которых основаны важные приложения.

В разделе 31.1 представлено введение в такие основные концепции теории чисел, как делимость, равенство по модулю и однозначность разложения на множители. В разделе 31.2 исследуется один из самых старых алгоритмов: алгоритм Евклида, предназначенный для вычисления наибольшего общего делителя двух целых чисел. В разделе 31.3 представлен обзор концепций модульной арифметики. Далее, в разделе 31.4 изучается множество чисел, которые являются кратными заданного числа a по модулю n, и показано, как с помощью алгоритма Евклида найти все решения уравнения $ax \equiv b \pmod{n}$. В разделе 31.5 представлена китайская теорема об остатках. В разделе 31.6 рассматриваются степени заданного числа a по модулю n, а также представлен алгоритм повторного возведения в квадрат (repeated-squaring algorithm) для эффективного вычисления величины $a^b \bmod n$ для заданных a, b и n. Эта операция является стержневой при эффективном тестировании простых чисел и для многих современных криптографических схем. Далее, в разделе 31.7 описывается криптографическая система с открытым ключом RSA. В разделе 31.8 исследуется рандомизированный тест простоты

чисел, с помощью которого можно организовать эффективный поиск больших простых чисел, что является важной задачей при создании ключей для криптографической системы RSA. Наконец, в разделе 31.9 представлен обзор простого, но эффективного эвристического подхода к задаче о разложении небольших целых чисел на множители. Интересно, что задача факторизации — одна из тех, которой, возможно, лучше было бы оставаться трудноразрешимой, поскольку от того, до какой степени трудно раскладывать на множители большие целые числа, зависит надежность RSA-кодирования.

Размер входных наборов данных и стоимость арифметических вычислений

Поскольку нам предстоит работать с большими целыми числами, следует уточнить, что́ будет подразумеваться под размером входных данных и под стоимостью элементарных арифметических операций.

В этой главе "большой ввод" обычно обозначает ввод, содержащий "большие целые числа", а не ввод, содержащий "большое количество целых чисел" (как это было в задаче сортировки). Таким образом, размер входных данных будет определяться *количеством битов*, необходимых для представления этих данных, а не просто количеством чисел в них. Время работы алгоритма, на вход которого подаются целые числа $a_1, a_2, \ldots, a_k$, является ***полиномиальным*** (polynomial), если он выполняется за время, которое выражается полиномиальной функцией от величин $\lg(a_1), \lg(a_2), \ldots, \lg(a_k)$, т.е. если это время является полиномиальным по длине входных величин в бинарном представлении.

В большей части настоящей книги было удобно считать элементарные арифметические операции (умножение, деление или вычисление остатка) примитивными, т.е. такими, которые выполняются за единичный промежуток времени. При таком предположении в качестве основы для разумной оценки фактического времени работы алгоритма на компьютере может выступать количество перечисленных арифметических операций, выполняющихся в алгоритме. Однако если входные числа достаточно большие, то для выполнения элементарных операций может потребоваться значительное время, и удобнее измерять сложность теоретико-числового алгоритма через количество ***битовых операций*** (bit operation). В этой модели для перемножения двух β-битовых целых чисел обычным методом необходимо $\Theta(\beta^2)$ битовых операций. Аналогично, операцию деления β-битового целого числа на более короткое целое число или операцию вычисления остатка от деления β-битового целого числа на более короткое целое число, с помощью простого алгоритма можно выполнить в течение времени $\Theta(\beta^2)$ (упражнение 31.1-11). Известны и более быстрые методы. Например, время перемножения двух β-битовых целых чисел методом разбиения равно $\Theta(\beta^{\lg 3})$, а время работы самого производительного из известных методов равно $\Theta(\beta \lg\beta \lg\lg\beta)$. Однако на практике

зачастую наилучшим оказывается алгоритм со временем работы $\Theta\left(\beta^2\right)$, и наш анализ будет основываться именно на этой оценке.

В этой главе алгоритмы обычно анализируются и в терминах количества арифметических операций, и в терминах количества битовых операций, требуемых для их выполнения.

31.1 Элементарные обозначения, принятые в теории чисел

В этом разделе дается краткий обзор обозначений, которые используются в элементарной теории чисел, изучающей свойства множества целых чисел $\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ и множества натуральных[1] чисел $\mathbf{N} = \{0, 1, 2, \ldots\}$.

Делимость и делители

Основным в теории чисел является обозначение того факта, что одно целое число делится на другое. Обозначение $d \mid a$ (читается "d ***делит*** a", или "a ***делится на*** d") подразумевает, что для некоторого целого числа k выполняется равенство $a = kd$. Число 0 делится на все целые числа. Если $a > 0$ и $d \mid a$, то $|d| \leqslant |a|$. Если $d \mid a$, то говорят также, что a — ***кратное*** (multiple) d. Если a не делится на d, то пишут $d \nmid a$.

Если $d \mid a$ и $d \geqslant 0$, то говорят, что d — ***делитель*** (divisor) числа a. Заметим, что $d \mid a$ тогда и только тогда, когда $-d \mid a$, поэтому без потери общности делители можно определить как неотрицательные числа, подразумевая, что число, противоположное по знаку любому делителю числа a, также является делителем числа a. Делитель числа a, отличного от нуля, лежит в пределах от 1 до $|a|$. Например, делителями числа 24 являются числа 1, 2, 3, 4, 6, 8, 12 и 24.

Каждое целое число a делится на ***тривиальные делители*** (trivial divisors) 1 и a. Нетривиальные делители числа a также называются ***множителями*** (factors) a. Например, множители числа 20 — 2, 4, 5 и 10.

Простые и составные числа

Целое число $a > 1$, единственными делителями которого являются тривиальные делители 1 и a, называют ***простым*** (prime) числом. Простые числа обладают многими особыми свойствами и играют важную роль в теории чисел. Ниже приведено 20 первых простых чисел в порядке возрастания:

[1] Определение натуральных чисел в американской литературе отличается от определения натуральных чисел в отечественной математике, где $\mathbf{N} = \{1, 2, \ldots\}$. В данной главе мы будем использовать определение натуральных чисел из оригинального издания книги. — *Прим. ред.*

$$2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19,\ 23,\ 29,\ 31,\ 37,\ 41,\ 43,\ 47,\ 53,\ 59,\ 61,\ 67,\ 71\ .$$

В упражнении 31.1-1 предлагается доказать, что простых чисел бесконечно много. Целое число $a > 1$, которое не является простым, называется ***составным*** (composite). Например, число 39 — составное, поскольку $3 \mid 39$. Целое число 1 называют ***единицей*** (unit), и оно не является ни простым, ни составным. Аналогично, целое число 0 и все отрицательные целые числа тоже не являются ни простыми, ни составными.

Теорема о делении, остатки и равенство по модулю

Относительно заданного числа n, все целые числа можно разбить на те, которые являются кратными числу n, и те, которые не являются кратными числу n. Бóльшая часть теории чисел основана на усовершенствовании этого деления, в котором последняя категория чисел классифицируется в соответствии с тем, чему равен остаток их деления на n. В основе этого усовершенствования лежит сформулированная ниже теорема (ее доказательство здесь не приводится, но его можно найти, например, в учебнике Найвена (Niven) и Цукермана (Zuckerman) [231]).

Теорема 31.1 (Теорема о делении). Для любого целого a и любого положительного целого n существует единственная пара целых чисел q и r, таких что $0 \leqslant r < n$ и $a = qn + r$. ■

Величина $q = \lfloor a/n \rfloor$ называется ***частным*** (quotient) деления. Величина $r = a \bmod n$ называется ***остатком*** (reminder или residue) деления. Таким образом, $n \mid a$ тогда и только тогда, когда $a \bmod n = 0$.

В соответствии с тем, чему равны остатки чисел по модулю n, их можно разбить на n классов эквивалентности. ***Класс эквивалентности по модулю*** n (equivalence class modulo n), в котором содержится целое число a, имеет вид

$$[a]_n = \{a + kn : k \in \mathbf{Z}\}\ .$$

Например, $[3]_7 = \{\ldots, -11, -4, 3, 10, 17, \ldots\}$; другие обозначения этого множества — $[-4]_7$ и $[10]_7$. Используя обозначения из раздела 3.2, можно сказать, что запись $a \in [b]_n$ — это то же самое, что и запись $a \equiv b \,(\mathrm{mod} n)$. Множество всех таких классов эквивалентности имеет вид

$$\mathbf{Z}_n = \{[a]_n : 0 \leqslant a \leqslant n - 1\}\ . \tag{31.1}$$

Часто встречается определение

$$\mathbf{Z}_n = \{0, 1, \ldots, n - 1\}\ , \tag{31.2}$$

которое эквивалентно уравнению (31.1) с учетом того, что 0 представляет класс $[0]_n$, 1 — класс $[1]_n$ и т.д. Каждый класс в этом представлении определяется своим наименьшим неотрицательным элементом. Однако при этом следует иметь в виду именно соответствующие классы эквивалентности. Например, -1 в качестве члена класса $\mathbf{Z}_n$ обозначает $[n-1]_n$, поскольку $-1 \equiv n-1 \pmod{n}$.

Общие делители и наибольшие общие делители

Если d — делитель числа a, а также делитель числа b, то d — общий делитель чисел a и b. Например, делители числа 30 — 1, 2, 3, 5, 6, 10, 15 и 30, а общие делители чисел 24 и 30 — 1, 2, 3 и 6. Заметим, что 1 — общий делитель двух любых целых чисел.

Важное свойство общих делителей заключается в том, что

$$\text{из } d \mid a \text{ и } d \mid b \text{ следует } d \mid (a+b) \text{ и } d \mid (a-b)\ . \tag{31.3}$$

В более общем виде можно написать, что для всех целых x и y

$$\text{из } d \mid a \text{ и } d \mid b \text{ следует } d \mid (ax+by)\ . \tag{31.4}$$

Кроме того, если $a \mid b$, то либо $|a| \leqslant |b|$, либо $b = 0$, так что

$$\text{из } a \mid b \text{ и } b \mid a \text{ следует } a = \pm b\ . \tag{31.5}$$

Наибольший общий делитель (greatest common divisor) двух целых чисел a и b, отличных от нуля, — это самый большой из общих делителей чисел a и b; он обозначается как $\gcd(a,b)$[2]. Например, $\gcd(24,30) = 6$, $\gcd(5,7) = 1$ и $\gcd(0,9) = 9$. Если числа a и b отличны от нуля, то значение $\gcd(a,b)$ находится в интервале от 1 до $\min(|a|,|b|)$. Величину $\gcd(0,0)$ считаем равной нулю; это необходимо для универсальной применимости стандартных свойств функции $\gcd$ (пример одного из них выражается уравнением (31.9)).

Ниже перечислены элементарные свойства функции $\gcd$:

$$\gcd(a,b) = \gcd(b,a)\ , \tag{31.6}$$
$$\gcd(a,b) = \gcd(-a,b)\ , \tag{31.7}$$
$$\gcd(a,b) = \gcd(|a|,|b|)\ , \tag{31.8}$$
$$\gcd(a,0) = |a|\ , \tag{31.9}$$
$$\gcd(a,ka) = |a| \quad \text{для всех } k \in \mathbf{Z}\ . \tag{31.10}$$

Сформулированная ниже теорема дает полезную альтернативную характеристику функции $\gcd(a,b)$.

[2] В отечественной литературе применяется также обозначение НОД(a,b). — *Прим. пер.*

Теорема 31.2. Если a и b — произвольные целые числа, отличные от нуля, то величина $\gcd(a, b)$ равна наименьшему положительному элементу множества $\{ax + by : x, y \in \mathbf{Z}\}$ линейных комбинаций чисел a и b.

Доказательство. Обозначим через s наименьшую положительную из таких линейных комбинаций чисел a и b, и пусть $s = ax + by$ при некоторых $x, y \in \mathbf{Z}$. Пусть также $q = \lfloor a/s \rfloor$. Тогда из равенства (3.8) следует

$$a \bmod s = a - qs = a - q(ax + by) = a(1 - qx) + b(-qy) \ ,$$

поэтому величина $a \bmod s$ также является линейной комбинацией чисел a и b. Но поскольку $0 \leqslant a \bmod s < s$, выполняется соотношение $a \bmod s = 0$, так как s — наименьшая положительная из таких линейных комбинаций. Поэтому $s \mid a$; аналогично можно доказать, что $s \mid b$. Таким образом, s — общий делитель чисел a и b, поэтому справедливо неравенство $\gcd(a, b) \geqslant s$. Из уравнения (31.4) следует, что $\gcd(a, b) \mid s$, поскольку величина $\gcd(a, b)$ делит и a, и b, а s — линейная комбинация чисел a и b. Но из того, что $\gcd(a, b) \mid s$, и $s > 0$ следует соотношение $\gcd(a, b) \leqslant s$. Совместное применение неравенств $\gcd(a, b) \geqslant s$ и $\gcd(a, b) \leqslant s$ доказывает справедливость равенства $\gcd(a, b) = s$; следовательно, можно сделать вывод, что s — наибольший общий делитель чисел a и b. ■

Следствие 31.3. Для любых целых чисел a и b из соотношений $d \mid a$ и $d \mid b$ следует $d \mid \gcd(a, b)$.

Доказательство. Это следствие является результатом уравнения (31.4), поскольку, согласно теореме 31.2, величина $\gcd(a, b)$ является линейной комбинацией чисел a и b. ■

Следствие 31.4. Для любых целых чисел a и b и произвольного неотрицательного целого числа n справедливо соотношение

$$\gcd(an, bn) = n \gcd(a, b) \ .$$

Доказательство. Если $n = 0$, то следствие доказывается тривиально. Если же $n > 0$, то $\gcd(an, bn)$ — наименьший положительный элемент множества $\{anx + bny\}$, в n раз превышающий наименьший положительный элемент множества $\{ax + by\}$. ■

Следствие 31.5. Для всех положительных целых чисел n, a и b из условий $n \mid ab$ и $\gcd(a, n) = 1$ следует соотношение $n \mid b$.

Доказательство. Доказательство этого следствия читателю предлагается выполнить самостоятельно в упражнении 31.1-4. ■

Взаимно простые целые числа

Два числа a и b называются ***взаимно простыми*** (relatively prime), если единственный их общий делитель равен 1, т.е. если $\gcd(a, b) = 1$. Например, числа 8 и 15 взаимно простые, поскольку делители числа 8 равны 1, 2, 4 и 8, а делители числа 15 — 1, 3, 5 и 15. В сформулированной ниже теореме утверждается, что если два целых числа взаимно простые с целым числом p, то их произведение тоже является взаимно простым с числом p.

Теорема 31.6. Для любых целых чисел a, b и p из соотношений $\gcd(a, p) = 1$ и $\gcd(b, p) = 1$ следует соотношение $\gcd(ab, p) = 1$.

Доказательство. Из теоремы 31.2 следует, что существуют такие целые числа x, y, x' и y', для которых

$$ax + py = 1,$$
$$bx' + py' = 1.$$

Умножив эти уравнения друг на друга и перегруппировав слагаемые, получим:

$$ab\left(xx'\right) + p\left(ybx' + y'ax + pyy'\right) = 1.$$

Поскольку положительная линейная комбинация чисел ab и p равна 1, применение теоремы 31.2 завершает доказательство. ■

Говорят, что целые числа $n_1, n_2, \ldots, n_k$ ***попарно взаимно простые*** (pairwise relatively prime), если при $i \neq j$ выполняется соотношение $\gcd(n_i, n_j) = 1$.

Единственность разложения на множители

Элементарный, но важный факт, касающийся делимости на простые числа, сформулирован в приведенной ниже теореме.

Теорема 31.7. Для любого простого числа p и всех целых чисел a и b из условия $p \mid ab$ следуют либо соотношение $p \mid a$, либо соотношение $p \mid b$, либо они оба.

Доказательство. Проведем доказательство методом “от противного”. Предположим, что $p \mid ab$, но $p \nmid a$ и $p \nmid b$. Таким образом, $\gcd(a, p) = 1$ и $\gcd(b, p) = 1$, поскольку единственными делителями числа p являются 1 и p, и, согласно предположению, на p не делится ни a, ни b. Тогда из теоремы 31.6 следует, что $\gcd(ab, p) = 1$, а это противоречит условию, что $p \mid ab$, поскольку из него следует, что $\gcd(ab, p) = p$. Это противоречие и служит доказательством теоремы. ■

Из теоремы 31.8 следует, что любое целое число единственным образом представляется в виде произведения простых чисел.

Теорема 31.8 (Единственность разложения на множители). Целое составное число a можно единственным образом представить как произведение вида $a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, где p_i — простые числа, $p_1 < p_2 < \cdots < p_r$, а e_i — целые положительные числа.

Доказательство. Доказательство этой теоремы в упражнении 31.1-10 читателю предлагается выполнить самостоятельно. ■

Например, число 6000 можно единственным образом разложить на множители $2^4 \cdot 3 \cdot 5^3$.

Упражнения

31.1-1. Докажите, что простых чисел бесконечно много. (*Указание*: покажите, что ни одно из простых чисел $p_1, p_2, \ldots, p_k$ не является делителем числа $(p_1 p_2 \cdots p_k) + 1$.)

31.1-2. Докажите, что если $a \mid b$ и $b \mid c$, то $a \mid c$.

31.1-3. Докажите, что если p — простое число и $0 < k < p$, то $\gcd(k, p) = 1$.

31.1-4. Докажите следствие 31.5.

31.1-5. Докажите, что если p — простое число и $0 < k < p$, то $p \mid \binom{p}{k}$. Сделайте отсюда вывод, что для всех целых чисел a и b и простых p выполняется соотношение $(a + b)^p \equiv a^p + b^p \pmod{p}$.

31.1-6. Докажите, что если a и b — положительные целые числа, такие что $a \mid b$, то для всех x выполняется соотношение

$$(x \bmod b) \bmod a = x \bmod a.$$

Докажите, что при тех же предположениях для всех целых чисел x и y из соотношения $x \equiv y \pmod{b}$ следует соотношение $x \equiv y \pmod{a}$.

31.1-7. Говорят, что для всех целых $k > 0$ целое число n — ***k-я степень*** (kth power), если существует целое число a, такое что $a^k = n$. Число $n > 1$ — ***нетривиальная степень*** (nontrivial power), если оно является k-й степенью для некоторого целого $k > 1$. Покажите, как определить, является ли заданное β-битовое целое число n нетривиальной степенью, за время, которое выражается полиномиальной функцией от β.

31.1-8. Докажите уравнения (31.6)–(31.10).

31.1-9. Покажите, что операция gcd обладает свойством ассоциативности, т.е. докажите, что для всех целых a, b и c справедливо соотношение

$$\gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c).$$

⋆ 31.1-10. Докажите теорему 31.8.

31.1-11. Разработайте эффективный алгоритм для операций деления β-битового целого числа на более короткое целое число и вычисления остатка от деления β-битового целого числа на более короткое целое число. Время работы алгоритма должно быть равно $O\left(\beta^2\right)$.

31.1-12. Разработайте эффективный алгоритм преобразования заданного β-битового (двоичного) целого числа в десятичное. Докажите, что если умножение или деление целых чисел, длина которых не превышает β, выполняется в течение времени $M(\beta)$, то преобразование из двоичной в десятичную систему счисления можно выполнить за время $\Theta(M(\beta)\lg\beta)$. (*Указание*: воспользуйтесь стратегией "разделяй и властвуй", при котором верхняя и нижняя половины результата получаются в результате отдельно выполненных рекурсивных процедур.)

31.2 Наибольший общий делитель

В этом разделе описан алгоритм Евклида, предназначенный для эффективного вычисления наибольшего общего делителя двух целых чисел. Анализ времени работы этого алгоритма выявляет удивительную связь с числами Фибоначчи, которые являются наихудшими входными данными для алгоритма Евклида.

Ограничимся в этом разделе неотрицательными целыми числами. Это ограничение обосновывается уравнением (31.8), согласно которому $\gcd(a,b) = \gcd(|a|,|b|)$.

В принципе, величину $\gcd(a,b)$ для положительных целых чисел a и b можно вычислить, пользуясь разложением чисел a и b на простые множители. В самом деле, если разложения чисел a и b имеют вид

$$a = p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r}, \tag{31.11}$$

$$b = p_1^{f_1} p_2^{f_2} \ldots p_r^{f_r}, \tag{31.12}$$

где нулевые показатели экспоненты используются для того, чтобы в обоих разложениях были представлены одинаковые множества простых чисел $p_1, p_2, \ldots, p_r$, то можно показать, что

$$\gcd(a,b) = p_1^{\min(e_1,f_1)} p_2^{\min(e_2,f_2)} \ldots p_r^{\min(e_r,f_r)}. \tag{31.13}$$

В упражнении 31.2-1 предлагается показать, что это действительно так. Однако в разделе 31.9 будет показано, что время работы самых эффективных на сегодняшний день алгоритмов разложения на множители не выражается полиномиальной функцией; они работают менее производительно.

Алгоритм Евклида, предназначенный для вычисления наибольшего общего делителя, основан на сформулированной ниже теореме.

Теорема 31.9 (Рекурсивная теорема о НОД). Для любого неотрицательного целого числа a и любого положительного целого числа b справедливо соотношение $\gcd(a, b) = \gcd(b, a \bmod b)$.

Доказательство. Покажем, что величины $\gcd(a, b)$ и $\gcd(b, a \bmod b)$ делятся друг на друга, поэтому они должны быть равны друг другу (поскольку оба они неотрицательны) согласно уравнению (31.5).

Сначала покажем, что $\gcd(a, b) \mid \gcd(b, a \bmod b)$. Если $d = \gcd(a, b)$, то $d \mid a$ и $d \mid b$. Согласно уравнению (3.8), $(a \bmod b) = a - qb$, где $q = \lfloor a/b \rfloor$. Поскольку величина $(a \bmod b)$ представляет собой линейную комбинацию чисел a и b, то из уравнения (31.4) следует, что $d \mid (a \bmod b)$. Таким образом, поскольку $d \mid b$ и $d \mid (a \bmod b)$, согласно следствию 31.3, $d \mid \gcd(b, a \bmod b)$ или, что то же самое,

$$\gcd(a, b) \mid \gcd(b, a \bmod b). \tag{31.14}$$

Соотношение $\gcd(b, a \bmod b) \mid \gcd(a, b)$ доказывается почти так же. Если ввести обозначение $d = \gcd(b, a \bmod b)$, то $d \mid b$ и $d \mid (a \bmod b)$. Поскольку $a = qb + (a \bmod b)$, где $q = \lfloor a/b \rfloor$, a представляет собой линейную комбинацию величин b и $(a \bmod b)$. Согласно уравнению (31.4), можно сделать вывод, что $d \mid a$. Поскольку $d \mid b$ и $d \mid a$, то, согласно следствию 31.3, справедливо соотношение $d \mid \gcd(a, b)$ или эквивалентное ему

$$\gcd(b, a \bmod b) \mid \gcd(a, b). \tag{31.15}$$

Использование уравнения (31.5) в комбинации с уравнениями (31.14) и (31.15) завершает доказательство. ■

Алгоритм Евклида

В книге Евклида "Начала" (около 300 г. до н.э.) описывается приведенный ниже алгоритм вычисления gcd (хотя на самом деле он может иметь более раннее происхождение). Алгоритм Евклида выражается в виде рекурсивной программы и непосредственно основан на теореме 31.9. В качестве входных данных в нем выступают произвольные неотрицательные целые числа a и b.

```
EUCLID(a, b)
1  if b = 0
2      then return a
3      else return EUCLID(b, a mod b)
```

В качестве примера работы процедуры EUCLID рассмотрим вычисление величины $\gcd(30, 21)$:

$$\begin{aligned}\text{EUCLID}(30, 21) &= \text{EUCLID}(21, 9) \\ &= \text{EUCLID}(9, 3) \\ &= \text{EUCLID}(3, 0) \\ &= 3 \ .\end{aligned}$$

В приведенных выше выкладках мы видим три рекурсивных вызова процедуры EUCLID.

Корректность процедуры EUCLID следует из теоремы 31.9, а также из того факта, что если алгоритм во второй строке возвращает значение a, то $b = 0$, а из уравнения (31.9) следует, что $\gcd(a, b) = \gcd(a, 0) = a$. Работа этого рекурсивного алгоритма не может продолжаться до бесконечности, поскольку второй аргумент процедуры всегда строго убывает при каждом рекурсивном вызове, и он всегда неотрицательный. Таким образом, алгоритм EUCLID всегда завершается и дает правильный ответ.

Время работы алгоритма Евклида

Проанализируем наихудшее время работы алгоритма EUCLID как функцию размера входных данных a и b. Без потери общности предположим, что $a > b \geqslant 0$. Это предположение легко обосновать, если заметить, что если $b > a \geqslant 0$, то в процедуре $\text{EUCLID}(a, b)$ сразу же производится рекурсивный вызов процедуры $\text{EUCLID}(b, a)$. Другими словами, если первый аргумент меньше второго, то алгоритм EUCLID затрачивает один дополнительный вызов на перестановку аргументов и продолжает свою работу. Аналогично, если $b = a > 0$, то процедура завершается после одного рекурсивного вызова, поскольку $a \bmod b = 0$.

Полное время работы алгоритма EUCLID пропорционально количеству выполняемых ею рекурсивных вызовов. В нашем анализе используются числа Фибоначчи F_k, определяемые рекуррентным соотношением (3.21).

Лемма 31.10. Если $a > b \geqslant 1$ и в результате вызова процедуры $\text{EUCLID}(a, b)$ выполняется $k \geqslant 1$ рекурсивных вызовов, то $a \geqslant F_{k+2}$ и $b \geqslant F_{k+1}$.

Доказательство. Докажем лемму путем индукции по k. В качестве базиса индукции примем $k = 1$. Тогда $b \geqslant 1 = F_2$ и, поскольку $a > b$, автоматически выполняется соотношение $a \geqslant 2 = F_3$. Поскольку $b > (a \bmod b)$, при каждом рекурсивном вызове первый аргумент строго больше второго; таким образом, предположение $a > b$ выполняется при каждом рекурсивном вызове процедуры.

Примем в качестве гипотезы индукции, что лемма справедлива, если произведено $k - 1$ рекурсивных вызовов; затем докажем, что она выполняется, если

произведено k рекурсивных вызовов. Поскольку $k > 0$, то и $b > 0$, и в процедуре EUCLID(a, b) рекурсивно вызывается процедура EUCLID$(b, a \bmod b)$, в которой, в свою очередь, выполняется $k - 1$ рекурсивных вызовов. Далее, согласно гипотезе индукции, выполняются неравенства $b \geqslant F_{k+1}$ (что доказывает часть леммы) и $(a \bmod b) \geqslant F_k$. Мы имеем

$$b + (a \bmod b) = b + (a - \lfloor a/b \rfloor\, b) \leqslant a,$$

поскольку из $a > b > 0$ следует $\lfloor a/b \rfloor \geqslant 1$. Таким образом,

$$a \geqslant b + (a \bmod b) \geqslant F_{k+1} + F_k = F_{k+2}.$$

■

Из этой леммы непосредственно следует сформулированная ниже теорема.

Теорема 31.11 (Теорема Ламе (Lame)). Если для произвольного целого числа $k \geqslant 1$ выполняются условия $a > b \geqslant 1$ и $b < F_{k+1}$, то в вызове процедуры EUCLID(a, b) производится менее k рекурсивных вызовов. ■

Можно показать, что верхняя граница теоремы 31.11 — лучшая из возможных. Последовательные числа Фибоначчи — наихудшие входные данные для процедуры EUCLID. Поскольку в процедуре EUCLID(F_3, F_2) производится ровно один рекурсивный вызов и поскольку при $k > 2$ выполняется соотношение $F_{k+1} \bmod F_k = F_{k-1}$, мы имеем

$$\gcd(F_{k+1}, F_k) = \gcd(F_k, (F_{k+1} \bmod F_k)) = \gcd(F_k, F_{k-1}).$$

Таким образом, в процедуре EUCLID(F_{k+1}, F_k) рекурсия осуществляется *в точности* $k - 1$ раз, что совпадает с верхней границей теоремы 31.11.

Поскольку число F_k приблизительно равно $\phi^k/\sqrt{5}$, где ϕ — золотое сечение $\left(1 + \sqrt{5}\right)/2$, определенное уравнением (3.22), количество рекурсивных вызовов в процедуре EUCLID равно $O(\lg b)$. (Более точная оценка предлагается в упражнении 31.2-5.) Отсюда следует, что если с помощью алгоритма EUCLID обрабатывается два β-битовых числа, то в нем производится $O(\beta)$ арифметических операций и $O\left(\beta^3\right)$ битовых операций (в предположении, что при умножении и делении β-битовых чисел выполняется $O\left(\beta^2\right)$ битовых операций. Справедливость этой оценки предлагается показать в задаче 31-2).

Развернутая форма алгоритма Евклида

Теперь перепишем алгоритм Евклида так, чтобы с его помощью можно было извлекать дополнительную полезную информацию, в частности, чтобы вычислять целые коэффициенты x и y, для которых справедливо равенство

$$d = \gcd(a, b) = ax + by. \tag{31.16}$$

Таблица 31.1. Пример работы алгоритма EXTENDED_EUCLID с входными числами 99 и 78

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	-11	14
78	21	3	3	3	-11
21	15	1	3	-2	3
15	6	2	3	1	-2
6	3	2	3	0	1
3	0	—	3	1	0

Заметим, что числа x и y могут быть равными нулю или отрицательными. Эти коэффициенты окажутся полезными позже при вычислении модульных мультипликативных обратных значений. В качестве ввода процедуры EXTENDED_EUCLID выступает пара неотрицательных целых чисел, а на выходе эта процедура возвращает тройку чисел (d, x, y), удовлетворяющих уравнению (31.16).

```
EXTENDED_EUCLID(a, b)
1  if b = 0
2     then return (a, 1, 0)
3  (d', x', y') ← EXTENDED_EUCLID(b, a mod b)
4  (d, x, y) ← (d', y', x' − ⌊a/b⌋ y')
5  return (d, x, y)
```

Работа алгоритма EXTENDED_EUCLID по вычислению величины $\gcd(99, 78)$ проиллюстрирована в табл. 31.1. В каждой строке показан один уровень рекурсии: входные величины a и b, вычисленная величина $\lfloor a/b \rfloor$, а также возвращаемые величины d, x и y. Возвращаемая тройка значений (d, x, y) становится тройкой, которая используется в ходе вычислений на следующем, более высоком уровне рекурсии. В результате вызова процедуры EXTENDED_EUCLID$(99, 78)$ возвращаются величины $(3, -11, 14)$, так что $\gcd(99, 78) = 3 = 99 \cdot (-11) + 78 \cdot 14$.

Процедура EXTENDED_EUCLID представляет собой разновидность процедуры EUCLID. Строка 1 в ней эквивалентна проверке равенства значение b нулю в первой строке процедуры EUCLID. Если $b = 0$, то процедура EXTENDED_EUCLID в строке 2 возвращает не только значение $d = a$, но и коэффициенты $x = 1$ и $y = 0$, так что $a = ax + by$. Если $b \neq 0$, то процедура EXTENDED_EUCLID сначала вычисляет набор величин (d', x', y'), таких что $d' = \gcd(b, a \bmod b)$ и

$$d' = bx' + (a \bmod b)\, y'. \tag{31.17}$$

Как и в процедуре EUCLID, в этом случае мы имеем $d = \gcd(a, b) = d' = \gcd(b, a \bmod b)$. Чтобы получить значения x и y, для которых выполняется

равенство $d = ax + by$, сначала перепишем уравнение (31.17) с использованием равенств $d = d'$ и (3.8):

$$d = bx' + (a - \lfloor a/b \rfloor\, b)\, y' = ay' + b\left(x' - \lfloor a/b \rfloor\, y'\right).$$

Таким образом, при выборе величин $x = y'$ и $y = x' - \lfloor a/b \rfloor\, y'$ удовлетворяется уравнение $d = ax + by$, что доказывает корректность процедуры EXTENDED_EUCLID.

Поскольку количество рекурсивных вызовов в процедуре EUCLID равно количеству рекурсивных вызовов в процедуре EXTENDED_EUCLID, время работы процедуры EUCLID с точностью до постоянного множителя равно времени работы процедуры EXTENDED_EUCLID. Другими словами, при $a > b > 0$ количество рекурсивных вызовов равно $O(\lg b)$.

Упражнения

31.2-1. Докажите, что из уравнений (31.11) и (31.12) следует уравнение (31.13).

31.2-2. Вычислите величины (d, x, y), которые возвращаются при вызове процедуры EXTENDED_EUCLID$(899, 493)$.

31.2-3. Докажите, что для всех целых чисел a, k и n выполняется соотношение $\gcd(a, n) = \gcd(a + kn, n)$.

31.2-4. Перепишите алгоритм EUCLID в итеративном виде, с использованием памяти фиксированного объема (т.е. в ней должно храниться не более некоторого фиксированного количества целочисленных значений).

31.2-5. Покажите, что если $a > b \geqslant 0$, то при вызове процедуры EUCLID(a, b) выполняется не более $1 + \log_\phi b$ рекурсивных вызовов. Улучшите эту оценку до $1 + \log_\phi (b/\gcd(a, b))$.

31.2-6. Какие значения возвращает процедура EXTENDED_EUCLID(F_{k+1}, F_k)? Докажите верность вашего ответа.

31.2-7. Определим функцию gcd для более чем двух аргументов с помощью рекурсивного уравнения $\gcd(a_0, a_1, \ldots, a_n) = \gcd(a_0, \gcd(a_1, a_2, \ldots, a_n))$. Покажите, что значение этой функции не зависит от порядка ее аргументов. Покажите также, как найти целые числа $x_0, x_1, \ldots, x_n$, такие что $\gcd(a_0, a_1, \ldots, a_n) = a_0x_0 + a_1x_1 + \cdots a_nx_n$. Покажите, что количество операций деления, которые производятся в алгоритме, равно $O(n + \lg(\max\{a_0, a_1, \ldots, a_n\}))$.

31.2-8. Определим ***наименьшее общее кратное*** (least common multiple) n целых чисел $a_1, a_2, \ldots, a_n$, обозначаемое $\operatorname{lcm}(a_1, a_2, \ldots, a_n)$, как наименьшее неотрицательное целое число, кратное каждому из аргументов a_i. Покажите, как эффективно вычислить величину $lcm(a_1, a_2, \ldots, a_n)$, используя в качестве подпрограммы (двухаргументную) операцию $\gcd$.

31.2-9. Докажите, что числа n_1, n_2, n_3 и n_4 попарно взаимно просты тогда и только тогда, когда

$$\gcd(n_1 n_2, n_3 n_4) = \gcd(n_1 n_3, n_2 n_4) = 1.$$

Покажите, что справедливо более общее утверждение, согласно которому числа $n_1, n_2, \ldots n_k$ попарно взаимно просты тогда и только тогда, когда множество $\lceil \lg k \rceil$ пар чисел, образованных из n_i, взаимно простые.

31.3 Модульная арифметика

Неформально говоря, модульную арифметику можно считать обычной арифметикой, вычисления в которой производятся над целыми числами, за исключением того, что если что-то нужно вычислить по модулю числа n, то любой результат x заменяется элементом множества $\{0, 1, \ldots, n-1\}$, равным числу x по модулю n (т.е. x заменяется величиной $x \bmod n$). Описанной выше неформальной модели достаточно, если ограничиться операциями сложения, вычитания и умножения. Более формализованная модель модульной арифметики, которая будет представлена ниже, лучше описывается в терминах теории групп.

Конечные группы

Группа (group) $(S, \oplus)$ — это множество S, для элементов которого определена бинарная операция $\oplus$, обладающая перечисленными ниже свойствами.

1. **Замкнутость:** для всех элементов $a, b \in S$ имеем $a \oplus b \in S$.
2. **Существование единицы:** существует элемент $e \in S$, который называется ***единичным*** (identity) элементом группы; для этого элемента и любого элемента $a \in S$ выполняется соотношение $e \oplus a = a \oplus e = a$.
3. **Ассоциативность:** для всех $a, b, c \in S$ выполняется соотношение $(a \oplus b) \oplus \oplus c = a \oplus (b \oplus c)$.
4. **Существование обратного элемента:** для каждого элемента $a \in S$ существует единственный элемент $b \in S$ (он называется ***обратным*** (inverse) к элементу a), такой что $a \oplus b = b \oplus a = e$.

В качестве примера рассмотрим уже знакомую нам группу $(\mathbf{Z}, +)$ целых чисел с операцией сложения: в ней единичный элемент — 0, а обратный элемент к любому числу a — число $-a$. Если группа $(S, \oplus)$ обладает ***свойством коммутативности*** (commutative law) $a \oplus b = b \oplus a$ для всех $a, b \in S$, то это ***абелева группа*** (abelian group). Если группа $(S, \oplus)$ удовлетворяет условию $|S| < \infty$, т.е. количество ее элементов конечно, то она называется ***конечной*** (finite group).

Группы, образованные сложением и умножением по модулю

С помощью операций сложения и умножения по модулю n, где n — положительное целое число, можно образовать две конечные абелевы группы. Эти группы основаны на классах эквивалентности целых чисел по модулю n, определенных в разделе 31.1.

Чтобы определить группу над множеством классов $\mathbf{Z}_n$, нужно задать подходящие бинарные операции, полученные путем переопределения обычных операций сложения и умножения. Операции сложения и умножения над $\mathbf{Z}_n$ определить легко, поскольку классы эквивалентности двух целых чисел однозначно определяют класс эквивалентности их суммы или произведения. Другими словами, если $a \equiv a' \pmod{n}$ и $b \equiv b' \pmod{n}$, то

$$\begin{aligned} a + b &\equiv a' + b' \pmod{n}, \\ ab &\equiv a'b' \pmod{n}. \end{aligned}$$

Таким образом, операции сложения и умножения по модулю n, которые мы обозначим как $+_n$ и $\cdot_n$, определяются следующим образом:

$$\begin{aligned} [a]_n +_n [b]_n &= [a + b]_n, \\ [a]_n \cdot_n [b]_n &= [ab]_n. \end{aligned} \tag{31.18}$$

(Вычитание в $\mathbf{Z}_n$ можно легко определить как $[a]_n -_n [b]_n = [a - b]_n$, однако с делением, как мы сможем вскоре убедиться, дело обстоит сложнее.) Эти факты подтверждают удобную общепринятую практику использования наименьшего неотрицательного элемента каждого класса эквивалентности как представительного при вычислениях в $\mathbf{Z}_n$. Сложение, вычитание и умножение над представителями классов выполняется как обычно, но затем каждый результат x заменяется соответствующим представителем класса эквивалентности (т.е. величиной $x \bmod n$).

На основе определения операции сложения по модулю n определяется ***аддитивная группа по модулю*** n (additive group modulo n) $(\mathbf{Z}_n, +_n)$. Размер аддитивной группы по модулю n равен $|\mathbf{Z}_n| = n$. В табл. 31.2 представлены результаты выполнения операций в группе $(\mathbf{Z}_6, +_6)$.

Теорема 31.12. Система $(\mathbf{Z}_n, +_n)$ образует конечную абелеву группу.

Доказательство. Из уравнения (31.18) следует замкнутость группы $(\mathbf{Z}_n, +_n)$. Ассоциативность и коммутативность операции $+_n$ следует из ассоциативности

и коммутативности операции $+$:

$$\begin{aligned}([a]_n +_n [b]_n) +_n [c]_n &= [a+b]_n +_n [c]_n = \\ &= [(a+b)+c]_n = \\ &= [a+(b+c)]_n = \\ &= [a]_n +_n [b+c]_n = \\ &= [a]_n +_n ([b]_n +_n [c]_n)\,,\end{aligned}$$

$$\begin{aligned}[a]_n +_n [b]_n &= [a+b]_n = \\ &= [b+a]_n = \\ &= [b]_n +_n [a]_n\,.\end{aligned}$$

В роли единичного элемента $(\mathbf{Z}_n, +_n)$ выступает 0 (т.е. класс $[0]_n$). Элемент, аддитивно обратный элементу a (т.е. классу $[a]_n$), представляет собой элемент $-a$ (т.е. класс $[-a]_n$ или $[n-a]_n$), поскольку $[a]_n +_n [-a]_n = [a-a]_n = [0]_n$. ■

На основе операции умножения по модулю n определяется ***мультипликативная группа по модулю*** n (multiplicative group modulo n) $(\mathbf{Z}_n^*, \cdot_n)$. Элементы этой группы образуют множество $\mathbf{Z}_n^*$, образованное из элементов множества $\mathbf{Z}_n$, взаимно простых с n:

$$\mathbf{Z}_n^* = \{[a]_n \in \mathbf{Z}_n : \gcd(a,n) = 1\}\,.$$

Чтобы убедиться, что группа $\mathbf{Z}_n^*$ вполне определена, заметим, что для $0 \leqslant a < n$ при всех целых k выполняется соотношение $a \equiv (a+kn) \pmod{n}$. Поэтому из $\gcd(a,n) = 1$, согласно результатам упражнения 31.2-3, для всех целых k следует, что $\gcd(a+nk, n) = 1$. Поскольку $[a]_n = \{a+kn : k \in \mathbf{Z}\}$, множество $\mathbf{Z}_n^*$ вполне определено. Примером такой группы является множество

$$\mathbf{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}\,,$$

в качестве групповой операции в которой выступает операция умножения по модулю 15. (Здесь элемент $[a]_{15}$ обозначается как a; например, элемент $[7]_{15}$ обозначается как 7.) В табл. 31.3 показаны результаты выполнения операции для данной группы. Например, $8 \cdot 11 \equiv 13 \pmod{15}$. Единичным элементом в этой группе является 1.

Теорема 31.13. Система $(\mathbf{Z}_n^*, \cdot_n)$ образует конечную абелеву группу.

Доказательство. Замкнутость $(\mathbf{Z}_n^*, \cdot_n)$ следует из теоремы 31.6. Ассоциативность и коммутативность для операции $\cdot_n$ можно доказать аналогично доказательству этих свойств для операции $+_n$ в теореме 31.12. В роли единичного здесь

Таблица 31.2. Группа $(\mathbf{Z}_6, +_6)$

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

Таблица 31.3. Группа $(Z_{15}^*, \cdot_{15})$

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

выступает элемент $[1]_n$. Чтобы показать наличие обратных элементов, предположим, что a — элемент множества $\mathbf{Z}_n^*$, а набор чисел (d, x, y) — выходные данные процедуры EXTENDED_EUCLID(a, n). Тогда $d = 1$, поскольку $a \in \mathbf{Z}_n^*$ и справедливо равенство

$$ax + ny = 1,$$

или, что то же самое,

$$ax \equiv 1 \,(\mathrm{mod} n)\,.$$

Таким образом, класс $[x]_n$ — обратный классу $[a]_n$ относительно операции умножения по модулю n. Доказательство того, что обратные элементы определяются однозначно, отложим до рассмотрения следствия 31.26. ■

В качестве примера вычисления мультипликативных обратных элементов рассмотрим случай $a = 5$ и $n = 11$. Процедура EXTENDED_EUCLID(a, n) возвращает тройку чисел $(d, x, y) = (1, -2, 1)$, так что $1 = 5 \cdot (-2) + 11 \cdot 1$. Таким образом, число -2 (т.е. $9 \bmod 11$) — мультипликативное обратное по модулю 11 к числу 5.

Далее в оставшейся части этой главы, когда речь будет идти о группах $(\mathbf{Z}_n, +_n)$ и $(\mathbf{Z}_n^*, \cdot_n)$, мы будем обозначать классы эквивалентности представляющими их элементами, а операции $+_n$ и $\cdot_n$ — знаками обычных арифметических операций $+$ и $\cdot$ (последний знак может опускаться) соответственно. Кроме того, эквивалентность по модулю n можно интерпретировать как равенство в $\mathbf{Z}_n$. Например, оба выражения, приведенные ниже, обозначают одно и то же:

$$ax \equiv b \,(\mathrm{mod} n)\,,$$
$$[a]_n \cdot_n [x]_n = [b]_n\,.$$

Для дальнейшего удобства иногда группа $(S, \oplus)$ будет обозначаться просто как S, а из контекста будет понятно, какая именно операция подразумевается. Таким образом, группы $(\mathbf{Z}_n, +_n)$ и $(\mathbf{Z}_n^*, \cdot_n)$ будут обозначаться как $\mathbf{Z}_n$ и $\mathbf{Z}_n^*$ соответственно.

Элемент, обратный (относительно умножения) к элементу a, будет обозначаться как $\left(a^{-1} \bmod n\right)$. Операция деления в множестве $\mathbf{Z}_n^*$ определяется уравнением $a/b \equiv ab^{-1} \,(\mathrm{mod} n)$. Например, в множестве $\mathbf{Z}_{15}^*$ $7^{-1} = 13 \,(\mathrm{mod} 15)$, поскольку $7 \cdot 13 \equiv 91 \equiv 1 \,(\mathrm{mod} 15)$, поэтому $4/7 \equiv 4 \cdot 13 \equiv 7 \,(\mathrm{mod} 15)$.

Обозначим размер множества $\mathbf{Z}_n^*$ как $\phi\,(n)$. Эта функция, известная под названием ***ϕ-функции Эйлера*** (Euler's phi function), удовлетворяет уравнению

$$\phi\,(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right), \tag{31.19}$$

где индекс p пробегает значения всех простых делителей числа n (включая само n, если оно простое). Здесь мы не станем доказывать эту формулу, а попробуем дать для нее интуитивное пояснение. Выпишем все возможные остатки от деления на $n - \{0, 1, \ldots, n-1\}$, а затем для каждого простого делителя p числа n вычеркнем из этого списка все элементы, кратные p. Например, простые делители числа 45 — числа 3 и 5, поэтому

$$\phi\,(45) = 45\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{5}\right) = 45\left(\frac{2}{3}\right)\left(\frac{4}{5}\right) = 24.$$

Если p — простое число, то $\mathbf{Z}_p^* = \{1, 2, \ldots, p-1\}$ и

$$\phi\,(p) = p - 1. \tag{31.20}$$

Если же n — составное, то $\phi\,(n) < n - 1$.

Подгруппы

Если $(S, \oplus)$ — группа, $S' \subseteq S$ и $(S', \oplus)$ — тоже группа, то $(S', \oplus)$ называется ***подгруппой*** (subgroup) группы $(S, \oplus)$. Например, четные числа образуют подгруппу группы целых чисел с операцией сложения. Полезным инструментом для распознавания подгрупп является сформулированная ниже теорема.

Теорема 31.14. (Непустое замкнутое подмножество конечной группы является подгруппой). Если $(S, \oplus)$ — конечная группа, а S' — любое непустое подмножество множества S, такое что $a \oplus b \in S'$ для всех $a, b \in S'$, то $(S', \oplus)$ — подгруппа группы $(S, \oplus)$.

Доказательство. Доказать эту теорему читателю предлагается самостоятельно в упражнении 31.3-2. ■

Например, множество $\{0, 2, 4, 6\}$ образует подгруппу группы $\mathbf{Z}_8$, поскольку оно непустое и замкнутое относительно операции $+$ (т.е. оно замкнуто относительно операции $+_8$).

Сформулированная ниже теорема крайне полезна для оценки размера подгруппы; доказательство этой теоремы опущено.

Теорема 31.15 (Теорема Лагранжа). Если $(S, \oplus)$ — конечная группа, и $(S', \oplus)$ — ее подгруппа, то $|S'|$ — делитель числа $|S|$. ■

Подгруппу S' группы S называют ***истинной*** (proper) подгруппой, если $S' \neq S$. Приведенное ниже следствие из этой теоремы окажется полезным при анализе теста простых чисел Миллера-Рабина (Miller-Rabin), описанного в разделе 31.8.

Следствие 31.16. Если S' — истинная подгруппа конечной группы S, то $|S'| \leqslant \leqslant |S|/2$. ■

Подгруппы, сгенерированные элементом группы

Теорема 31.14 обеспечивает интересный способ, позволяющий сформировать подгруппу конечной группы $(S, \oplus)$. Для этого следует выбрать какой-нибудь элемент группы a и выделить все элементы, которые можно сгенерировать с помощью элемента a и групповой операции. Точнее говоря, определим элемент $a^{(k)}$ для $k > 1$ как

$$a^{(k)} = \bigoplus_{i=1}^{k} a = \underbrace{a \oplus a \oplus \cdots \oplus a}_{k} .$$

Например, если в группе $\mathbf{Z}_6$ задать элемент $a = 2$, то последовательность $a^{(1)}$, $a^{(2)}, \ldots$ имеет вид $2, 4, 0, 2, 4, 0, 2, 4, 0, \ldots$.

В группе $\mathbf{Z}_n$ справедливо соотношение $a^{(k)} = ka \bmod n$, а в группе $\mathbf{Z}_n^*$ — соотношение $a^{(k)} = a^k \bmod n$. ***Подгруппа, сгенерированная элементом*** a (subgroup generated by a) обозначается как $\langle a \rangle$ или $(\langle a \rangle, \oplus)$ и определяется следующим образом:

$$\langle a \rangle = \left\{ a^{(k)} : k \geqslant 1 \right\} .$$

Говорят, что элемент a ***генерирует*** (generates) подгруппу $\langle a\rangle$ или что a — ***генератор*** (generator) подгруппы $\langle a\rangle$. Поскольку множество S конечное, то $\langle a\rangle$ — конечное подмножество множества S, которое может содержать все элементы множества S. Поскольку из ассоциативности операции $\oplus$ следует соотношение

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)},$$

то подмножество $\langle a\rangle$ — замкнутое и, согласно теореме 31.14, $\langle a\rangle$ — подгруппа группы S. Например, в группе $\mathbf{Z}_6$ справедливы соотношения

$$\begin{aligned}\langle 0\rangle &= \{0\},\\ \langle 1\rangle &= \{0,1,2,3,4,5\},\\ \langle 2\rangle &= \{0,2,4\}.\end{aligned}$$

Аналогично, в группе $\mathbf{Z}_7^*$

$$\begin{aligned}\langle 1\rangle &= \{1\},\\ \langle 2\rangle &= \{1,2,4\},\\ \langle 3\rangle &= \{1,2,3,4,5,6\}.\end{aligned}$$

Порядок (order) элемента a (в группе S) обозначается как $\operatorname{ord}(a)$ и определяется как наименьшее положительное целое число t, для которого выполняется соотношение $a^{(t)} = e$.

Теорема 31.17. Для любой конечной группы $(S,\oplus)$ и любого ее элемента $a \in S$ порядок элемента равен размеру генерируемой им подгруппы, т.е. $\operatorname{ord}(a) = |\langle a\rangle|$.

Доказательство. Пусть $t = \operatorname{ord}(a)$. Поскольку для $k \geqslant 1$ справедливы соотношения $a^{(t)} = e$ и $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$, то при $i > t$ для некоторого $j < i$ выполняется равенство $a^{(i)} = a^{(j)}$. Таким образом, после элемента $a^{(t)}$ не появляется никаких новых элементов, так что $\langle a\rangle = \left\{a^{(1)}, a^{(2)}, \ldots, a^{(t)}\right\}$ и $|\langle a\rangle| \leqslant t$. Докажем неравенство $|\langle a\rangle| \geqslant t$ методом "от противного". Предположим, что при некоторых i и j, удовлетворяющих неравенству $1 \leqslant i < j \leqslant t$, выполняется равенство $a^{(i)} = a^{(j)}$. Тогда при $k \geqslant 0$ $a^{(i+k)} = a^{(j+k)}$. Однако из этого следует, что $a^{(i+(t-j))} = a^{(j+(t-j))} = e$. Так мы приходим к противоречию, поскольку $i+(t-j) < t$, но t — наименьшее положительное значение такое, что $a^{(t)} = e$. Поэтому все элементы последовательности $a^{(1)}, a^{(2)}, \ldots, a^{(t)}$ различны, и $|\langle a\rangle| \geqslant t$. Таким образом, мы приходим к заключению, что $\operatorname{ord}(a) = |\langle a\rangle|$. ■

Следствие 31.18. Последовательность $a^{(1)}, a^{(2)}, \ldots$ является периодической с периодом $t = \operatorname{ord}(a)$; т.е. $a^{(i)} = a^{(j)}$ тогда и только тогда, когда $i \equiv j \pmod t$. ■

С приведенным следствием согласуется определение $a^{(0)}$ как e, а $a^{(i)}$ для всех целых i как $a^{(i \bmod t)}$, где $t = \operatorname{ord}(a)$.

Следствие 31.19. Если $(S, \oplus)$ — конечная группа с единичным элементом e, то для всех $a \in S$ выполняется соотношение

$$a^{(|S|)} = e.$$

Доказательство. Из теоремы Лагранжа следует, что $\operatorname{ord}(a) \mid |S|$, так что $|S| \equiv \equiv 0 \pmod{t}$, где $t = \operatorname{ord}(a)$. Следовательно, $a^{(|S|)} = a^{(0)} = e$. ■

Упражнения

31.3-1. Составьте таблицу операций для групп $(\mathbf{Z}_4, +_4)$ и $(\mathbf{Z}_5^*, \cdot_5)$. Покажите, что эти группы изоморфны. Для этого обоснуйте взаимное однозначное соответствие α между элементами этих групп, такое что $a + b \equiv c \pmod{4}$ тогда и только тогда, когда $\alpha(a) \cdot \alpha(b) \equiv \alpha(c) \pmod{5}$.

31.3-2. Докажите теорему 31.14.

31.3-3. Покажите, что если p — простое число, а e — положительное целое число, то справедливо соотношение

$$\phi(p^e) = p^{e-1}(p-1).$$

31.3-4. Покажите, что для любого числа $n > 1$ и для любого $a \in \mathbf{Z}_n^*$ функция $f_a : \mathbf{Z}_n^* \to \mathbf{Z}_n^*$, определяемая соотношением $f_a(x) = ax \bmod n$, представляет собой перестановку $\mathbf{Z}_n^*$.

31.3-5. Перечислите все подгруппы групп $\mathbf{Z}_9$ и $\mathbf{Z}_{13}^*$.

31.4 Решение модульных линейных уравнений

Теперь рассмотрим задачу о решении уравнений вида

$$ax \equiv b \pmod{n}, \tag{31.21}$$

где $a > 0$ и $n > 0$. Эта задача имеет несколько приложений. Например, она используется как часть процедуры, предназначенной для поиска ключей для криптографической схемы RSA с открытым ключом, описанной в разделе 31.7. Предположим, что для заданных чисел a, b и n нужно найти все значения переменной x, которые удовлетворяют уравнению (31.21). Количество решений может быть равным нулю, единице, или быть больше единицы.

Обозначим через $\langle a \rangle$ подгруппу группы $\mathbf{Z}_n$, сгенерированную элементом a. Поскольку $\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \bmod n : x > 0\}$, уравнение (31.21) имеет

решение тогда и только тогда, когда $b \in \langle a \rangle$. В теореме Лагранжа (теорема 31.15) утверждается, что величина $|\langle a \rangle|$ должна быть делителем числа n. Сформулированная ниже теорема дает точную характеристику подгруппы $\langle a \rangle$.

Теорема 31.20. Для всех положительных целых чисел a и n из соотношения $d = \gcd(a, n)$ следует, что

$$\langle a \rangle = \langle d \rangle = \{0, d, 2d, \ldots, ((n/d) - 1)\, d\} \tag{31.22}$$

в $\mathbf{Z}_n$, так что $|\langle a \rangle| = n/d$.

Доказательство. Для начала покажем, что $d \in \langle a \rangle$. Напомним, что процедура EXTENDED_EUCLID(a, n) выдает целые числа x' и y', удовлетворяющие уравнению $ax' + ny' = d$. Таким образом, выполняется соотношение $ax' \equiv d \pmod{n}$, так что $d \in \langle a \rangle$.

Поскольку $d \in \langle a \rangle$, можно сделать вывод, что подгруппе $\langle a \rangle$ принадлежат все числа, кратные d, так как любое число, кратное числу, кратному a, является кратным числу a. Таким образом, подгруппа $\langle a \rangle$ содержит все элементы множества $\{0, d, 2d, \ldots, ((n/d) - 1)\, d\}$, т.е. $\langle d \rangle \subseteq \langle a \rangle$.

Теперь покажем, что $\langle a \rangle \subseteq \langle d \rangle$. Если $m \in \langle a \rangle$, то для некоторого целого числа x выполняется равенство $m = ax \bmod n$, так что $m = ax + ny$ для некоторого целого y. Однако $d \mid a$ и $d \mid n$, поэтому $d \mid m$ согласно уравнению (31.4). Таким образом, $m \in \langle d \rangle$.

Объединяя эти результаты, получаем $\langle a \rangle = \langle d \rangle$. Чтобы убедиться, что $|\langle a \rangle| = n/d$, заметим, что между числами 0 и $n - 1$ включительно, имеется ровно n/d чисел, кратных d. ■

Следствие 31.21. Уравнение $ax \equiv b \pmod{n}$ разрешимо относительно неизвестной величины x тогда и только тогда, когда $\gcd(a, n) \mid b$. ■

Следствие 31.22. Уравнение $ax \equiv b \pmod{n}$ либо имеет $d = \gcd(a, n)$ различных решений по модулю n, либо не имеет решений вовсе.

Доказательство. Если уравнение $ax \equiv b \pmod{n}$ имеет решение, то $b \in \langle a \rangle$. Согласно теореме 31.17, $\operatorname{ord}(a) = |\langle a \rangle|$, поэтому из следствия 31.18 и теоремы 31.20 вытекает, что последовательность $ai \bmod n$ при $i = 0, 1, \ldots, n - 1$ периодична с периодом $|\langle a \rangle| = n/d$. Если $b \in \langle a \rangle$ то b появляется в последовательности $ai \bmod n$ ($i = 0, 1, \ldots, n - 1$) ровно d раз, поскольку при возрастании индекса i от 0 до $n - 1$ блок элементов подгруппы $\langle a \rangle$ длиной n/d повторяется ровно d раз. Индексы x тех d позиций, для которых $ax \bmod n = b$, и являются решениями уравнения $ax \equiv b \pmod{n}$. ■

Теорема 31.23. Пусть $d = \gcd(a, n)$, и предположим, что равенство $d = ax' + by'$ выполняется для некоторых значений x' и y' (эти значения можно получить, например, с помощью процедуры EXTENDED_EUCLID). Если $d \mid b$, то одно из решений уравнения $ax \equiv b \pmod{n}$ — значение $x_0 = x'(b/d) \bmod n$.

Доказательство. Справедлива цепочка равенств

$$ax_0 \equiv ax'(b/d) \pmod{n} \equiv d(b/d) \pmod{n} \equiv b \pmod{n},$$

где второе тождество следует из того, что $ax' \equiv d \pmod{n}$, так что x_0 является решением уравнения $ax \equiv b \pmod{n}$. ■

Теорема 31.24. Предположим, что уравнение $ax \equiv b \pmod{n}$ имеет решение (т.е. $d \mid b$, где $d = \gcd(a, n)$), и что x_0 — некоторое решение этого уравнения. Тогда это уравнение имеет ровно d различных решений по модулю n, и эти решения выражаются соотношением $x_i = x_0 + i(n/d)$ при $i = 0, 1, \ldots, d-1$.

Доказательство. Поскольку $n/d > 0$, и $0 \leqslant i(n/d) < n$ при $i = 0, 1, \ldots, d-1$, все значения последовательности $x_0, x_1, \ldots, x_{d-1}$ по модулю n различны. Поскольку x_0 — решение уравнения $ax \equiv b \pmod{n}$, справедливо соотношение $ax_0 \bmod n = b$. Таким образом, при $i = 0, 1, \ldots, d-1$ выполняются соотношения

$$\begin{aligned} ax_i \bmod n &= a(x_0 + in/d) \bmod n = \\ &= (ax_0 + ain/d) \bmod n = \\ &= ax_0 \bmod n = \\ &= b, \end{aligned}$$

где третье равенство следует из того, что $d \mid a$, поэтому все x_i тоже являются решениями данного уравнения. В соответствии со следствием 31.22, всего имеется d различных решений, так что все рассматриваемые значения $x_0, x_1, \ldots, x_{d-1}$ должны быть таковыми. ■

Теперь у нас имеется математический аппарат, необходимый для решения уравнения вида $ax \equiv b \pmod{n}$, а ниже приведен алгоритм, который выводит все его решения. На вход алгоритма подаются произвольные положительные целые числа a и n, и произвольное целое число b.

```
MODULAR_LINEAR_EQUATION_SOLVER(a, b, n)
1  (d, x', y') ← EXTENDED_EUCLID(a, n)
2  if d | b
3     then x_0 ← x'(b/d) mod n
4          for i ← 0 to d − 1
5              do print (x_0 + i(n/d)) mod n
6     else  print "Решений нет"
```

В качестве примера работы этой процедуры рассмотрим уравнение $14x \equiv 30 \pmod{100}$ (здесь $a = 14$, $b = 30$ и $n = 100$). Взывав в строке 1 процедуру EXTENDED_EUCLID, получаем набор выходных данных $(d, x, y) = (2, -7, 1)$. Поскольку $2 \mid 30$, выполняются строки 3–5. В строке 3 вычисляется значение $x_0 = (-7)(15) \bmod 100 = 95$. В цикле в строках 4–5 выводятся два решения уравнения, равные 95 и 45.

Опишем работу процедуры MODULAR_LINEAR_EQUATION_SOLVER. В строке 1 вычисляется величина $d = \gcd(a, n)$, а также величины x' и y', такие что $d = ax' + ny'$, что свидетельствует о том, что x' удовлетворяет исходному уравнению. Если d не является делителем числа b, то уравнение решений не имеет согласно следствию 31.21. В строке 2 проверяется, делится ли b на d; если нет, то в строке 6 выводится сообщение об отсутствии решений заданного уравнения. В противном случае в строке 3, в соответствии с теоремой 31.23, вычисляется решение x_0 уравнения $ax \equiv b \pmod{n}$. Если найдено одно решение, то, согласно теореме 31.24, остальные $d - 1$ решений можно получить путем добавления величин, кратных (n/d) по модулю n. Это и делается в цикле **for** в строках 4–5, где с шагом (n/d) по модулю n выводятся все d решений, начиная с x_0.

В процедуре MODULAR_LINEAR_EQUATION_SOLVER выполняется $O(\lg n + \gcd(a, n))$ арифметических операций, поскольку в процедуре EXTENDED_EUCLID выполняется $O(\lg n)$ арифметических операций, и при каждой итерации цикла **for** в строках 4-5 производится фиксированное количество арифметических операций.

Сформулированные ниже следствия теоремы 31.24 позволяют сделать уточнения, представляющие особый интерес.

Следствие 31.25. Пусть $n > 1$. Если $\gcd(a, n) = 1$, то уравнение $ax \equiv b \pmod{n}$ имеет единственное решение по модулю n. ■

Значительный интерес представляет весьма распространенный случай $b = 1$. При этом искомое число x является ***мультипликативным обратным*** (multiplicative inverse) к числу a по модулю n.

Следствие 31.26. Пусть $n > 1$. Если $\gcd(a, n) = 1$, то уравнение $ax \equiv 1 \pmod{n}$ обладает единственным решением по модулю n. В противном случае оно не имеет решений. ■

Следствие 31.26 позволяет использовать запись $\left(a^{-1} \bmod n\right)$ для мультипликативных обратных чисел модулю n, где a и n — взаимно простые. Если $\gcd(a, n) = 1$, то единственное решение уравнения $ax \equiv 1 \pmod{n}$ — целое число x, возвращаемое процедурой EXTENDED_EUCLID, поскольку из уравнения

$$\gcd(a, n) = 1 = ax + ny$$

следует, что $ax \equiv 1 \pmod{n}$. Таким образом, можно эффективно вычислить величину $\left(a^{-1} \bmod n\right)$ с помощью процедуры EXTENDED_EUCLID.

Упражнения

31.4-1. Найдите все решения уравнения $35x \equiv 10 \pmod{50}$.

31.4-2. Докажите, что из уравнения $ax \equiv ay \pmod{n}$ следует соотношение $x \equiv y \pmod{n}$, если $\gcd(a, n) = 1$. Покажите, что условие $\gcd(a, n) = 1$ является необходимым, приведя контрпример при $\gcd(a, n) > 1$.

31.4-3. Рассмотрим приведенную ниже модификацию строки 3 в процедуре MODULAR_LINEAR_EQUATION_SOLVER:

3 **then** $x_0 \leftarrow x'(b/d) \bmod (n/d)$

Будет ли работать такая видоизмененная процедура? Обоснуйте ответ.

⋆ 31.4-4. Пусть $f(x) \equiv f_0 + f_1 x + \cdots + f_t x^t \pmod{p}$ — полином степени t с коэффициентами f_i, которые являются элементами множества $\mathbf{Z}_p$, где p — простое число. Говорят, что $a \in \mathbf{Z}_p$ — ***нуль*** (zero) полинома f, если $f(a) \equiv 0 \pmod{p}$. Докажите, что если a — нуль полинома f, то $f(x) \equiv (x - a)\, g(x) \pmod{p}$ для некоторого полинома $g(x)$ степени $t - 1$. Докажите методом математической индукции по t, что полином $f(x)$ степени t может иметь не более t различных нулей по модулю простого числа p.

31.5 Китайская теорема об остатках

Приблизительно в 100 г. н.э. китайский математик Сунь-Цзы (Sun-Tsŭ) решил задачу поиска целых чисел x, которые при делении на 3, 5 и 7 дают остатки 2, 3 и 2 соответственно. Одно из таких решений — $x = 23$, а общее решение имеет вид $23 + 105k$, где k — произвольное целое число. "Китайская теорема об остатках" устанавливает соответствие между системой уравнений по взаимно простым модулям (например, 3, 5 и 7) и уравнением по модулю произведения этих чисел (в данном примере, 105).

Китайская теорема об остатках имеет два основных применения. Пусть целое число n раскладывается на попарно взаимно простые множители $n = n_1 n_2 \ldots n_k$. Во-первых, эта теорема играет роль описательной "структурной теоремы". Согласно ей, структура $\mathbf{Z}_n$ представляет собой декартово произведение $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \times \cdots \times \mathbf{Z}_{n_k}$ с покомпонентным сложением и умножением по модулю n_i в i-м компоненте. Во-вторых, с помощью этого описания во многих случаях можно получить эффективные алгоритмы, поскольку работа с каждой из систем $\mathbf{Z}_{n_i}$ может оказаться эффективнее (в терминах битовых операций), чем работа по модулю n.

Теорема 31.27 (Китайская теорема об остатках). Пусть $n = n_1 n_2 \ldots n_k$, где n_i — попарно взаимно простые числа. Рассмотрим соответствие

$$a \leftrightarrow (a_1, a_2, \ldots, a_k), \tag{31.23}$$

где $a \in \mathbf{Z}_n$, $a_i \in \mathbf{Z}_{n_i}$ и $a_i = a \bmod n_i$ при $i = 1, 2, \ldots, k$. Тогда отображение (31.23) является взаимно однозначным соответствием (биекцией) между множеством $\mathbf{Z}_n$ и декартовым произведением $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \cdots \times \mathbf{Z}_{n_k}$. Операции, которые выполняются над элементами множества $\mathbf{Z}_n$, можно эквивалентно выполнять над соответствующими кортежами из k величин путем независимого выполнения операций над каждым компонентом. Таким образом, если

$$a \leftrightarrow (a_1, a_2, \ldots, a_k) \text{ и } b \leftrightarrow (b_1, b_2, \ldots, b_k),$$

то справедливы соотношения

$$(a + b) \bmod n \leftrightarrow ((a_1 + b_1) \bmod n_1, \ldots, (a_k + b_k) \bmod n_k), \tag{31.24}$$

$$(a - b) \bmod n \leftrightarrow ((a_1 - b_1) \bmod n_1, \ldots, (a_k - b_k) \bmod n_k), \tag{31.25}$$

$$(ab) \bmod n \leftrightarrow ((a_1 b_1) \bmod n_1, \ldots, (a_k b_k) \bmod n_k). \tag{31.26}$$

Доказательство. Преобразование от одного представления к другому осуществляется довольно просто. Переход от a к $(a_1, a_2 \ldots, a_k)$ очень простой, и для него требуется выполнить всего k операций деления. Вычислить элемент a по заданным входным элементам $(a_1, a_2 \ldots, a_k)$ несколько сложнее, и это делается следующим образом. Сначала определим величины $m_i = n/n_i$ для $i = 1, 2, \ldots, k$. Другими словами, m_i — произведение всех значений n_j, отличных от n_i: $m_i = n_1 n_2 \cdots n_{i-1} n_{i+1} \cdots n_k$. Затем определим

$$c_i = m_i \left(m_i^{-1} \bmod n_i \right) \tag{31.27}$$

для $i = 1, 2, \ldots, k$. Уравнение (31.27) всегда вполне определено: поскольку числа m_i и n_i взаимно простые (согласно теореме 31.6), следствие 31.26 гарантирует существование величины $\left(m_i^{-1} \bmod n_i \right)$. Наконец, величину a можно вычислить как функцию величин $(a_1, a_2 \ldots, a_k)$ по формуле

$$a \equiv (a_1 c_1 + a_2 c_2 + \cdots + a_k c_k) \,(\mathrm{mod} n). \tag{31.28}$$

Теперь покажем, что уравнение (31.28) обеспечивает справедливость соотношения $a \equiv a_i \,(\mathrm{mod} n_i)$ при $i = 1, 2, \ldots, k$. Заметим, что если $j \neq i$, то $m_j \equiv 0 \;(\mathrm{mod} n_i)$, откуда следует, что $c_j \equiv m_j \equiv 0 \,(\mathrm{mod} n_i)$. Заметим также, что из уравнения (31.27) следует $c_1 \equiv 1 \,(\mathrm{mod} n_i)$. Таким образом, получаем красивое и полезное соответствие

$$c_i \leftrightarrow (0, 0, \ldots, 0, 1, 0, \ldots, 0),$$

где все компоненты равны нулю, кроме i-го, который равен 1. Векторы c_i в определенном смысле образуют базис представления. Поэтому для каждого i можно

записать

$$\begin{aligned} a &\equiv a_i c_i \,(\mathrm{mod} n_i) \equiv \\ &\equiv a_i m_i \left(m_i^{-1} \bmod n_i\right) (\mathrm{mod} n_i) \equiv \\ &\equiv a_i \,(\mathrm{mod} n_i)\,, \end{aligned}$$

что и требовалось показать: представленный метод вычисления величины a по заданным значениям a_i дает результат, удовлетворяющий условию $a \equiv a_i \,(\mathrm{mod} n_i)$ для $i = 1, 2, \ldots, k$. Это соответствие взаимно однозначное, поскольку преобразования можно производить в обоих направлениях. Наконец, уравнения (31.24)–(31.26) непосредственно следуют из результатов упражнения 31.1-6, поскольку соотношение $x \bmod n_i = (x \bmod n) \bmod n_i$ справедливо при любом x и $i = 1, 2, \ldots, k$. ■

Сформулируем следствия, которые понадобятся нам в последующих разделах.

Следствие 31.28. Если $n_1, n_2, \ldots, n_k$ — попарно взаимно простые числа и $n = n_1 n_2 \cdots n_k$, то для любого набора целых чисел $a_1, a_2, \ldots a_k$ система уравнений

$$x \equiv a_i \,(\mathrm{mod} n_i)$$

при $i = 1, 2, \ldots, k$ имеет единственное решение по модулю n относительно неизвестной величины x. ■

Следствие 31.29. Если $n_1, n_2, \ldots, n_k$ — попарно взаимно простые числа и $n = n_1 n_2 \cdots n_k$, то для любых целых чисел x и a соотношение

$$x \equiv a \,(\mathrm{mod} n_i)$$

для $i = 1, 2, \ldots, k$ выполняется тогда и только тогда, когда

$$x \equiv a \,(\mathrm{mod} n)\,.$$

■

В качестве примера применения китайской теоремы об остатках рассмотрим систему уравнений

$$\begin{aligned} a &\equiv 2 \,(\mathrm{mod} 5)\,, \\ a &\equiv 3 \,(\mathrm{mod} 13)\,, \end{aligned}$$

так что $a_1 = 2$, $n_1 = m_2 = 5$, $a_2 = 3$ и $n_2 = m_1 = 13$. Нам нужно вычислить величину $a \bmod 65$, так как $n = 65$. Поскольку $13^{-1} \equiv 2 \,(\mathrm{mod} 5)$ и $5^{-1} \equiv 8 \,(\mathrm{mod} 13)$, получаем

$$\begin{aligned} c_1 &= 13 \,(2 \bmod 5) = 26, \\ c_2 &= 5 \,(8 \bmod 13) = 40, \end{aligned}$$

Таблица 31.4. Иллюстрация китайской теоремы об остатках при $n_1 = 5$ и $n_2 = 13$

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

и

$$\begin{aligned} a &\equiv (2 \cdot 26 + 3 \cdot 40) \,(\mathrm{mod} 65) \equiv \\ &\equiv (52 + 120) \,(\mathrm{mod} 65) \equiv \\ &\equiv 42 \,(\mathrm{mod} 65) \,. \end{aligned}$$

Данный пример проиллюстрирован в табл. 31.4. На пересечении строки i и столбца j приведено значение a по модулю 65, удовлетворяющее уравнениям $(a \bmod 5) = i$ и $(a \bmod 13) = j$. Обратите внимание, что в ячейке на пересечении строки 0 и столбца 0 содержится значение 0. Поскольку $c_1 = 26$, сдвиг вниз вдоль столбца приводит к увеличению значения a на 26. Аналогично, то, что $c_2 = 40$, означает, что сдвиг вправо вдоль строки приводит к увеличению значения a на 40. Увеличение значения a на 1 соответствует сдвигу по диагонали вниз и вправо.

Таким образом, операции по модулю n можно выполнять непосредственно, а можно перейти к представлению, в котором вычисления производятся отдельно по модулям n_i. Эти вычисления полностью эквивалентны, и есть возможность выбирать более удобный способ.

Упражнения

31.5-1. Найдите все решения уравнений $x \equiv 4 \,(\mathrm{mod} 5)$ и $x \equiv 5 \,(\mathrm{mod} 11)$.

31.5-2. Найдите все целые числа x, которые при делении на 9, 8 и 7 дают остатки 1, 2 и 3 соответственно.

31.5-3. Докажите, что при выполнении условий теоремы 31.27, если $\gcd(a, n) = 1$, то

$$\left(a^{-1} \bmod n\right) \leftrightarrow \left(\left(a_1^{-1} \bmod n_1\right), \left(a_2^{-1} \bmod n_1\right), \ldots, \left(a_k^{-1} \bmod n_k\right)\right).$$

31.5-4. Докажите, что при выполнении условий теоремы 31.27 количество корней уравнения $f(x) \equiv 0 \,(\mathrm{mod} n)$, где f — произвольный полином, равно произведению количества корней уравнений $f(x) \equiv 0 \,(\mathrm{mod} n_1)$, $f(x) \equiv 0 \,(\mathrm{mod} n_2)$, …, $f(x) \equiv 0 \,(\mathrm{mod} n_k)$.

31.6 Степени элемента

Рассматривать последовательность

$$a^0, a^1, a^2, a^3, \ldots \tag{31.29}$$

степеней числа a по модулю n, где $a \in \mathbf{Z}_n^*$, так же естественно, как и последовательность чисел, кратных a по модулю n. Индексация начинается от нуля; нулевой член последовательности равен $a^0 \bmod n = 1$, а i-й член — $a^i \bmod n = 1$. Ниже в качестве примера приведены степени чисел 3 и 2 по модулю 7:

i	0	1	2	3	4	5	6	7	8	9	10	11	$\cdots$
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2	6	4	5	$\cdots$
$2^i \bmod 7$	1	2	4	1	2	4	1	2	4	1	2	4	$\cdots$

В этом разделе при помощи $\langle a \rangle$ будет обозначаться подгруппа группы $\mathbf{Z}_n^*$, сгенерированная элементом a путем повторного умножения, а $\mathrm{ord}_n(a)$ (порядок a по модулю n) будет обозначать порядок элемента a в группе $\mathbf{Z}_n^*$. Например, в группе $\mathbf{Z}_7^*$ $\langle 2 \rangle = \{1, 2, 4\}$, а $\mathrm{ord}_7(2) = 3$. Используя определение функции Эйлера $\phi(n)$ в качестве размера группы $\mathbf{Z}_n^*$ (см. упражнение 31.3), преобразуем следствие 31.19 с использованием обозначений $\mathbf{Z}_n^*$, чтобы получить теорему Эйлера и сформулировать ее для частного случая группы $\mathbf{Z}_p^*$, где p — простое число, что даст нам теорему Ферма.

Теорема 31.30 (Теорема Эйлера). При любом целом значении $n > 1$ для всех $a \in \mathbf{Z}_n^*$

$$a^{\phi(n)} \equiv 1 \,(\mathrm{mod} n)\,.$$

■

Теорема 31.31 (Теорема Ферма). Если p — простое число, то для всех $a \in \mathbf{Z}_n^*$

$$a^{p-1} \equiv 1 \,(\mathrm{mod} p)\,.$$

Доказательство. Согласно уравнению (31.20), для простых чисел p функция Эйлера равна $\phi(p) = p - 1$. ■

Это следствие применимо к каждому элементу группы $\mathbf{Z}_p$ за исключением нуля, поскольку $0 \notin \mathbf{Z}_p^*$. Однако для всех $a \in \mathbf{Z}_p$, если p — простое число, справедливо соотношение $a^p \equiv a \,(\mathrm{mod} p)$.

Если $\mathrm{ord}_n(g) = |\mathbf{Z}_n^*|$, то каждый элемент группы $\mathbf{Z}_n^*$ является степенью элемента g по модулю n, и говорят, что g — ***первообразный корень*** (primitive root), или ***генератор*** (generator) группы $\mathbf{Z}_n^*$. Например, 3 — первообразный корень по модулю 7, но 2 таковым не является. Если в группе $\mathbf{Z}_n^*$ существует первообразный корень, то говорят, что она ***циклическая*** (cyclic). Доказательство сформулированной ниже теоремы, доказанной Нивеном (Niven) и Цукерманом (Zuckerman) [231], опускается.

Теорема 31.32. Значения $n > 1$, для которых группа $\mathbf{Z}_n^*$ является циклической, равны 2, 4, p^e и $2p^e$ для всех простых чисел $p > 2$ и всех положительных целых показателях степени e. ■

Если g — первообразный корень группы $\mathbf{Z}_n^*$, а a — произвольный элемент этой группы, то существует такой элемент z, что $g^z \equiv a \pmod{n}$. Это значение z называется ***дискретным логарифмом*** (discrete logarithm) или ***индексом*** (index) элемента a по модулю n по основанию g, и обозначается как $\mathrm{ind}_{n,g}(a)$.

Теорема 31.33 (Теорема о дискретном логарифме). Если g — первообразный корень группы $\mathbf{Z}_n^*$, то уравнение $g^x \equiv g^y \pmod{n}$ справедливо тогда и только тогда, когда выполняется соотношение $x \equiv y \pmod{\phi(n)}$.

Доказательство. Сначала предположим, что $x \equiv y \pmod{\phi(n)}$. Тогда существует такое целое значение k, для которого выполняется равенство $x = y + k\phi(n)$. Таким образом получаем

$$\begin{aligned} g^x &\equiv g^{y+k\phi(n)} \pmod{n} \equiv \\ &\equiv g^y \cdot \left(g^{\phi(n)}\right)^k \pmod{n} \equiv \\ &\equiv g^y \cdot 1^k \pmod{n} \equiv \\ &\equiv g^y \pmod{n}, \end{aligned}$$

где предпоследнее тождество следует из теоремы Эйлера. Теперь предположим, что $g^x \equiv g^y \pmod{n}$. Поскольку последовательность степеней g генерирует все элементы подгруппы $\langle g \rangle$, и $|\langle g \rangle| = \phi(n)$, то из следствия 31.18 вытекает, что последовательность степеней элемента g периодична с периодом $\phi(n)$. Таким образом, если $g^x \equiv g^y \pmod{n}$, то справедливо соотношение $x \equiv y \pmod{\phi(n)}$. ■

Вычисление дискретных логарифмов иногда позволяет упростить рассуждения, касающиеся модульных уравнений. Это иллюстрирует доказательство приведенной ниже теоремы.

Теорема 31.34. Если p — нечетное простое число и $e \geqslant 1$, то уравнение

$$x^2 \equiv 1 \pmod{p^e} \tag{31.30}$$

имеет всего два решения, а именно $x = \pm 1$.

Доказательство. Пусть $n = p^e$. Из теоремы 31.32 следует, что группа $\mathbf{Z}_n^*$ имеет первообразный корень g. Уравнение (31.30) можно переписать следующим образом:

$$\left(g^{\mathrm{ind}_{n,g}(x)}\right)^2 \equiv g^{\mathrm{ind}_{n,g}(1)} \pmod{n}. \tag{31.31}$$

С учетом того, что $\mathrm{ind}_{n,g}(1) = 0$, из теоремы 31.33 следует, что уравнение (31.31) эквивалентно следующему:

$$2 \cdot \mathrm{ind}_{n,g}(x) \equiv 0 \pmod{\phi(n)}. \tag{31.32}$$

Чтобы решить это уравнение относительно неизвестной величины $\mathrm{ind}_{n,g}(x)$, воспользуемся методами, изложенными в разделе 31.4. Согласно уравнению (31.19), имеем $\phi(n) = p^e(1 - 1/p) = (p-1)p^{e-1}$. Введя обозначение $d = \gcd(2, \phi(n)) = \gcd\left(2, (p-1)p^{e-1}\right) = 2$ и заметив, что $d \mid 0$, согласно теореме 31.24 выясняем, что уравнение (31.32) имеет ровно $d = 2$ решений. Итак, уравнение (31.32) имеет ровно 2 решения, которые, как и ожидалось, равны $x = 1$ и $x = -1$. ■

Число x называется ***нетривиальным квадратным корнем из 1 по модулю*** n (nontrivial square root of 1, modulo n), если справедливо уравнение $x^2 \equiv 1 \pmod{n}$, но x не эквивалентно ни одному из "тривиальных" квадратных корней по модулю n, — ни 1, ни -1. Например, 6 — нетривиальный квадратный корень 1 по модулю 35. Приведенное ниже следствие теоремы 31.34 будет использовано в разделе 31.8 при доказательстве корректности процедуры Миллера-Рабина (Miller-Rabin), в которой производится проверка простоты чисел.

Следствие 31.35. Если существует нетривиальный квадратный корень 1 по модулю n, то число n — составное.

Доказательство. Используем теорему, обратную теореме 31.34: если существует нетривиальный квадратный корень 1 по модулю n, то n не может быть нечетным простым числом или его степенью. Если $x^2 \equiv 1 \pmod{2}$, то $x \equiv 1 \pmod{2}$, поэтому все квадратные корни 1 по модулю 2 тривиальны. Таким образом, n не может быть простым. Наконец, чтобы существовал нетривиальный квадратный корень 1, n должно удовлетворять неравенству $n > 1$. Поэтому число n должно быть составным. ■

Возведение в степень путем последовательного возведения в квадрат

В теоретико-числовых вычислениях часто встречается операция возведения одного числа в степень по модулю другого числа, известная под названием ***модульное возведение в степень*** (modular exponentiation). Другими словами, нам нужно найти эффективный способ вычисления величины $a^b \bmod n$, где a и b — неотрицательные целые числа, а n — положительное целое число. Операция модульного возведения в степень играет важную роль во многих подпрограммах, производящих проверку простоты чисел, а также в криптографической системе с открытым ключом RSA. Метод эффективного решения этой задачи, в котором

используется бинарное представление числа b, называется методом ***повторного возведения в квадрат*** (repeated squaring).

Пусть $\langle b_k, b_{k-1}, \ldots, b_1, b_0 \rangle$ — бинарное представление числа b. (Длина бинарного представления равна $k+1$, старший бит — b_k, младший — b_0.) Приведенная ниже процедура вычисляет $a^c \bmod n$, где индекс c в ходе работы процедуры удваивается и увеличивается на 1, возрастая от 0 до b.

```
MODULAR_EXPONENTIATION(a, b, n)
 1  c ← 0
 2  d ← 1
 3  Пусть ⟨b_k, b_{k-1}, ..., b_0⟩ — бинарное представление числа b
 4  for i ← k downto 0
 5       do c ← 2c
 6          d ← (d · d) mod n
 7          if b_i = 1
 8             then c ← c + 1
 9                  d ← (d · a) mod n
10  return d
```

Важной частью процедуры, объясняющей ее название (“повторное возведение в степень”), является возведение в квадрат в строке 6, которое производится в каждой итерации. В качестве иллюстрации работы алгоритма рассмотрим пример $a = 7$, $b = 560$ и $n = 561$. В этом случае алгоритм вычисляет последовательность величин по модулю 561, приведенную в табл. 31.5. Использованная в алгоритме последовательность показателей степени содержится в строке таблицы с меткой c.

Значения переменной c на самом деле не нужны; они включены только для пояснения. В алгоритме поддерживается сформулированный ниже инвариант цикла, состоящий их двух частей.

> Непосредственно перед каждой итерацией цикла **for** в строках 4–9 выполняются следующие условия.
>
> 1. Значение величины c совпадает с префиксной частью $\langle b_k, b_{k-1}, \ldots, b_{i+1} \rangle$ бинарного представления числа b.
> 2. $d = a^c \bmod n$.

Таблица 31.5. Работа процедуры MODULAR_EXPONENTIATION по вычислению $7^{560} \pmod{561}$

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

Используем этот инвариант цикла следующим образом.

Инициализация. Изначально $i = k$, поэтому префикс $\langle b_k, b_{k-1}, \ldots, b_{i+1}\rangle$ пуст, что соответствует значению $c = 0$. Кроме того, $d = 1 = a^0 \bmod n$.

Сохранение. Обозначим через c' и d' значения переменных c и d в конце итерации цикла **for**, которые в то же время являются значениями перед следующей итерацией. В каждой итерации эти значения обновляются с помощью операции $c' \leftarrow 2c$ (если $b_i = 0$) или $c' \leftarrow 2c + 1$ (если $b_i = 1$), так что перед следующей итерацией значение c оказывается корректным. Если $b_i = 0$, то

$$d' = d^2 \bmod n = (a^c)^2 \bmod n = a^{2c} \bmod n = a^{c'} \bmod n.$$

Если же $b_i = 1$, то

$$d' = d^2 a \bmod n = (a^c)^2\, a \bmod n = a^{2c+1} \bmod n = a^{c'} \bmod n.$$

В каждом случае перед очередной итерацией выполняется равенство $d = a^c \bmod n$.

Завершение. В момент завершения $i = -1$. Таким образом, $c = b$, поскольку значение этой переменной равно префиксу $\langle b_k, b_{k-1}, \ldots, b_1, b_0\rangle$ бинарного представления b. Следовательно, $d = a^c \bmod n = a^b \bmod n$.

Если входные числа a, b и n являются β-битовыми, то общее количество необходимых арифметических операций равно $O(\beta)$, а общее количество необходимых битовых операций — $O(\beta^3)$.

Упражнения

31.6-1. Составьте таблицу, в которой был бы показан порядок каждого элемента группы $\mathbf{Z}_{11}^*$. Выберите наименьший первообразный корень g и вычислите таблицу значений $\mathrm{ind}_{11,g}(x)$ для всех $x \in \mathbf{Z}_{11}^*$.

31.6-2. Разработайте алгоритм модульного возведения в степень, в котором биты числа b проверяются не слева направо, а справа налево.

31.6-3. Считая величину $\phi(n)$ известной, объясните, как с помощью процедуры MODULAR_EXPONENTIATION вычислить величину $a^{-1} \bmod n$ для любого $a \in \mathbf{Z}_{11}^*$.

31.7 Криптосистема с открытым ключом RSA

Криптографическую систему с открытым ключом можно использовать для шифровки сообщений, которыми обмениваются два партнера, чтобы посторонний, перехвативший зашифрованное сообщение, не смог его расшифровать. Кроме того, криптографическая система с открытым ключом позволяет партнерам

добавлять в конце электронных сообщений “цифровые подписи”. Такая подпись представляет собой электронную версию подписи, поставленной от руки на бумажном документе. Любой без труда сможет ее проверить, но подделать не сможет никто. Кроме того, при изменении хотя бы одного бита в электронном сообщении оно теряет свою достоверность. Таким образом, цифровая подпись не только позволяет идентифицировать автора сообщения, но и обеспечивает целостность его содержимого. Она является прекрасным инструментом для подписанных с помощью электронных средств деловых контрактов, электронных чеков, оформляемых в электронном виде заказов на поставку и других электронных документов, которые необходимо идентифицировать.

Криптографическая система с открытым ключом RSA основана на драматическом различии в том, насколько легко находить большие простые числа и насколько сложно раскладывать на множители произведение двух больших простых чисел. В разделе 31.8 описана эффективная процедура поиска больших простых чисел, а в разделе 31.9 обсуждается проблема разложения больших целых чисел на множители.

Криптографические системы с открытым ключом

В криптографической системе с открытым ключом каждый участник располагает как ***открытым ключом*** (public key), так и ***секретным ключом*** (secret key). Каждый ключ — это часть информации. Например, в криптографической системе RSA каждый ключ состоит из пары целых чисел. Пусть в процессе обмена сообщениями принимают участие “Алиса” и “Борис”. Обозначим открытый и секретный ключи Алисы через P_A и S_A, а Бориса — через P_B и S_B.

Каждый участник создает свой открытый и секретный ключ самостоятельно. Секретный ключ каждый из них держит в секрете, а открытые ключи можно сообщать кому угодно или даже публиковать их. Фактически, часто удобно предполагать, что открытый ключ каждого пользователя доступен в каталоге общего пользования, поэтому любой участник общения может легко получить открытый ключ любого другого участника.

Открытый и секретный ключи задают функции, применимые к любому сообщению. Обозначим через $\mathcal{D}$ множество допустимых сообщений. Например, $\mathcal{D}$ может быть множеством битовых последовательностей конечной длины. В простейшей первоначальной формулировке криптографии с открытым ключом требуется, чтобы открытый и секретный ключи задавали взаимно однозначную функцию, отображающую множество $\mathcal{D}$ само на себя. Обозначим функцию, соответствующую открытому ключу Алисы P_A, через $P_A()$, а функцию, соответствующую ее секретному ключу S_A — через $S_A()$. Таким образом, функции $P_A()$ и $S_A()$ являются перестановками множества $\mathcal{D}$. Предполагается, что существует эффек-

тивный алгоритм вычисления функций $P_A()$ и $S_A()$ по заданным ключам P_A и S_A.

Открытый и секретный ключи каждого участника обмена сообщениями образуют "согласованную пару" в том смысле, что они являются взаимно обратными. Другими словами, для любого сообщения $M \in \mathcal{D}$ выполняются соотношения

$$M = S_A(P_A(M)), \tag{31.33}$$

$$M = P_A(S_A(M)). \tag{31.34}$$

Произведя последовательное преобразование сообщения M с помощью ключей P_A и S_A (в любом порядке), снова получим сообщение M.

В криптографической системе с открытым ключом существенное обстоятельство заключается в том, чтобы никто, кроме Алисы, не мог вычислить функцию $S_A()$ за приемлемое с практической точки зрения время. Секретность сообщений электронной почты, зашифрованных и отправленных Алисе, а также подлинность цифровых подписей Алисы основывается на предположении о том, что только Алиса способна вычислить функцию $S_A()$. Это требование является причиной, по которой Алиса должна держать в секрете ключ S_A. Если бы она этого не делала, то ее ключ потерял бы секретность и криптографическая система не смогла бы предоставить Алисе упомянутые выше возможности. Предположение о том, что только Алиса может вычислить функцию $S_A()$, должно соблюдаться даже несмотря на то, что каждому известен ключ P_A и каждый может эффективно вычислить функцию $P_A()$, обратную к функции $S_A()$. Основная сложность, возникающая при разработке работоспособной криптографической системы с открытым ключом, заключается в создании системы, в которой можно легко найти преобразование $P_A()$, но невозможно (или очень сложно) вычислить соответствующее обратное преобразование $S_A()$.

В криптографической системе с открытым ключом кодирование производится так, как показано на рис. 31.1. Предположим, Борис хочет отправить Алисе сообщение M, зашифрованное таким образом, чтобы для постороннего оно выглядело как бессмысленный набор символов. Сценарий отправки сообщения можно представить следующим образом.

- Борис получает открытый ключ Алисы P_A (из каталога общего пользования или каталога Алисы).
- Борис производит вычисления $C = P_A(M)$, результатом которых является ***зашифрованный текст*** (ciphertext), соответствующий сообщению M, и отправляет Алисе сообщение C.
- Получив зашифрованное сообщение C, Алиса применяет свой секретный ключ S_A, чтобы преобразовать его в исходное сообщение: $M = S_A(C)$.

Поскольку функции $S_A()$ и $P_A()$ являются обратными по отношению друг к другу, Алиса имеет возможность получить сообщение M, располагая сообщением C.

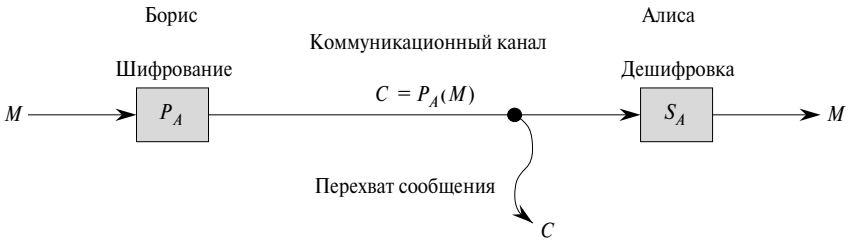


Рис. 31.1. Процесс шифрования в схеме с открытым ключом

Поскольку только Алиса может вычислять функцию $S_A()$, она — единственная, кто способен расшифровать сообщение C. Шифрование сообщения M с помощью функции $P_A()$ защищает его содержимое от всех, кроме Алисы.

Реализовать цифровые подписи в сформулированной модели криптографической системы с открытым ключом так же просто. (Заметим, что существуют другие подходы к проблеме создания цифровых подписей, которые здесь не рассматриваются.) Предположим, что Алисе нужно отправить Борису ответ M', подтвержденный цифровой подписью. Сценарий создания цифровой подписи проиллюстрирован на рис. 31.2.

- Алиса создает ***цифровую подпись*** (digital signature) σ для сообщения M' с помощью своего цифрового ключа S_A и уравнения $\sigma = S_A(M')$.
- Алиса отправляет Борису пару (M', σ), состоящую из сообщения и подписи.
- Получив пару (M', σ), Борис с помощью открытого ключа Алисы может проверить, что автор сообщения M' — действительно Алиса. (Сообщение M' может содержать имя Алисы, чтобы Борис знал, чей открытый ключ использовать.) Для проверки используется уравнение $M' = P_A(\sigma)$. Если уравнение выполняется, можно смело делать вывод, что сообщение M' действительно подписано Алисой. В противном случае Борис с полным основанием сможет заподозрить, что сообщение M' или цифровая подпись σ были искажены в процессе передачи, либо что пара (M', σ) — попытка подлога.

Поскольку цифровая подпись обеспечивает как аутентификацию автора сообщения, так и подтверждение целостности содержимого подписанного сообщения, она служит аналогом подписи, сделанной от руки в конце рукописного документа.

Важное свойство цифровой подписи заключается в том, что ее может проверить каждый, кто имеет доступ к открытому ключу ее автора. Один из участников обмена сообщениями после проверки подлинности цифровой подписи может передать подписанное сообщение еще кому-то, кто тоже в состоянии проверить эту подпись. Например, Алиса может переслать Борису в сообщении электронный чек. После того как Борис проверит подпись Алисы на чеке, он может передать

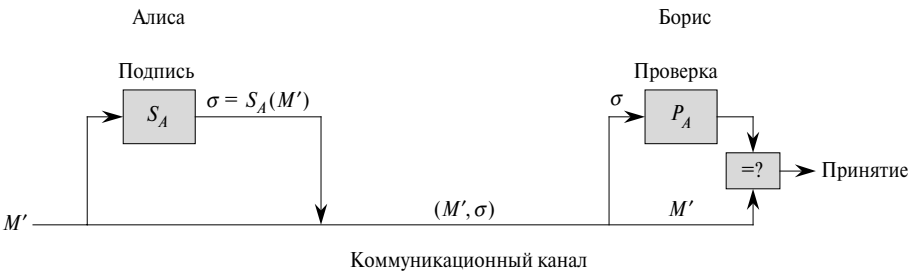


Рис. 31.2. Цифровые подписи в системе с открытым ключом

его в свой банк, служащие которого также имеют возможность проверить подпись и осуществить соответствующую денежную операцию.

Заметим, что подписанное сообщение не зашифровано; оно пересылается в исходном виде и его содержимое не защищено. Путем совместного применения приведенных выше схем можно создавать сообщения, которые будут и зашифрованы, и содержать цифровую подпись. Для этого автор сначала должен добавить к сообщению свою цифровую подпись, а затем — зашифровать получившуюся в результате пару (состоящую из самого сообщения и подписи к нему) с помощью открытого ключа, принадлежащего получателю. Получатель расшифровывает полученное сообщение с помощью своего секретного ключа. Таким образом он получит исходное сообщение и цифровую подпись отправителя, которую он затем сможет проверить с помощью соответствующего открытого ключа. Если проводить аналогию с пересылкой обычных бумажных документов, то этот процесс похож на то, как если бы автор документа поставил под ним свою подпись, а затем положил его в бумажный конверт и запечатал, с тем чтобы конверт был распечатан только тем человеком, кому адресовано сообщение.

Криптографическая система RSA

В ***криптографической системе с открытым ключом RSA*** (RSA public-key cryptosystem) участники обмена сообщениями создают свои открытые и секретные ключи в соответствии с описанной ниже процедурой.

1. Случайным образом выбираются два больших (скажем, длиной 512 битов) простых числа p и q, причем $p \neq q$.
2. Вычисляется число $n = pq$.
3. Выбирается маленькое нечетное целое число e, взаимно простое со значением функции $\phi(n)$ (которое, согласно уравнению (31.19), равно $(p-1) \times \times (q-1)$).
4. Вычисляется число d, мультипликативное обратное по модулю $\phi(n)$ к e. (В соответствии со следствием 31.26, такое число существует и является

единственным. Чтобы вычислить d по заданным $\phi(n)$ и e, можно воспользоваться методом, описанным в разделе 31.4.)

5. Пара $P = (e, n)$ публикуется в качестве ***открытого ключа RSA*** (RSA pubic key).
6. Пара $S = (d, n)$, играющая роль ***секретного ключа RSA*** (RSA secret key), сохраняется в секрете.

В этой схеме в роли области $\mathcal{D}$ выступает множество $\mathbf{Z}_n$. Преобразование сообщения M, связанное с открытым ключом $P = (e, n)$, имеет вид

$$P(M) = M^e \pmod{n}, \tag{31.35}$$

а преобразование зашифрованного текста C, связанное с секретным ключом $S = (d, n)$, выполняется как

$$S(C) = C^d \pmod{n}. \tag{31.36}$$

Эти уравнения можно использовать для создания как зашифрованных сообщений, так и цифровых подписей. Чтобы создать подпись, ее автор применяет свой секретный ключ не к зашифрованному, а к подписываемому сообщению. Для проверки подписи открытый ключ подписавшегося применяется именно к подписи, а не для шифровки сообщения.

Операции с открытым и секретным ключами можно реализовать с помощью процедуры MODULAR_EXPONENTIATION, описанной в разделе 31.6. Чтобы проанализировать время выполнения этих операций, предположим, что открытый ключ (e, n) и секретный ключ (d, n) удовлетворяют соотношениям $\lg e = O(1)$, $\lg d \leqslant \beta$ и $\lg n \leqslant \beta$. Тогда в процессе применения открытого ключа выполняется $O(1)$ умножений по модулю и используется $O(\beta^2)$ битовых операций. В ходе применения секретного ключа выполняется $O(\beta)$ умножений по модулю и используется $O(\beta^3)$ битовых операций.

Теорема 31.36 (Корректность схемы RSA). Уравнения (31.35) и (31.36), на которых основана схема RSA, определяют взаимно обратные преобразования множества $\mathbf{Z}_n$, удовлетворяющие уравнениям (31.33) и (31.34).

Доказательство. Из уравнений (31.35) и (31.36) для любого $M \in \mathbf{Z}_n$ получаем, что

$$P(S(M)) = S(P(M)) = M^{ed} \pmod{n}.$$

Поскольку числа e и d являются взаимно обратные относительно умножения по модулю $\phi(n) = (p-1)(q-1)$,

$$ed = 1 + k(p-1)(q-1)$$

при некотором целом k. Тогда, если $M \not\equiv 0 \pmod{p}$, можно записать

$$\begin{aligned} M^{ed} &\equiv M\left(M^{p-1}\right)^{k(q-1)} \pmod{p} \equiv \\ &\equiv M\,(1)^{k(q-1)} \pmod{p} \equiv \\ &\equiv M \pmod{p}, \end{aligned}$$

где второе тождество следует из теоремы 31.31. Если же $M \equiv 0 \pmod{p}$, то $M^{ed} \equiv$ $\equiv M \pmod{p}$. Таким образом, при всех M выполняется равенство

$$M^{ed} \equiv M \pmod{p}.$$

Аналогично можно показать, что для всех M

$$M^{ed} \equiv M \pmod{q}.$$

Таким образом, согласно следствию 31.29 из Китайской теоремы об остатках, для всех M выполняется равенство

$$M^{ed} \equiv M \pmod{n}.$$

■

Безопасность криптографической системы RSA в значительной мере основана на сложностях, связанных с разложением больших целых чисел на множители. Если бы кто-нибудь посторонний мог разложить на множители элемент открытого ключа n, то он смог бы использовать множители p и q точно так же, как это сделал создатель открытого ключа, и получить таким образом секретный ключ из открытого. Поэтому если бы разложение больших целых чисел было простой задачей, то так же легко было бы взломать криптографическую систему RSA. Обратное утверждение, которое состоит в том, что сложность взлома схемы RSA обусловлена сложностью разложения больших целых чисел на множители, не доказано. Однако на протяжении двадцатилетних исследований не удалось найти более простой способ взлома, чем тот, при котором необходимо разложить на множители число n. В разделе 31.9 будет показано, что задача разложения больших целых чисел на множители оказалась на удивление сложной. Путем случайного выбора и перемножения друг на друга двух 512-битовых простых чисел можно создать открытый ключ, который не удается "разбить" за приемлемое время с помощью имеющихся в наличии технологий. Если в разработке теоретико-числовых алгоритмов не произойдет фундаментального прорыва, и если реализовывать криптографическую схему RSA, придерживаясь сформулированных ниже рекомендаций, то эта схема способна обеспечить высокую степень надежности.

Однако для достижения безопасности в криптографической схеме RSA желательно работать с целыми числами, длина которых достигает нескольких сотен битов. Это позволит уберечься от возможных достижений в искусстве разложения

чисел на множители. Во время написания этой книги (2001 год) в RSA-модулях обычно использовались числа, длина которых находилась в пределах от 768 до 2048 битов. Для создания таких модулей необходимо иметь возможность эффективно находить большие простые числа. Этой задаче посвящен раздел 31.8.

Для повышения эффективности схема RSA часто используется в "гибридном", или "управляемом ключом" режиме совместно с криптографическими системами, не обладающими открытым ключом. В такой системе ключи, предназначенные для зашифровки и расшифровки, идентичны. Если Алисе нужно в приватном порядке отправить Борису длинное сообщение M, она выбирает случайный ключ K, предназначенный для производительной криптографической системы без открытого ключа, и шифрует с его помощью сообщение M. В результате получается зашифрованное сообщение C, длина которого совпадает с длиной сообщения M. При этом ключ K довольно короткий. Затем Алиса шифрует ключ K с помощью открытого RSA-ключа Бориса. Поскольку ключ K короткий, величина $P_B(K)$ вычисляется быстро (намного быстрее, чем величина $P_B(M)$). После этого Алиса пересылает пару $(C, P_B(K))$ Борису, который расшифровывает часть $P_B(K)$, получая в результате ключ K, позволяющий расшифровать сообщение C. В результате этой процедуры Борис получит исходное сообщение M.

Аналогичный гибридный подход часто используется для эффективной генерации цифровых подписей. В этом подходе схема RSA применяется совместно с открытой ***устойчивой к коллизиям хэш-функцией*** h (collision-resistant hash function h) — функцией, которую легко вычислить, но для которой нельзя вычислительными средствами найти пару сообщений M и M', удовлетворяющих условию $h(M) = h(M')$. Величина $h(M)$ представляет собой короткий (скажем, 160-битовый) "отпечаток" сообщения M. Если Алиса захочет подписать сообщение M, то она сначала применяет функцию h к сообщению M, получив в результате отпечаток $h(M)$, который она затем шифрует с помощью своего секретного ключа. После этого она отправляет Борису пару $(M, S_A(h(M)))$ в качестве подписанной версии сообщения M. Чтобы проверить подпись, Борис может вычислить величину $h(M)$ и убедиться, что она совпадает с величиной, полученной в результате применения ключа P_A к полученной компоненте $S_A(h(M))$. Поскольку никто не может создать двух сообщений с одинаковыми отпечатками, невозможно вычислительными средствами изменить подписанное сообщение и в то же время сохранить достоверность подписи.

Наконец, заметим, что распространение открытых ключей значительно облегчается благодаря ***сертификатам*** (certificates). Например, предположим, что существует "авторитетный источник" T, чей открытый ключ широко известен. Алиса может получить от T подписанное сообщение (свой сертификат), в котором утверждается, что P_A — открытый ключ Алисы. Этот сертификат является "самоаутентифицируемым", поскольку ключ P_T известен всем. В свое подписанное сообщение Алиса может включить свой сертификат, чтобы получатель сразу

имел в распоряжении ее открытый ключ и смог проверить ее подпись. Поскольку ключ Алисы подписан источником T, получатель убеждается, что ключ Алисы действительно принадлежит ей.

Упражнения

31.7-1. Рассмотрим набор значений $p = 11$, $q = 29$, $n = 312$ и $e = 3$, образующих ключи в системе RSA. Какое значение d следует использовать в секретном ключе? Как выглядит результат шифровки сообщения $M = 100$?

31.7-2. Докажите, что если показатель степени e в открытом ключе Алисы равен 3, и если постороннему известен показатель степени d секретного ключа Алисы, где $0 < d < \phi(n)$, то он может разложить на множители входящее в состав ключа число n в течение времени, которое выражается полиномиальной функцией от количества битов в числе n. (Читателю может быть интересен тот факт (хотя в упражнении не требуется его доказать), что этот результат остается истинным даже без условия $e = 3$. Дополнительные сведения можно почерпнуть из статьи Миллера (Miller) [221].)

⋆ 31.7-3. Докажите, что схема RSA является мультипликативной в том смысле, что

$$P_A(M_1) P_A(M_2) \equiv P_A(M_1 M_2) \pmod{n}.$$

Докажите с помощью этого факта, что если злоумышленник располагает процедурой, способной эффективно расшифровать 1 процент зашифрованных с помощью ключа P_A сообщений из $\mathbf{Z}_n$, то он может воспользоваться вероятностным алгоритмом, чтобы с высокой степенью вероятности расшифровывать все сообщения, зашифрованные с помощью ключа P_A.

⋆ 31.8 Проверка простоты

В этом разделе рассматривается задача поиска больших простых чисел. Начнем с того, что обсудим плотность распределения простых чисел на числовой оси, после чего перейдем к исследованию правдоподобного (но неполного) подхода к проверке простоты чисел. Затем будет представлен эффективный рандомизированный тест простоты, предложенный Миллером (Miller) и Рабином (Rabin).

Плотность распределения простых чисел

Во многих приложениях (например, в криптографии) возникает необходимость поиска больших "случайных" простых чисел. К счастью, большие простые числа встречаются достаточно часто, поэтому для поиска простого числа путем проверки случайных целых чисел соответствующего размера потребуется не так уж много времени. ***Функция распределения простых чисел*** (prime distribution function) $\pi(n)$ определяется как количество простых чисел, не превышающих числа n. Например, $\pi(10) = 4$, поскольку количество простых чисел, не превышающих 10, равно 4 (это числа 2, 3, 5 и 7). В теореме о распределении простых чисел приводится полезное приближение функции $\pi(n)$.

Теорема 31.37 (Теорема о простых числах).

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1.$$

■

Приближенная оценка $n/\ln n$ дает достаточно точную оценку функции $\pi(n)$ даже при малых n. Например, при $n = 10^9$, когда $\pi(n) = 50\,847\,534$, а $n/\ln n \approx$ $\approx 48\,254\,942$, отклонение не превышает 6%. (Для специалиста в области теории чисел 10^9 — число небольшое.)

С помощью теоремы о простых числах вероятность того, что случайным образом выбранное число n окажется простым, можно оценить как $1/\ln n$. Таким образом, чтобы найти простое число, длина которого совпадает с длиной числа n, понадобится проверить приблизительно $\ln n$ целых чисел, выбрав их случайным образом в окрестности числа n. Например, чтобы найти 512-битовое простое число, понадобится перебрать приблизительно $\ln 2^{512} \approx 355$ случайным образом выбранных 512-битовых чисел, проверяя их простоту. (Ограничившись только нечетными числами, это количество можно уменьшить в два раза.)

В оставшейся части этого раздела рассматривается задача по определению того, является ли простым большое целое число n. Для удобства обозначений предположим, что разложение числа n на простые множители имеет вид

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}, \tag{31.37}$$

где $r \geqslant 1$, $p_1, p_2, \ldots, p_r$ — простые множители числа n, а $e_1, e_2, \ldots, e_r$ — положительные целые числа. Конечно же, число n является простым тогда и только тогда, когда $r = 1$ и $e_1 = 1$.

Одним из элементарных подходов к задаче проверки на простоту является ***пробное деление*** (trial division). При этом предпринимается попытка нацело разделить n на все целые числа $2, 3, \ldots, \lfloor\sqrt{n}\rfloor$. (Здесь также можно опустить четные числа, большие 2.) Легко понять, что число n простое тогда и только тогда, когда оно не делится ни на один из пробных множителей. Если предположить, что

для обработки каждого пробного делителя требуется фиксированное время, то в худшем случае время решения задачи при таком подходе будет равно $\Theta(\sqrt{n})$, а это означает, что оно выражается экспоненциальной функцией от длины числа n. (Напомним, что если бинарное представление числа n имеет длину β, то $\beta = \lceil \lg(n+1) \rceil$, поэтому $\sqrt{n} = \Theta\left(2^{\beta/2}\right)$.) Таким образом, пробное деление хорошо работает только при условии, что число n очень мало, или если окажется, что у него есть маленький простой делитель. Преимущество этого метода заключается в том, что он не только позволяет определить, является ли число n простым, но и находит один из его простых делителей, если оно составное.

В этом разделе нас будет интересовать только тот факт, является ли заданное число n простым; если оно составное, мы не станем искать его простые множители. Как будет показано в разделе 31.9, разложение чисел на простые множители вычислительными средствами — дорогостоящая операция. Возможно, может показаться странным, что намного легче определить, является ли заданное число n простым, чем разложить на простые множители составное число.

Проверка псевдопростых чисел

Рассмотрим "почти работающий" метод проверки простых чисел, который оказывается вполне пригодным для многих практических приложений. Позже будет представлена усовершенствованная версия этого метода, устраняющая его небольшой дефект. Обозначим через $\mathbf{Z}_n^+$ ненулевые элементы множества $\mathbf{Z}_n$:

$$\mathbf{Z}_n^+ = \{1, 2, \ldots, n-1\}.$$

Если n — простое, то $\mathbf{Z}_n^+ = \mathbf{Z}_n^*$.

Говорят, что число n ***псевдопростое по основанию*** a (base-a pseudoprime), если оно составное и

$$a^{n-1} \equiv 1 \pmod{n}. \tag{31.38}$$

Из теоремы Ферма (теорема 31.31) следует, что если n — простое, то оно удовлетворяет уравнению (31.38) для каждого элемента a множества $\mathbf{Z}_n^+$. Таким образом, если удается найти какой-нибудь элемент $a \in \mathbf{Z}_n^+$, при котором n *не* удовлетворят уравнению (31.38), то n — определенно составное. Удивительно, что *почти* справедливо обратное утверждение, поэтому этот критерий является почти идеальным тестом на простоту. Мы проверяем, удовлетворяет ли число n уравнению (31.38) при $a = 2$. Если это не так, то делается вывод, что n — составное. В противном случае можно выдвинуть гипотезу, что n — простое (в то время как фактически известно лишь то, что оно либо простое, либо псевдопростое по основанию 2).

Приведенная ниже процедура проверяет число n на простоту описанным способом. В ней используется процедура MODULAR_EXPONENTIATION, описанная в разделе 31.6. Предполагается, что входное значение n — нечетное целое число, большее 2.

```
PSEUDOPRIME(n)
1  if MODULAR_EXPONENTIATION(2, n − 1, n) ≠ 1 (mod n)
2      then return СОСТАВНОЕ                ▷ Определенно
3      else return ПРОСТОЕ                  ▷ Предположительно
```

Эта процедура может допускать ошибки, но только одного типа. Если процедура говорит, что n — составное, то это всегда верно. Если же она утверждает, что n — простое, то это заключение ошибочно только тогда, когда n — псевдопростое по основанию 2.

Как часто процедура допускает ошибки? На удивление редко. Всего имеется лишь 22 значения n, меньших 10 000, для которых ответ будет неправильным. Первые четыре таких значения равны 341, 561, 654 и 1105. Можно показать, что вероятность того, что в этой процедуре будет допущена ошибка для случайно выбранного β-битового числа, стремится к нулю при $\beta \to \infty$. Воспользовавшись результатами статьи Померанца (Pomerance) [244], содержащей более точную оценку количества псевдопростых чисел фиксированного размера по основанию 2, можно заключить, что случайным образом выбранное 512-битовое число, названное приведенной выше процедурой простым, окажется псевдопростым по основанию 2 в менее чем одном случае из 10^{20}, а случайным образом выбранное 1024-битовое число, названное простым, окажется псевдопростым по основанию 2 в менее чем одном случае из 10^{41}. Поэтому если достаточно найти большое простое число для каких-то приложений, то для всех практических целей почти никогда не возникнет ошибки, если большие числа подбираются случайным образом до тех пор, пока для одного из них процедура PSEUDOPRIME выдаст результат ПРОСТОЕ. Но если числа, которые проверяются на простоту, подбираются не случайным образом, необходим более качественный тест. Как станет ясно через некоторое время, немного сообразительности и рандомизация позволят создать программу проверки простых чисел, которая будет хорошо работать для любых входных данных.

К сожалению, путем проверки уравнения (31.38) не удается полностью исключить все ошибки, возникающие из-за псевдопростых чисел по другим основаниям, например, при $a = 3$, поскольку существуют составные числа n, удовлетворяющие уравнению (31.38) при *всех* $a \in \mathbf{Z}_n^*$. Эти числа известны как ***числа Кармайкла*** (Carmichael numbers). Первые три числа Кармайкла — 561, 1105 и 1729. Числа Кармайкла встречаются крайне редко, например, имеется всего 255 таких чисел, меньших 100 000 000. Понять, почему эти числа так редко встречаются, поможет упражнение 31.8-2.

В следующем подразделе будет показано, как улучшить тест простоты таким образом, чтобы в нем не возникали ошибки из-за чисел Кармайкла.

Рандомизированный тест простоты Миллера-Рабина

Тест простоты Миллера-Рабина позволяет решить проблемы, возникающие в простом тесте PSEUDOPRIME. Этого удается достичь за счет двух модификаций, описанных ниже.

- В этом тесте производится проверка не одного, а нескольких случайным образом выбранных значений a.
- В процессе вычисления каждой степени по модулю в тесте проверяется, обнаружен ли нетривиальный квадратный корень из 1 по модулю n в ходе последней серии возведений в квадрат. Если это так, то процедура останавливает свою работу и выдает сообщение СОСТАВНОЕ. Правильность выявления составных чисел таким способом подтверждается следствием 31.35.

Ниже приведен псевдокод теста простоты Миллера-Рабина. На его вход подается проверяемое нечетное число $n > 2$ и значение s, — количество случайным образом выбранных значений из множества $\mathbf{Z}_n^+$, относительно которых проверяется простота. В этом коде используется генератор случайных чисел RANDOM, описанный в главе 5: процедура RANDOM$(1, n-1)$ возвращает случайное целое число, удовлетворяющее неравенству $1 \leqslant a \leqslant n-1$. В коде используется вспомогательная процедура WITNESS. Процедура WITNESS(a, n) выдает значение TRUE тогда и только тогда, когда значение a "свидетельствует" о том, что число n составное, т.е. если с помощью a можно доказать (способом, который станет понятен через некоторое время), что n — составное. Тест WITNESS(a, n) — это более эффективное обобщение теста PSEUDOPRIME, основанного на проверке соотношения

$$a^{n-1} \not\equiv 1 \pmod{n}$$

при $a = 2$. Сначала представим и обоснуем схему работы процедуры WITNESS, а затем покажем, как она используется в тесте простоты Миллера-Рабина. Пусть $n-1 = 2^t u$, где $t \geqslant 1$, а u — нечетное; т.е. бинарное представление значения $n-1$ имеет вид бинарного представления нечетного числа u, после которого следует ровно t нулей. Таким образом, выполняется соотношение $a^{n-1} \equiv (a^u)^{2^t} \pmod{n}$, поэтому чтобы вычислить величину $a^{n-1} \bmod n$, можно сначала вычислить величину $a^u \bmod n$, а затем последовательно возвести ее в квадрат t раз.

```
WITNESS(a, n)
1  Пусть n − 1 = 2^t u, где t ⩾ 1 и u — нечетное
2  x_0 ← MODULAR_EXPONENTIATION(a, u, n)
3  for i ← 1 to t
4      do x_i ← x_{i−1}^2 mod n
5         if x_i = 1 и x_{i−1} ≠ 1 и x_{i−1} ≠ n − 1
6            then return TRUE
```

```
7   if x_t ≠ 1
8     then return TRUE
9   return FALSE
```

В псевдокоде процедуры WITNESS вычисляется величина $a^{n-1} \bmod n$. Для этого сначала в строке 2 вычисляется значение $x_0 = a^u \bmod n$, а затем результат последовательно t раз возводится в квадрат в цикле **for** в строках 3–6. Применив индукцию по i, можно сделать вывод, что последовательность вычисленных значений $x_0, x_1, \ldots, x_t$ удовлетворяет уравнению $x_i \equiv a^{2^i u} \pmod{n}$ при $i = 0, 1, \ldots, t$. В частности, $x_t \equiv a^{n-1} \pmod{n}$. Однако этот цикл может закончиться раньше, если после очередного возведения в квадрат в строке 4 в строках 5–6 будет обнаружен нетривиальный квадратный корень 1. В этом случае работа алгоритма завершается, и он возвращает значение TRUE. Это же значение возвращается и в строках 7–8, если значение, вычисленное из соотношения $x_t \equiv a^{n-1} \pmod{n}$, не равно 1. Это именно тот случай, когда процедура PSEUDOPRIME выдает сообщение СОСТАВНОЕ. Если в строках 6 или 8 не возвращается значение TRUE, в строке 9 возвращается значение FALSE.

Теперь покажем, что если процедура WITNESS(a, n) возвращает значение TRUE, то с помощью величины a можно доказать, что число n — составное.

Если процедура WITNESS(a, n) возвращает значение TRUE в строке 8, то она обнаружила, что справедливо соотношение $x_t = a^{n-1} \bmod n \neq 1$. Однако если число n — простое, то для всех $a \in \mathbf{Z}_n^+$ выполняется равенство $a^{n-1} \equiv 1 \pmod{n}$ согласно теореме Эйлера (теорема 31.31). Поэтому n не может быть простым, и неравенство $a^{n-1} \bmod n \neq 1$ служит доказательством этого факта.

Если процедура WITNESS(a, n) возвращает значение TRUE в строке 6, то она обнаружила, что значение x_{i-1} является нетривиальным квадратным корнем $x_i = 1$ по модулю n, поскольку выполняется соотношение $x_{i-1} \not\equiv \pm 1 \pmod{n}$, но $x_i \equiv x_{i-1}^2 \equiv 1 \pmod{n}$. В следствии 31.35 утверждается, что нетривиальный квадратный корень из 1 по модулю n может существовать лишь тогда, когда n — составное. Таким образом, тот факт, что x_{i-1} — нетривиальный квадратный корень из 1 по модулю n, доказывает, что n — составное.

На этом доказательство корректности процедуры WITNESS завершено. Если при вызове процедуры WITNESS(a, n) выдается значение TRUE, то n — гарантированно составное; доказательство этого факта легко провести, пользуясь значениями a и n.

Сейчас будет представлено краткое альтернативное описание поведения алгоритма WITNESS как функции от последовательности $X = \langle x_0, x_1, \ldots, x_t \rangle$. Это описание окажется полезным впоследствии при анализе эффективности работы проверки простоты Миллера-Рабина. Заметим, что если при некоторых значениях $0 \leqslant i < t$ выполняется равенство $x_i = 1$, то остальная часть последовательности в процедуре WITNESS может не вычисляться. В таком случае все значения

$x_{i+1}, x_{i+2}, \ldots, x_t$ равны 1, и мы считаем, что в последовательности X на этих позициях находятся единицы. Имеем четыре различных случая.

1. $X = \langle \ldots, d \rangle$, где $d \neq 1$: последовательность X не заканчивается единицей. В этом случае возвращается значение TRUE; о том, что n — составное, свидетельствует значение a (согласно теореме Ферма).
2. $X = \langle 1, 1, \ldots, 1 \rangle$: последовательность X состоит из одних единиц. В этом случае возвращается значение FALSE; значение a не свидетельствует о том, что n — составное.
3. $X = \langle \ldots, -1, 1, \ldots, 1 \rangle$: последовательность X оканчивается единицей, и последнее отличное от единицы значение равно -1. В этом случае возвращается значение FALSE; значение a не свидетельствует о том, что n — составное.
4. $X = \langle \ldots, d, 1, \ldots, 1 \rangle$, где $d \neq \pm 1$: последовательность X оканчивается единицей, но последнее отличное от единицы значение не равно -1. В этом случае возвращается значение TRUE; значение a свидетельствует о том, что n — составное, поскольку d — нетривиальный квадратный корень 1.

Теперь рассмотрим тест Миллера-Рабина на простоту, основанный на применении процедуры WITNESS. Как и раньше, предполагается, что n — нечетное целое число, большее 2.

```
MILLER_RABIN(n, s)
1  for j ← 1 to s
2        do a ← RANDOM(1, n − 1)
3           if WITNESS(a, n)
4              then return СОСТАВНОЕ        ▷ Наверняка.
5  return ПРОСТОЕ                          ▷ Почти наверняка.
```

В процедуре MILLER_RABIN реализован вероятностный поиск доказательства того факта, что число n — составное. В основном цикле (который начинается в строке 1) выбирается s случайных значений величины a из множества $\mathbf{Z}_n^+$ (строка 2). Если одно из этих значений свидетельствует о том, что число n — составное, то процедура MILLER_RABIN в строке 4 выдает значение СОСТАВНОЕ. Такой вывод всегда верный, согласно корректности процедуры WITNESS для этого случая. Если в ходе s попыток не было таких свидетельств, то в процедуре MILLER_RABIN предполагается, что нет причин считать число n составным, так что оно считается простым. Скоро можно будет убедиться, что при достаточно больших значениях s этот вывод правильный с высокой вероятностью, но тем не менее существует небольшая вероятность, что в процедуре могут неудачно выбираться значения a, и что существуют свидетельства того, что n — составное, хотя ни одно из них не было найдено.

Чтобы проиллюстрировать работу процедуры MILLER_RABIN, предположим, что n равно числу Кармайкла 561, так что $n - 1 = 560 = 2^4 \cdot 35$. Из табл. 31.4

видно, что если выбрано значение $a = 7$, то в процедуре WITNESS будет найдено значение $x_0 = a^{35} \equiv 241 \pmod{561}$, что даст нам последовательность $X = \langle 241, 289, 166, 67, 1 \rangle$. Таким образом, при последнем возведении в квадрат обнаружен нетривиальный корень из 1, поскольку $a^{280} \equiv 67 \pmod{n}$ и $a^{560} \equiv 1 \pmod{n}$. Таким образом, значение $a = 7$ свидетельствует о том, что число n — составное, процедура WITNESS$(7, n)$ возвращает значение TRUE, а процедура MILLER_RABIN — значение СОСТАВНОЕ.

Если длина числа n равна β битов, то для выполнения процедуры MILLER_RABIN требуется $O(s\,\beta)$ арифметических операций и $O\left(s\,\beta^3\right)$ битовых операций, поскольку в асимптотическом пределе требуется выполнять не более s возведений в степень по модулю.

Частота ошибок в тесте Миллера-Рабина

Если процедура MILLER_RABIN выводит значение ПРОСТОЕ, то с небольшой вероятностью в ней может быть допущена ошибка. Однако, в отличие от процедуры PSEUDOPRIME, эта вероятность не зависит от n; другими словами, для этой процедуры не существует неблагоприятных входных данных. Зато вероятность ошибки в процедуре MILLER_RABIN зависит от s и от того, насколько удачно выбирались значения a. Кроме того, поскольку каждая проверка является более строгой, чем обычная проверка уравнения (31.38); исходя из общих принципов можно ожидать, что для случайно выбранных целых значений n частота ошибок должна быть очень небольшой. Более точное обоснование представлено в сформулированной ниже теореме.

Теорема 31.38. Если n — нечетное составное число, то количество свидетельств того, что n — составное, не меньше $(n-1)/2$.

Доказательство. В ходе доказательства теоремы будет показано, что количество значений, не являющихся свидетельствами, не превышает $(n-1)/2$, из чего следует справедливость теоремы.

Начнем с утверждения, что любое значение, которое не является свидетельством, должно быть элементом множества $\mathbf{Z}_n^*$. Почему? Рассмотрим такое значение a, не являющееся свидетельством. Оно должно удовлетворять соотношению $a^{n-1} \equiv 1 \pmod{n}$ или эквивалентному соотношению $a \cdot a^{n-2} \equiv 1 \pmod{n}$. Таким образом, a^{n-2} является решением уравнения $ax \equiv 1 \pmod{n}$. Согласно следствию 31.21, $\gcd(a, n) \mid 1$, из чего в свою очередь следует, что $\gcd(a, n) = 1$. Поэтому a является элементом множества $\mathbf{Z}_n^*$; этому множеству принадлежат все значения оснований, не свидетельствующие о том, что число n — составное.

Чтобы завершить доказательство, покажем, что все значения оснований, не свидетельствующие о том, что число n — составное, не просто содержатся в множестве $\mathbf{Z}_n^*$, но и находятся в истинной подгруппе B группы $\mathbf{Z}_n^*$. (Напомним, что B

называется *истинной* подгруппой группы $\mathbf{Z}_n^*$, если она является подгруппой $\mathbf{Z}_n^*$, но не равна $\mathbf{Z}_n^*$.) Согласно свойству 31.16, выполняется неравенство $|B| \leqslant |\mathbf{Z}_n^*|/2$. Поскольку $|\mathbf{Z}_n^*| \leqslant n-1$, мы получаем неравенство $|B| \leqslant (n-1)/2$. Поэтому количество значений оснований, не являющихся свидетельствами, не превышает $(n-1)/2$, следовательно, количество свидетельств не меньше $(n-1)/2$.

Теперь покажем, как найти истинную подгруппу B группы $\mathbf{Z}_n^*$, которая содержит все значения оснований, не являющиеся свидетельствами. Выделим два случая.

Случай 1: существует значение $x \in \mathbf{Z}_n^*$, такое что $x^{n-1} \not\equiv 1 \pmod{n}$.

Другими словами, n не является числом Кармайкла. Поскольку, как мы уже знаем, числа Кармайкла встречаются крайне редко, то случай 1 — основной случай, который встречается "на практике" (т.е. тогда, когда число n выбрано случайным образом и производится проверка его простоты).

Пусть $B = \left\{b \in \mathbf{Z}_n^* : b^{n-1} \equiv 1 \pmod{n}\right\}$. Очевидно, что множество B непустое, так как $1 \in B$. Поскольку группа B замкнута относительно операции умножения по модулю n, то из теоремы 31.14 следует, что B — подгруппа группы $\mathbf{Z}_n^*$. Заметим, что каждое значение a, которое не является свидетельством, принадлежит множеству B, поскольку такое a удовлетворяет соотношению $a^{n-1} \equiv 1 \pmod{n}$. Поскольку $x \in \mathbf{Z}_n^* - B$, то B — истинная подгруппа группы $\mathbf{Z}_n^*$.

Случай 2: для каждого $x \in \mathbf{Z}_n^*$ выполняется соотношение

$$x^{n-1} \equiv 1 \pmod{n} \tag{31.39}$$

Другими словами, n — число Кармайкла. На практике этот случай встречается крайне редко. Однако тест MILLER_RABIN (в отличие от теста на псевдопростоту), как сейчас будет показано, в состоянии эффективно определить, что число Кармайкла — составное.

В этом случае число n не может быть степенью простого числа. Чтобы понять, почему это так, предположим обратное, т.е. что $n = p^e$, где p — простое, а $e > 1$. Противоречие мы получим следующим образом. Поскольку предполагается, что n — нечетное, то число p также должно быть нечетным. Из теоремы 31.32 следует, что $\mathbf{Z}_n^*$ — циклическая группа: она содержит генератор g, такой что

$$\operatorname{ord}_n(g) = |\mathbf{Z}_n^*| = \phi(n) = p^e(1 - 1/p) = (p-1)p^{e-1}.$$

Согласно уравнению (31.39), $g^{n-1} \equiv 1 \pmod{n}$. Тогда из теоремы о дискретном логарифме (теорема 31.33 при $y = 0$) следует, что $n - 1 \equiv 0 \pmod{\phi(n)}$ или

$$(p-1)p^{e-1} \mid p^e - 1.$$

При $e > 1$ мы получаем противоречие, поскольку $(p-1)p^{e-1}$ делится на простое число p, а $p^e - 1$ — нет. Таким образом, число n не является степенью простого числа.

Поскольку нечетное составное число n не является степенью простого, оно раскладывается на множители $n_1 n_2$, где n_1 и n_2 — взаимно простые нечетные числа, большие 1. (Таких разложений может быть несколько; в этом случае не играет роли, какое из них выбирается. Например, если $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, то можно выбрать $n_1 = p_1^{e_1}$ и $n_2 = p_2^{e_2} p_3^{e_3} \cdots p_r^{e_r}$.)

Напомним, что значения t и u связаны соотношением $n - 1 = 2^t u$, где $t \geqslant 1$, а u — нечетно, и что в процедуре WITNESS для входного значения a вычисляется последовательность

$$X = \left\langle a^u, a^{2u}, a^{2^2 u}, \ldots, a^{2^t u} \right\rangle$$

(все вычисления производятся по модулю n).

Назовем пару (v, j) целых чисел ***приемлемой*** (acceptable), если $v \in \mathbf{Z}_n^*$, $j \in \{0, 1, \ldots, t\}$ и

$$v^{2^j u} \equiv -1 \,(\mathrm{mod} n)\,.$$

Для нечетных значений u приемлемые пары точно существуют; если выбрать $v = n - 1$ и $j = 0$, то такая пара будет приемлемой. Теперь выберем наибольшее из возможных значений j, для которого существует приемлемая пара (v, j), и зафиксируем значение v. Пусть

$$B = \left\{ x \in \mathbf{Z}_n^* : x^{2^j u} \equiv \pm 1 \,(\mathrm{mod} n) \right\}.$$

Поскольку множество B замкнуто относительно операции умножения по модулю n, то оно является подгруппой группы $\mathbf{Z}_n^*$. Поэтому, согласно следствию 31.16, $|B|$ является делителем $|\mathbf{Z}_n^*|$. Каждое значение, которое не является свидетельством, должно быть элементом множества B, поскольку последовательность X, образованная такими значениями, должна либо полностью состоять из единиц, либо содержать -1 в позиции, расположенной не позже j-й, в соответствии с условием максимальности значения j. (Если пара (a, j') приемлемая, а значение a не является свидетельством, то из способа выбора значения j должно следовать неравенство $j' \leqslant j$.)

Теперь воспользуемся фактом существования значения v, чтобы продемонстрировать, что существует элемент $w \in \mathbf{Z}_n^* - B$. Поскольку $v^{2^j u} \equiv -1 \,(\mathrm{mod} n)$, из следствия 31.29 из китайской теоремы об остатках вытекает, что $v^{2^j u} \equiv \equiv -1 \,(\mathrm{mod} n_1)$. Согласно следствию 31.28, существует значение w, которое одновременно удовлетворяет таким уравнениям:

$$w \equiv v \,(\mathrm{mod} n_1)\,,$$
$$w \equiv 1 \,(\mathrm{mod} n_2)\,.$$

Поэтому

$$w^{2^j u} \equiv -1 \,(\mathrm{mod} n_1)\,,$$
$$w^{2^j u} \equiv 1 \,(\mathrm{mod} n_2)\,.$$

Согласно следствию 31.29, из соотношения $w^{2^j u} \not\equiv 1 \,(\mathrm{mod} n_1)$ следует, что $w^{2^j u} \not\equiv \not\equiv 1 \,(\mathrm{mod} n)$, а из соотношения $w^{2^j u} \not\equiv -1 \,(\mathrm{mod} n_2)$ — что $w^{2^j u} \not\equiv -1 \,(\mathrm{mod} n)$. Таким образом, $w^{2^j u} \not\equiv \pm 1 \,(\mathrm{mod} n)$, поэтому $w \notin B$.

Остается показать, что $w \in \mathbf{Z}_n^*$. Для этого построим рассуждения отдельно по модулю n_1 и по модулю n_2. Что касается операций по модулю n_1, заметим, что поскольку $v \in \mathbf{Z}_n^*$, справедливо равенство $\gcd(v, n) = 1$, так что и $\gcd(v, n_1) = 1$; если у числа v нет общих делителей с n, то у него также нет общих делителей с n_1. Поскольку $w \equiv v \,(\mathrm{mod} n_1)$, можно сделать вывод, что $\gcd(w, n_1) = 1$. Что же касается операций по модулю n_2, заметим, что из соотношения $w \equiv 1 \,(\mathrm{mod} n_2)$ следует, что $\gcd(w, n_2) = 1$. Для того чтобы объединить эти результаты, воспользуемся теоремой 31.6, из которой следует, что $\gcd(w, n_1 n_2) = \gcd(w, n) = 1$, т.е. $w \in \mathbf{Z}_n^*$.

Следовательно, $w \in \mathbf{Z}_n^* - B$, и мы приходим к выводу, что в случае 2 B является истинной подгруппой $\mathbf{Z}_n^*$.

Итак, мы убедились, что в обоих случаях количество свидетельств того, что число n — составное, не меньше $(n-1)/2$. ■

Теорема 31.39. При любом нечетном $n > 2$ и положительном целом s вероятность того, что процедура MILLER_RABIN(n, s) выдаст неправильный результат, не превышает 2^{-s}.

Доказательство. Из теоремы 31.38 следует, что если n — составное, то при каждом выполнении цикла **for** в строках 1–4, вероятность обнаружить свидетельство x того, что n — составное, не меньше $1/2$. Процедура MILLER_RABIN допускает ошибку только в том случае, когда ей не удалось обнаружить такое свидетельство в каждой из s итераций основного цикла. Вероятность подобной последовательности неудач не превышает 2^{-s}. ■

Таким образом, если выбрать $s = 50$, то этого должно хватить почти для любого приложения, какое только можно себе представить. Если поиск больших простых чисел производится путем применения процедуры MILLER_RABIN к *случайно выбранным* большим целым числам, то можно показать (хотя мы не станем здесь этого делать), что выбор небольшого значения s (скажем, 3) с очень малой вероятностью приведет к ошибочным результатам. Другими словами, для случайным образом выбранного нечетного составного целого числа n математическое ожидание количества значений оснований, не являющихся свидетельствами того,

что n — составное, с большой вероятностью намного меньше $(n-1)/2$. Однако если число n выбирается не случайным образом, самое лучшее, что можно доказать с помощью улучшенной версии теоремы 31.38, — что количество значений оснований, не являющихся свидетельствами, не превышает $(n-1)/4$. Более того, существуют такие целые числа n, для которых это количество равно $(n-1)/4$.

Упражнения

31.8-1. Докажите, что если целое нечетное число $n > 1$ не является простым числом или степенью простого числа, то существует нетривиальный квадратный корень из 1 по модулю n.

⋆ 31.8-2. Теорему Эйлера можно слегка усилить, придав ей такой вид:

$$a^{\lambda(n)} \equiv 1 \pmod{n} \quad \text{для всех } a \in \mathbf{Z}_n^*,$$

где $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, а функция $\lambda(n)$ определена как

$$\lambda(n) = \operatorname{lcm}\left(\phi\left(p_1^{e_1}\right), \ldots, \phi\left(p_r^{e_r}\right)\right). \tag{31.40}$$

Докажите, что $\lambda(n) \mid \phi(n)$. Составное число n является числом Кармайкла, если $\lambda(n) \mid n-1$. Наименьшее из чисел Кармайкла равно $561 = 3 \cdot 11 \cdot 17$; при этом $\lambda(n) = \operatorname{lcm}(2, 10, 16) = 80$, а это делитель 560. Докажите, что числа Кармайкла должны быть "свободными от квадратов" (т.е. не делиться на квадрат ни одного простого числа) и в то же время представлять собой произведение не менее трех простых чисел. По этой причине они встречаются не очень часто.

31.8-3. Докажите, что если x — нетривиальный квадратный корень из единицы по модулю n, то и $\gcd(x-1, n)$, и $\gcd(x+1, n)$ являются нетривиальными делителями n.

⋆ 31.9 Целочисленное разложение

Предположим, задано целое число n, которое нужно ***разложить*** (factor) на простые множители. Тест простоты, представленный в предыдущем разделе, может дать информацию о том, что число n — составное, однако он обычно не выводит его простых множителей. Разложение больших целых чисел n представляется намного более сложной задачей, чем определение того, является ли число n простым или составным. Располагая суперкомпьютерами и наилучшими на сегодняшний день алгоритмами, нереально разложить произвольное 1024-битовое число.

Эвристический ρ-метод Полларда

Пробное деление на каждое целое число вплоть до B гарантирует, что будет полностью разложено любое число вплоть до B^2. Представленная ниже процедура позволяет разложить любое число вплоть до B^4 (если мы не окажемся неудачниками), выполнив тот же объем работы. Поскольку эта процедура носит лишь эвристический характер, ничего нельзя утверждать наверняка ни о времени ее работы, ни о том, что она действительно достигнет успеха. Несмотря на это, данная процедура оказывается очень эффективной на практике. Другое преимущество процедуры POLLARD_RHO состоит в том, что в ней используется лишь фиксированное количество памяти. (Для небольших чисел ее легко реализовать даже на программируемом калькуляторе.)

```
POLLARD_RHO(n)
 1  i ← 1
 2  x_1 ← RANDOM(0, n − 1)
 3  y ← x_1
 4  k ← 2
 5  while TRUE
 6      do i ← i + 1
 7         x_i ← (x_{i−1}^2 − 1) mod n
 8         d ← gcd(y − x_i, n)
 9         if d ≠ 1 и d ≠ n
10           then print d
11         if i = k
12           then y ← x_i
13                k ← 2k
```

Опишем работу этой процедуры. В строках 1-2 переменной i присваивается начальное значение 1, а переменной x_1 — случайное значение из $\mathbf{Z}_n$. Итерации цикла **while**, который начинается в строке 5, продолжаются до бесконечности в поисках множителей числа n. При каждой итерации этого цикла в строке 7 используется рекуррентное соотношение

$$x_i \leftarrow \left(x_{i-1}^2 - 1\right) \bmod n, \tag{31.41}$$

позволяющее получить очередное значение x_i бесконечной последовательности

$$x_1, x_2, x_3, \ldots, \tag{31.42}$$

а соответствующее значение индекса i увеличивается в строке 6. Несмотря на то, что для простоты восприятия в коде используются значения переменных с индексами x_i, программа работает точно так же, если мы опустим все индексы,

поскольку при каждой итерации необходимо поддерживать только одно последнее значение x_i. С учетом этой модификации в процедуре используется лишь фиксированное количество ячеек памяти.

Время от времени программа сохраняет самые последние из сгенерированных значений x_i в переменной y. Сохраняются те значения, индексы которых равны степени двойки:

$$x_1, x_2, x_4, x_8, x_{16}, \ldots .$$

В строке 3 сохраняется величина x_1, а в строке 12 — величины x_k, когда i становится равным k. В строке 4 переменной k присваивается начальное значение 2, а строке 13 эта переменная удваивается при каждом обновлении значения переменной y. Поэтому переменная k пробегает значения 1,2,4,8,..., каждое из которых используется в качестве индекса очередной величины x_k, сохраняемой в переменной y.

В строках 8–10 предпринимается попытка разложить число n с помощью сохраненного значения y и текущего значения x_i. В частности, в строке 8 вычисляется наибольший общий делитель $d = \gcd(y - x_i, n)$. Если значение переменной d — нетривиальный делитель числа n (это проверяется в строке 9), то оно выводится в строке 10.

Возможно, эта процедура поиска делителей на первый взгляд может показаться несколько загадочной. Однако заметим, что она никогда не выдает неверного ответа; все числа, которые выводит процедура POLLARD_RHO, являются нетривиальными делителями n. Тем не менее, эта процедура может вообще не выводить никаких данных; совершенно нет уверенности в том, что она даст хоть какой-то результат. Тем не менее, как мы сможем убедиться, есть веская причина ожидать, что процедура POLLARD_RHO выведет множитель p числа n после $\Theta\left(\sqrt{p}\right)$ итераций цикла **while**. Таким образом, если n — составное число, то эта процедура после приблизительно $n^{1/4}$ обновлений обнаружит достаточное количество делителей, чтобы можно было полностью разложить число n, поскольку все простые множители p числа n, кроме, возможно, последнего, меньше $\sqrt{n}$.

Начнем анализ поведения представленной выше процедуры с того, что исследуем, сколько времени должно пройти, пока в случайной последовательности по модулю n не повторится значение. Поскольку множество $\mathbf{Z}_n$ конечное, и поскольку каждое значение последовательности (31.42) зависит только от предыдущего, то эта последовательность в конце концов начнет повторяться. Достигнув некоторого значения x_i, такого что при некотором $j < i$ выполняется равенство $x_i = x_j$, последовательность зациклится, поскольку $x_{i+1} = x_{j+1}$, $x_{i+2} = x_{j+2}$ и т.д. Понять, почему этот метод был назван "эвристическим ρ-методом", помогает рис. 31.3. Часть последовательности $x_1, x_2, \ldots, x_{j-1}$ можно изобразить в виде "хвоста" греческой буквы ρ, а цикл $x_j, x_{j+1}, \ldots, x_i$ — в виде "тела" этой буквы. На рис. 31.3*a* приведены значения, полученные из рекуррентного соотношения

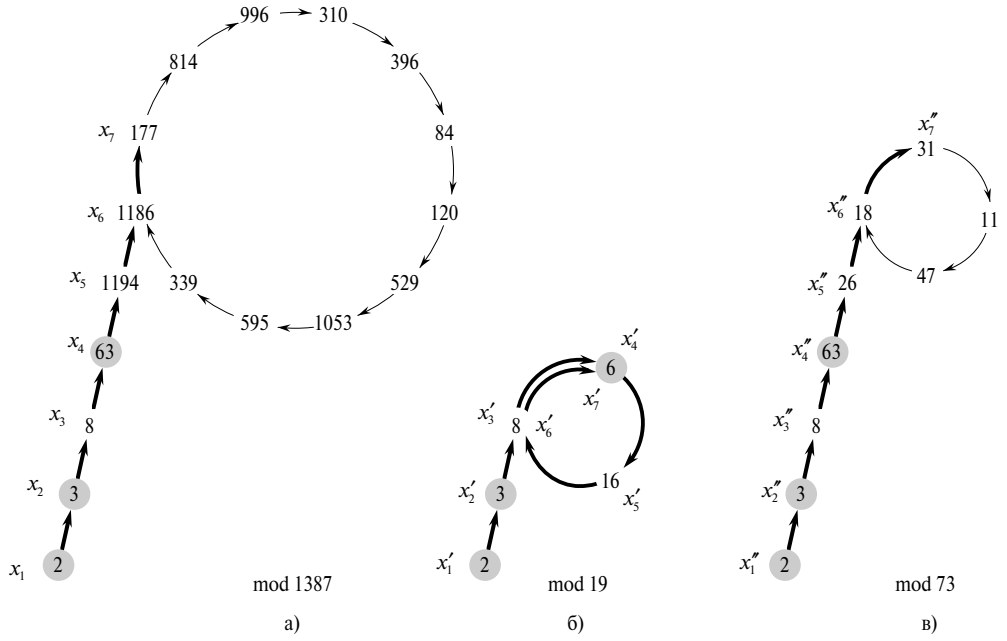


Рис. 31.3. Эвристический ρ-метод Полларда

$x_{i+1} \leftarrow \left(x_i^2 - 1\right) \bmod 1387$, начиная с $x_1 = 2$. Разложение на простые множители числа 1387 имеет вид $19 \cdot 73$. Жирными стрелками показаны шаги итерации, которые выполняются перед обнаружением множителя 19. Стрелки, изображенные тонкими линиями, указывают на те значения в итерации, которые так и не достигаются. С их помощью иллюстрируется форма буквы ρ. Серым фоном выделены те значения, которые сохраняются процедурой POLLARD_RHO в переменной y. Множитель 19 обнаруживается при достижении значения $x_7 = 177$, когда вычисляется величина $\gcd(63 - 177, 1387) = 19$. Первое значение x, которое впоследствии повторится, равно 1186, однако множитель 19 обнаруживается еще до повтора. На рис. 31.3*б* приведены значения, полученные из того же рекуррентного соотношения, что и значения на рис. 31.3*а*, но по модулю 19. Каждое значение x_i из рис. 31.3*а* эквивалентно приведенному здесь значению x'_i по модулю 19. Например, значения $x_4 = 63$ и $x_7 = 177$ эквивалентны 6 по модулю 19. Значения, приведенные на рис. 31.3*в*, получены из того же рекуррентного соотношения, но по модулю 73. Каждое значение x_i из рис. 31.3*а* эквивалентно приведенному в этой части значению x''_i по модулю 73. Согласно китайской теореме об остатках, каждый узел на рис. 31.3*а* соответствует паре узлов, — одному на рис. 31.3*б*, и второму — на рис. 31.3*в*.

Рассмотрим вопрос о том, сколько времени понадобится последовательности значений x_i, чтобы она начала повторяться. Это не совсем то, что нам нужно, но вскоре станет понятно, как модифицировать этот параметр.

Чтобы оценить это время, будем считать, что функция

$$f_n(x) = \left(x^2 - 1\right) \bmod n$$

ведет себя как "случайная" функция. Конечно, на самом деле она не случайная, но это предположение согласуется с наблюдаемым поведением процедуры POLLARD_RHO. Далее, предположим, что каждое значение x_i извлекается независимым образом из множества $\mathbf{Z}_n$, и что все они распределены в этом множестве равномерно. В соответствии с анализом парадокса о днях рождения, проведенном в разделе 5.4.1, математическое ожидание количества шагов, предпринятых перед зацикливанием последовательности, равно $\Theta(\sqrt{n})$.

Теперь перейдем к требуемой модификации. Пусть p — нетривиальный множитель числа n, для которого справедливо уравнение $\gcd(p, n/p) = 1$. Например, если разложение числа n имеет вид $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, то в качестве множителя p можно выбрать величину $p_1^{e_1}$. (Если $e_1 = 1$, то толь p будет играть наименьший простой множитель числа n; это полезно взять на заметку.)

Последовательность $\langle x_i \rangle$ порождает соответствующую последовательность $\langle x'_i \rangle$ по модулю p, где $x'_i = x_i \bmod p$ для всех i.

Кроме того, поскольку в определении функции f_n содержатся только арифметические операции (возведение в квадрат и вычитание) по модулю n, нетрудно показать, что величину x'_{i+1} можно вычислить на основании величины x'_i. Рассмотрение последовательности "по модулю p" — уменьшенная версия того, что происходит по модулю n:

$$\begin{aligned}
x'_{i+1} &= x_{i+1} \bmod p = \\
&= f_n(x_i) \bmod p = \\
&= \left(\left(x_i^2 - 1\right) \bmod n\right) \bmod p = \\
&= \left(x_i^2 - 1\right) \bmod p = \\
&= \left((x_i \bmod p)^2 - 1\right) \bmod p = \\
&= \left(\left(x'_i\right)^2 - 1\right) \bmod p = \\
&= f_p\left(x'_i\right),
\end{aligned}$$

где четвертое равенство следует из упражнения 31.1-6. Таким образом, хотя мы и не вычислили явным образом последовательность $\langle x'_i \rangle$, эта последовательность вполне определена и подчиняется тому же рекуррентному соотношению, что и последовательность $\langle x_i \rangle$.

Рассуждая так, как и ранее, можно прийти к выводу, что математическое ожидание количества шагов, выполненных перед зацикливанием последовательности $\langle x'_i \rangle$, равно $\Theta\left(\sqrt{p}\right)$. Если значение p мало по сравнению с n, то последовательность $\langle x'_i \rangle$ может начать повторяться намного быстрее, чем последовательность $\langle x_i \rangle$. В самом деле, последовательность $\langle x'_i \rangle$ начнет повторяться, как только два элемента последовательности $\langle x_i \rangle$ окажутся эквивалентными по модулю p, а не по модулю n. Вышесказанное проиллюстрировано на рис. 31.3*б* и *в*.

Обозначим через t индекс первого повторившегося значения последовательности $\langle x'_i \rangle$, а через $u > 0$ — длину цикла, полученного таким образом. Другими словами, t и $u > 0$ — наименьшие значения, для которых при всех $i \geqslant 0$ выполняется равенство $x'_{t+i} = x'_{t+u+i}$. Согласно приведенным выше рассуждениям, математическое ожидание величин t и u равно $\Theta\left(\sqrt{p}\right)$. Заметим, что если $x'_{t+i} = x'_{t+u+i}$, то $p \mid (x_{t+u+i} - x_{t+i})$. Таким образом, $\gcd\left(x_{t+u+i} - x_{t+i}, n\right) > 1$.

Поэтому после того как в процедуре POLLARD_RHO в переменной y будет сохранено любое значение x_k, у которого $k \geqslant t$, величина $y \bmod p$ всегда будет находиться в цикле по модулю p. (Если в переменной y будет сохранено новое значение, оно также окажется в цикле по модулю p.) В конце концов переменной k будет присвоено значение, превышающее u, после чего в процедуре будет пройден полный цикл по модулю p, не сопровождающийся изменением значения y. Множитель числа n будет обнаружен тогда, когда x_i "натолкнется" на ранее сохраненное значение y по модулю p, т.е. когда $x_i \equiv y \,(\mathrm{mod} p)$.

Скорее всего, найденный множитель будет равен p, хотя это случайно может оказаться число, кратное p. Поскольку математическое ожидание величин t и u равно $\Theta\left(\sqrt{p}\right)$, математическое ожидание количества шагов, необходимых для получения множителя p, равно $\Theta\left(\sqrt{p}\right)$.

Есть две причины, по которым этот алгоритм может оказаться не таким эффективным, как ожидается. Во-первых, проведенный выше эвристический анализ времени работы нестрогий, и цикл значений по модулю p может оказаться намного длиннее, чем $\sqrt{p}$. В этом случае алгоритм работает правильно, но намного медленнее, чем хотелось бы. Практически же это представляется сомнительным. Во-вторых, среди делителей числа n, которые выдаются этим алгоритмом, всегда может оказаться один из тривиальных множителей 1 или n. Например, предположим, что $n = pq$, где p и q — простые числа. Может оказаться, что значения t и u для множителя p идентичны значениям t и u для множителя q, и поэтому множитель p будет всегда обнаруживаться в результате выполнения той же операции $\gcd$, в которой обнаруживается множитель q. Поскольку оба множителя находятся одновременно, процедура выдает тривиальный множитель $pq = n$, который бесполезен. На практике и эта проблема кажется несущественной. При необходимости нашу эвристическую процедуру можно перезапустить с другим рекуррентным соотношением, которое имеет вид $x_{i+1} \leftarrow \left(x_i^2 - c\right) (\mathrm{mod} n)$. (Значений параметра

c, равных 0 и 2, следует избегать по причинам, вникать в которые мы здесь не станем; другие значения вполне подходят.)

Конечно же, этот анализ нестрогий и эвристический, поскольку рекуррентное соотношение на самом деле не является "случайным". Тем не менее, на практике процедура работает хорошо, и, по-видимому, ее эффективность совпадает с полученной в ходе эвристического анализа. Она представляет собой вероятностный метод поиска небольших простых множителей, на которые раскладывается большое число. Все, что нужно для полного разложения β-битового составного числа n, — найти все его простые множители, меньшие $\lfloor n^{1/2} \rfloor$, поэтому можно ожидать, что процедуре POLLARD_RHO понадобится не более $n^{1/4} = 2^{\beta/4}$ арифметических операций и не более $n^{1/4}\beta^2 = 2^{\beta/4}\beta^2$ битовых операций. Зачастую наиболее привлекательной особенностью этой процедуры оказывается ее способность находить небольшой множитель p числа n, выполнив при этом операции, математическое ожидание количества которых равно $\Theta\left(\sqrt{p}\right)$.

Упражнения

31.9-1. Когда процедура POLLARD_RHO выведет множитель 73 числа 1387, если история вычислений в ней имеет вид, показанный на рис. 31.3*a*?

31.9-2. Пусть задана функция $f : \mathbf{Z}_n \to \mathbf{Z}_n$ и начальное значение $x_0 \in \mathbf{Z}_n$. Определим рекуррентное соотношение $x_i = f(x_{i-1})$ для $i = 1, 2, \ldots$. Пусть t и $u > 0$ — наименьшие значения, для которых выполняется уравнение $x_{t+i} = x_{t+u+i}$ для $i = 1, 2, \ldots$ (в терминах ρ-алгоритма Полларда, t — это длина хвоста, а u — длина цикла ρ). Сформулируйте эффективный алгоритм, позволяющий точно определить значения t и u, и проанализируйте время его работы.

31.9-3. Чему равно математическое ожидание количества шагов, которые понадобится выполнить процедуре POLLARD_RHO, чтобы обнаружить множитель вида p^e, где p — простое число, а $e > 1$?

⋆ 31.9-4. Как уже упоминалось, одним из недостатков процедуры POLLARD_RHO является то, что на каждом шаге рекурсии в ней необходимо выполнять операцию gcd. Было высказано предположение, что вычисления gcd можно объединить, накопив произведение нескольких величин x_i и используя это произведение при вычислении $\gcd$ вместо величины x_i. Приведите подробное описание того, как можно было бы реализовать эту идею, почему она работает и какой размер группы можно было бы выбрать для достижения максимального эффекта при обработке β-битового числа n.

Задачи

31-1. **Бинарный алгоритм gcd**

На большинстве компьютеров операции вычитания, проверки четности (нечетное или четное) и деления пополам выполняется быстрее, чем вычисление остатка. В этой задаче исследуется ***бинарный алгоритм gcd*** (binary gcd algorithm), позволяющий избежать вычисления остатков, которые используются в алгоритме Евклида.

а) Докажите, что если числа a и b — четные, то справедливо уравнение $\gcd(a, b) = 2\gcd(a/2, b/2)$.

б) Докажите, что если a — нечетное, а b — четное, то справедливо уравнение $\gcd(a, b) = \gcd(a, b/2)$.

в) Докажите, что если числа a и b — нечетные, то справедливо уравнение $\gcd(a, b) = \gcd((a - b)/2, b)$.

г) Разработайте эффективный бинарный алгоритм поиска $\gcd$ для целых чисел a и b, где $a \geqslant b$, время работы которого равно $O(\lg a)$. Предполагается, что на каждое вычитание, проверку на четность и деление пополам затрачивается единичный интервал времени.

31-2. **Анализ битовых операций в алгоритме Евклида**

а) Рассмотрим обычный алгоритм деления "на бумаге" a на b, в результате которого получается частное q и остаток r. Покажите, что в этом методе требуется выполнить $O((1 + \lg q)\lg b)$ битовых операций.

б) Определим функцию $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. Покажите, что количество битовых операций, которые выполняются в алгоритме EUCLID при сведении задачи вычисления величины $\gcd(a, b)$ к вычислению величины $\gcd(b, a \bmod b)$, не превышает $c(\mu(a, b) - -\mu(b, a \bmod b))$ для некоторой достаточно большой константы $c > 0$.

в) Покажите, что выполнение алгоритма EUCLID(a, b) требует в общем случае $O(\mu(a, b))$ битовых операций, и $O(\beta^2)$ битовых операций, если оба входных значения являются β-битовыми числами.

31-3. **Три алгоритма вычисления чисел Фибоначчи**

В этой задаче производится сравнение производительности работы трех методов вычисления n-го числа Фибоначчи F_n при заданном n. Считаем, что стоимость сложения, вычитания или умножения двух чисел независимо от их размера равна $O(1)$.

а) Покажите, что время работы прямого рекурсивного метода вычисления числа F_n на основе рекуррентного соотношения (3.21) увеличивается экспоненциально с ростом n.

б) Покажите, как с помощью запоминания вычислить число F_n за время $O(n)$.

в) Покажите, как вычислить число F_n за время $O(\lg n)$, используя только сложения и умножения целых чисел. (*Указание*: рассмотрите матрицу

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

и ее степени.)

г) Теперь предположим, что для сложения двух β-битовых чисел требуется время $\Theta(\beta)$, а для их умножения — время $\Theta(\beta^2)$. Чему равно время работы перечисленных выше алгоритмов с учетом такой стоимости выполнения элементарных арифметических операций?

31-4. **Квадратичные вычеты**

Пусть p — нечетное целое число. Число $a \in \mathbf{Z}_p^*$ является ***квадратичным вычетом*** (quadratic residue), если уравнение $x^2 = a \pmod{p}$ имеет решение относительно неизвестного x.

а) Покажите, что существует ровно $(p-1)/2$ квадратичных вычета по модулю p.

б) Если число p — простое, то определим ***символ Лежандра*** (Legendre symbol) $\left(\frac{a}{p}\right)$ для $a \in \mathbf{Z}_p^*$, как равный 1, если a — квадратичный вычет по модулю p, и -1 в противном случае. Докажите, что если $a \in \mathbf{Z}_p^*$, то

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}.$$

Разработайте эффективный алгоритм, позволяющий определить, является ли заданное число a квадратичным вычетом по модулю p. Проанализируйте время работы этого алгоритма.

в) Докажите, что если p — простое число вида $4k+3$, а a — квадратичный вычет в $\mathbf{Z}_p^*$, то величина $a^{k+1} \bmod p$ равна квадратному корню a по модулю p. Сколько времени понадобится для поиска квадратного корня квадратичного вычета a по модулю p?

г) Разработайте эффективный рандомизированный алгоритм поиска неквадратичного вычета по модулю, равному произвольному простому числу p. Другими словами, нужно найти элемент $\mathbf{Z}_p^*$, который

не является квадратичным вычетом. Сколько в среднем операций понадобится выполнить этому алгоритму?

Заключительные замечания

Книга Нивена (Niven) и Цукермана (Zuckerman) [231] содержит прекрасное введение в элементарную теорию чисел. У Кнута (Knuth) [183] приведено подробное обсуждение алгоритмов, предназначенных для поиска наибольших общих делителей, а также других основных теоретико-числовых алгоритмов. Бэтч (Batch) [28] и Ризель (Riesel) [258] представили более современный обзор вычислительных приложений теории чисел. В статье Диксона (Dixon) [78] приведен обзор методов разложения и проверки простоты чисел. В трудах конференции под редакцией Померанца (Pomerance) [245] содержится несколько превосходных обзорных статей. Позже Бэтч и Шаллит (Shallit) [29] представили прекрасный обзор основных вычислительных приложений теории чисел.

В книге Кнута [183] обсуждается история возникновения алгоритма Евклида. Он встречается в трактате "Начала" греческого математика Евклида, опубликованном в 300 году до н.э., в седьмой книге в теоремах 1 и 2. Описанный Евклидом алгоритм мог быть получен из алгоритма, предложенного Евдоксом около 375 года до н.э. Не исключено, что алгоритм Евклида является старейшим нетривиальным алгоритмом; соперничать с ним может только алгоритм умножения, известный еще древним египтянам. В статье Шаллита [274] описана история анализа алгоритма Евклида.

Авторство частного случая китайской теоремы об остатках (теорема 31.27) Кнут приписывает китайскому математику Сунь-Цзы, который жил приблизительно между 200 г. до н.э. и 200 г. н.э., — дата довольно неопределенная. Тот же частный случай сформулирован греческим математиком Никомахусом (Nichomachus) около 100 г. н.э. В 1247 году он был обобщен Чином Чиу-Шао (Chhin Chiu-Shao). Наконец, в 1734 году Л. Эйлер (L. Euler) сформулировал и доказал китайскую теорему об остатках в общем виде.

Представленный в этой книге рандомизированный алгоритм проверки простоты чисел взят из статей Миллера (Miller) [221] и Рабина (Rabin) [254]; это самый быстрый (с точностью до постоянного множителя) из известных рандомизированных алгоритмов проверки простоты. Доказательство теоремы 31.39 слегка адаптировано по сравнению с тем, что было предложено Монье (Monier) [224, 225]. Рандомизация представляется необходимой для того, чтобы получить алгоритм проверки простоты, время работы которого выражалось бы полиномиальной функцией. Самый быстрый из известных детерминистических алгоритмов этого вида — версия Кохена-Ленстры (Cohen-Lenstra) [65] теста на простоту, предложенного Адлеманом (Adleman), Померанцом и Раминли (Rumenly) [3]. Проверяя простоту

числа n длиной $\lceil \lg (n+1) \rceil$, он затрачивает время $(\lg n)^{O(\lg \lg \lg n)}$, лишь немного превышающее полиномиальное.

Обстоятельное обсуждение задачи поиска больших "случайных" простых чисел содержится в статье Бьючимина (Beauchemin), Брассарда (Brassard), Крипо (Crepeau), Готье (Goutier) и Померанца [33].

Концепция криптографической системы с открытым ключом сформулирована Диффи (Diffie) и Хеллманом (Hellman) [74]. Криптографическая система RSA была предложена в 1977 году Ривестом (Rivest), Шамиром (Shamir) и Адлеманом (Adleman) [259]. С тех пор в области криптографии были достигнуты большие успехи. Углубилось понимание криптографической системы RSA, и в ее современных реализациях представленные здесь основные методы существенно улучшены. Кроме того, разработаны многие новые методы доказательства безопасности криптографических систем. Например, Голдвассер (Goldwasser) и Микали (Micali) [123] показали, что рандомизация может выступать в роли эффективного инструмента при разработке безопасных схем кодирования с открытым ключом. Гольдвассер, Микали и Ривест [124] представили схему цифровой подписи, для которой доказана трудность ее подделки. В книге Менезеса (Menezes) и др. [220] представлен обзор прикладной криптографии.

Эвристический ρ-подход, предназначенный для разложения целых чисел на множители, был изобретен Поллардом (Pollard) [242]. Представленный в этой книге вариант является версией, предложенной Брентом (Brent) [48].

Время работы наилучшего из алгоритмов, предназначенных для разложения больших чисел, приближенно выражается экспоненциальной функцией от кубического корня из n, где n — длина подлежащего разложению числа. Общий теоретико-числовой алгоритм разложения по принципу решета, разработанный Бахлером (Buhler) и др. [51] как обобщение идей, заложенных в теоретико-числовой алгоритм разложения по принципу решета Полларда [243] и Ленстры и др. [201] и усовершенствованный Копперсмитом (Coppersmith) [69], — по-видимому, наиболее эффективный среди подобных алгоритмов для обработки больших входных чисел. Несмотря на то, что строгий анализ этого алгоритма провести сложно, при разумных предположениях можно оценить время его работы как $L(1/3, n)^{1.902+o(1)}$, где $L(\alpha, n) = e^{(\ln n)^{\alpha}(\ln \ln n)^{1-\alpha}}$.

Метод эллиптических кривых, предложенный Ленстрой (Lenstra) [202], может оказаться эффективнее для некоторых входных данных, чем теоретико-числовой метод решета, поскольку, как и ρ-метод Полларда, он достаточно быстро позволяет найти небольшой простой множитель p. Время его работы при поиске множителя p оценивается как $L(1/2, p)^{\sqrt{2}+o(1)}$.

ГЛАВА 32

Поиск подстрок

В программах, предназначенных для редактирования текста, часто возникает задача поиска всех фрагментов текста, которые совпадают с заданным образцом. Обычно текст — это редактируемый документ, а образец — искомое слово, введенное пользователем. Эффективные алгоритмы решения этой задачи могут повышать способность текстовых редакторов к реагированию. Кроме того, алгоритмы поиска подстрок используются, например, для поиска заданных образцов в молекулах ДНК.

Приведем формальную постановку ***задачи поиска подстрок*** (string-matching problem). Предполагается, что текст задан в виде массива $T[1..n]$ длины n, а образец — в виде массива $P[1..m]$ длины $m \leqslant n$. Далее, предполагается, что элементы массивов P и T — символы из конечного алфавита Σ. Например, алфавит может иметь вид $\Sigma = \{0, 1\}$ или $\Sigma = \{a, b, \ldots, z\}$. Символы массивов P и T часто называют ***строками*** (strings) или символами.

Говорят, что образец P ***встречается*** в тексте T ***со сдвигом*** s (occurs with shift s), если $0 \leqslant s \leqslant n - m$ и $T[s + 1..s + m] = P[1..m]$ (другими словами, если при $1 \leqslant j \leqslant m$ $T[s + j] = P[j]$). (Можно также сказать, что образец P ***встречается*** в тексте T, ***начиная с позиции*** $s + 1$.) Если образец P встречается в тексте T со сдвигом s, то величину s называют ***допустимым сдвигом*** (valid shift); в противном случае ее называют ***недопустимым сдвигом*** (invalid shift). Задача поиска подстрок — это задача поиска всех допустимых сдвигов, с которыми заданный образец P встречается в тексте T. Эти определения проиллюстрированы на рис. 32.1. В представленном на этом рисунке примере предлагается найти все вхождения образца $P = abaa$ в текст $T = abcabaabcabac$. Образец встречается в тексте только один раз, со сдвигом $s = 3$. Говорят, что сдвиг $s = 3$

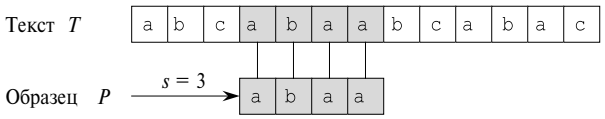


Рис. 32.1. Задача поиска подстрок

является допустимым. Каждый символ образца соединен вертикальной линией с соответствующим символом в тексте, и все совпадающие символы выделены серым фоном.

Все представленные в этой главе алгоритмы, кроме описанного в разделе 32.1 обычного алгоритма решения задачи "в лоб", производят определенную предварительную обработку представленного образца, а затем находят все допустимые сдвиги; последняя фаза будет называться поиском. В табл. 32.1 приведены времена предварительной обработки и сравнения для каждого из представленных в этой главе алгоритмов. Полное время работы каждого алгоритма равно сумме времени предварительной обработки и времени сравнения. В разделе 32.2 описан интересный алгоритм поиска подстрок, предложенный Рабином (Rabin) и Карпом (Karp). Несмотря на то, что время работы этого алгоритма в наихудшем случае (равное $\Theta((n-m+1)m)$) не лучше, чем время работы простейшего метода "в лоб", на практике в среднем он работает намного лучше. Он также легко обобщается на случаи других подобных задач. Затем в разделе 32.3 описывается алгоритм поиска подстрок, работа которого начинается с конструирования конечного автомата, специально предназначенного для поиска совпадений заданного образца P с фрагментами текста. Время предварительной обработки в этом алгоритме равно $O(m|\Sigma|)$, зато время сравнения в нем равно всего лишь $\Theta(n)$. Аналогичный, но более совершенный алгоритм Кнута-Морриса-Пратта (Knuth-Morris-Pratt, или KMP) представлен в разделе 32.4; время сравнения в этом алгоритме остается тем же, т.е. оно равно $\Theta(n)$, но время предварительной обработки в нем уменьшается до величины $\Theta(m)$.

Таблица 32.1. Алгоритмы поиска подстрок и время их предварительной обработки и сравнения

Алгоритм	Время предварительной обработки	Время сравнения
Простейший	0	$O((n-m+1)m)$
Рабина-Карпа	$\Theta(m)$	$O((n-m+1)m)$
Конечный автомат	$\Theta(m\|\Sigma\|)$	$\Theta(n)$
Кнута-Морриса-Пратта	$\Theta(m)$	$\Theta(n)$

Обозначения и терминология

Обозначим через Σ^* множество всех строк конечной длины, образованных с помощью символов алфавита Σ. В этой главе рассматриваются только строки конечной длины. ***Пустая строка*** (empty string) конечной длины, которая обозначается ε, также принадлежит множеству Σ^*. Длина строки x обозначается $|x|$. ***Конкатенация*** (concatenation) двух строк x и y, которая обозначается xy, имеет длину $|x| + |y|$ и состоит из символов строки x, после которых следуют символы строки y.

Говорят, что строка w — ***префикс*** (prefix) строки x (обозначается $w \sqsubset x$), если существует такая строка $y \in \Sigma^*$, что $x = wy$. Заметим, что если $w \sqsubset x$, то $|w| \leqslant |x|$. Аналогично, строку w называют ***суффиксом*** (suffix) строки x (обозначается как $w \sqsupset x$), если существует такая строка $y \in \Sigma^*$, что $x = yw$. Из соотношения $w \sqsupset x$ также следует неравенство $|w| \leqslant |x|$. Пустая строка ε является одновременно и суффиксом, и префиксом любой строки. В качестве примеров префикса и суффикса можно привести $ab \sqsubset abcca$ и $cca \sqsupset abcca$. Следует иметь в виду, что для произвольных строк x и y и для любого символа a соотношение $x \sqsupset y$ выполняется тогда и только тогда, когда $xa \sqsupset ya$. Кроме того, заметим, что отношения $\sqsubset$ и $\sqsupset$ являются транзитивными. Впоследствии окажется полезной сформулированная ниже лемма.

Лемма 32.1 (Лемма о перекрывающихся суффиксах). Предположим, что x, y и z — строки, для которых выполняются соотношения $x \sqsupset z$ и $y \sqsupset z$. Если $|x| \leqslant |y|$, то $x \sqsupset y$. Если $|x| \geqslant |y|$, то $y \sqsupset x$. Если $|x| = |y|$, то $x = y$.

Доказательство. Графическое доказательство представлено на рис. 32.2. На каждой из трех частей рисунка проиллюстрированы три случая леммы. Вертикальными линиями соединены совпадающие области строк (выделенные серым фоном). В части *а* показан случай, когда $|x| \leqslant |y|$; при этом $x \sqsupset y$. В части *б* показан случай, когда $|x| \geqslant |y|$, и $y \sqsupset x$. Из части *в* видно, что если $|x| = |y|$, то $x = y$. ■

Обозначим для краткости k-символьный префикс $P[1..k]$ образца $P[1..m]$ через P_k. Таким образом, $P_0 = \varepsilon$ и $P_m = P = P[1..m]$. Аналогично, k-символьный префикс текста T обозначим T_k. С помощью этих обозначений задачу ***поиска подстрок*** можно сформулировать как задачу о выявлении всех сдвигов s в интервале $0 \leqslant s \leqslant n - m$, таких что $P \sqsupset T_{s+m}$.

В псевдокоде предполагается, что сравнение двух строк одинаковой длины является примитивной операцией. Если строки сравниваются слева направо, и процесс сравнения прерывается, когда обнаружено несовпадение, то считается, что время, которое требуется для подобного теста, выражается линейной функцией от количества совпавших символов. Точнее говоря, считается, что тест "$x = y$" выполняется за время $\Theta(t + 1)$, где t — длина самой длинной строки z, такой

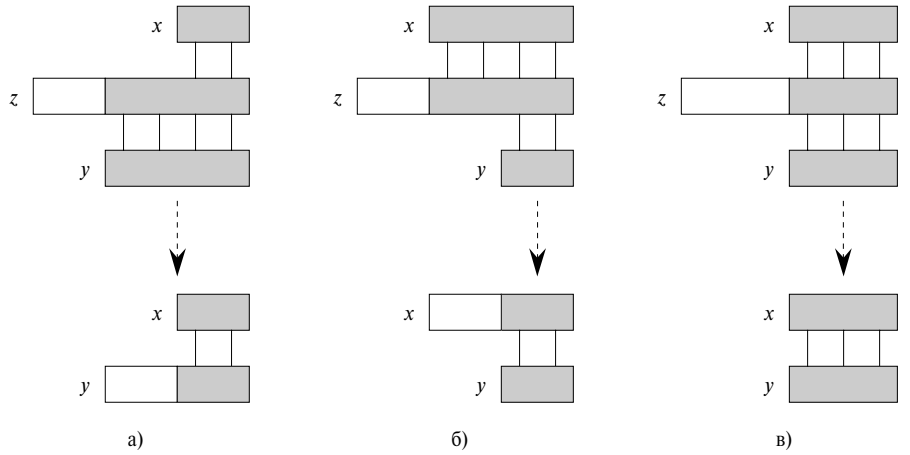


Рис. 32.2. Графическое доказательство леммы 32.1

что $z \sqsubset x$ и $z \sqsubset y$. (Чтобы учесть случай, когда $t = 0$, вместо $\Theta(t)$ мы пишем $\Theta(t+1)$. В этой ситуации не совпадает первый же сравниваемый символ, но для проверки этого требуется некоторое положительное время.)

32.1 Простейший алгоритм поиска подстрок

В простейшем алгоритме поиск всех допустимых сдвигов производится с помощью цикла, в котором проверяется условие $P[1..m] = T[s+1..s+m]$ для каждого из $n - m + 1$ возможных значений s.

NAIVE_STRING_MATCHER(T, P)
1 $n \leftarrow length[T]$
2 $m \leftarrow length[P]$
3 **for** $s \leftarrow 0$ **to** $n - m$
4 **do if** $P[1\,..\,m] = T[s+1\,..\,s+m]$
5 **then** print "Образец обнаружен при сдвиге" s

Простейшую процедуру поиска подстрок можно интерпретировать графически как скольжение "шаблона" с образцом по тексту, в процессе которого отмечается, для каких сдвигов все символы шаблона равны соответствующим символам текста. В частях *а–г* рис. 32.3, который иллюстрирует эту интерпретацию, показаны четыре последовательных положения, проверяемых в простейшем алгоритме поиска подстрок. В каждой части рисунка вертикальные линии соединяют соответствующие области, для которых обнаружено совпадение (эти области выделены серым цветом), а ломаные линии — первые не совпавшие символы (если такие имеются). В части *в* показано одно обнаруженное вхождение образца со сдвигом

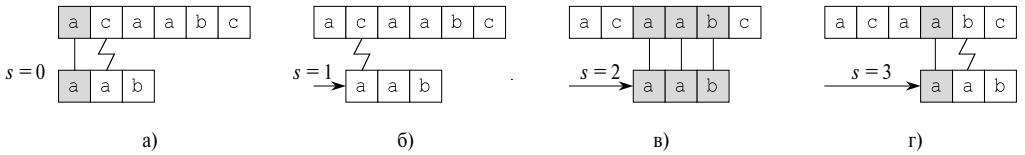


Рис. 32.3. Принцип работы простейшего алгоритма поиска подстрок для образца $P = aab$ и текста $T = acaabc$

$s = 2$. В цикле **for**, который начинается в строке 3, явным образом рассматриваются все возможные сдвиги. В строке 4 проверяется, действителен ли текущий сдвиг; в этот тест неявно включен цикл для проверки символов, которые находятся в соответствующих позициях, до тех пор, пока не совпадут все символы или не будет обнаружено несовпадение. В строке 5 выводится каждый допустимый сдвиг s.

Время работы процедуры NAIVE_STRING_MATCHER равно $O\left((n-m+1)\,m\right)$, и эта оценка достаточно точна в наихудшем случае. Например, рассмотрим текстовую строку a^n (строку, состоящую из n символов a) и образец a^m. Для каждого из $n-m+1$ возможных значений сдвига s неявный цикл в строке 4 для проверки совпадения соответствующих символов должен произвести m итераций, чтобы подтвердить допустимость сдвига. Таким образом, время работы в наихудшем случае равно $\Theta\left((n-m+1)\,m\right)$, так что в случае $m = \lfloor n/2 \rfloor$ оно становится равным $\Theta\left(n^2\right)$. Время работы процедуры NAIVE_STRING_MATCHER равно времени сравнения, так как фаза предварительной обработки отсутствует.

Вскоре вы увидите, что процедура NAIVE_STRING_MATCHER не является оптимальной для этой задачи. В этой главе будет приведен алгоритм, время предварительной обработки в котором в наихудшем случае равно $\Theta\left(m\right)$, а время сравнения в наихудшем случае — $\Theta\left(n\right)$. Простейший алгоритм поиска подстрок оказывается неэффективным, поскольку информация о тексте, полученная для одного значения s, полностью игнорируется при рассмотрении других значений s. Однако эта информация может стать очень полезной. Например, если образец имеет вид $P = aaab$, а значение одного из допустимых сдвигов равно нулю, то ни один из сдвигов, равных 1, 2 или 3, не могут быть допустимыми, поскольку $T\left[4\right] = b$. В последующих разделах исследуется несколько способов эффективного использования информации такого рода.

Упражнения

32.1-1. Покажите, какие сравнения производятся простейшим алгоритмом поиска подстрок, если образец имеет вид $P = 0001$, а текст — $T = 000010001010001$.

32.1-2. Предположим, что в образце P все символы различны. Покажите, как ускорить процедуру NAIVE_STRING_MATCHER, чтобы время ее выполнения при обработке n-символьного текста T было равно $O(n)$.

32.1-3. Предположим, что образец P и текст T — строки длиной m и n соответственно, *случайно* выбранные из d-символьного алфавита $\Sigma_d = \{0, 1, \ldots, d-1\}$, где $d \geqslant 2$. Покажите, что *математическое ожидание* количества сравнений одного символа с другим, производимых в неявном цикле в строке 4 простейшего алгоритма, во всех итерациях этого цикла равно

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}} \leqslant 2(n-m+1).$$

(Предполагается, что в простейшем алгоритме сравнение символов для данного сдвига прекращается, как только будет обнаружено несовпадение или совпадет весь образец.) Таким образом, для случайных строк простейший алгоритм вполне эффективен.

32.1-4. Предположим, в образце P может содержаться ***символ пробела*** (gap character) $\diamondsuit$, который может соответствовать произвольной строке (даже нулевой длины). Например, образец $ab\diamondsuit ba\diamondsuit c$ встречается в тексте $cabccbacbacab$ как

$$c\underbrace{ab}_{ab}\underbrace{cc}_{\diamondsuit}\underbrace{ba}_{ba}\underbrace{cba}_{\diamondsuit}\underbrace{c}_{c}ab$$

и как

$$c\underbrace{ab}_{ab}\underbrace{ccbac}_{\diamondsuit}\underbrace{ba}_{ba}\underbrace{}_{\diamondsuit}\underbrace{c}_{c}ab.$$

Заметим, что в образце символ пробела может встречаться произвольное количество раз, но предполагается, что в тексте он не встречается вообще. Сформулируйте алгоритм с полиномиальным временем работы, определяющий, встречается ли заданный образец P в тексте T, и проведите анализ времени его работы.

32.2 Алгоритм Рабина-Карпа

Рабин (Rabin) и Карп (Karp) предложили алгоритм поиска подстрок, показывающий на практике хорошую производительность, а также допускающий обобщения на другие родственные задачи, такие как задача о сопоставлении двумерного образца. В алгоритме Рабина-Карпа время $\Theta(m)$ затрачивается на предварительную обработку, а время его работы в наихудшем случае равно $\Theta((n-m+1)m)$. Однако с учетом определенных предположений удается показать, что среднее время работы этого алгоритма оказывается существенно лучше.

В этом алгоритме используются обозначения из элементарной теории чисел, такие как эквивалентность двух чисел по модулю третьего числа. Соответствующие определения можно найти в разделе 31.1.

Для простоты предположим, что $\Sigma = \{0, 1, \ldots, 9\}$, т.е. каждый символ — это десятичная цифра. (В общем случае можно предположить, что каждый символ — это цифра в системе счисления с основанием d, где $d = |\Sigma|$.) После этого строку из k последовательных символов можно рассматривать как число длиной k. Таким образом, символьная строка 31415 соответствует числу 31415. При такой двойной интерпретации входных символов и как графических знаков, и как десятичных чисел, в этом разделе удобнее подразумевать под ними цифры, входящие в стандартный текстовый шрифт.

Для заданного образца $P[1..m]$ обозначим через p соответствующее ему десятичное значение. Аналогично, для заданного текста $T[1..n]$ обозначим через t_s десятичное значение подстроки $T[s+1..s+m]$ длиной m при $s = 0, 1, \ldots, n - m$. Очевидно, что $t_s = p$ тогда и только тогда, когда $T[s+1..s+m] = P[1..m]$; таким образом, s — допустимый сдвиг тогда и только тогда, когда $t_s = p$. Если бы значение p можно было вычислить за время $\Theta(m)$, а все значения t_s — за суммарное время $\Theta(n-m+1)$[1], то значения всех допустимых сдвигов можно было бы определить за время $\Theta(m) + \Theta(n-m+1) = \Theta(n)$ путем сравнения значения p с каждым из значений t_s. (Пока что мы не станем беспокоиться по поводу того, что величины p и t_s могут оказаться очень большими числами.)

С помощью правила Горнера (см. раздел 30.1) величину p можно вычислить за время $\Theta(m)$:

$$p = P[m] + 10\left(P[m-1] + 10\left(P[m-2] + \cdots + 10\left(P[2] + 10P[1]\right)\cdots\right)\right).$$

Значение t_0 можно вычислить из массива $T[1..m]$ за время $\Theta(m)$ аналогичным способом. Чтобы вычислить остальные значения $t_1, t_2, \ldots, t_{n-m}$ за время $\Theta(n-m)$, достаточно заметить, что величину t_{s+1} можно вычислить из величины t_s за фиксированное время, так как

$$t_{s+1} = 10\left(t_s - 10^{m-1}T[s+1]\right) + T[s+m+1]. \tag{32.1}$$

Например, если $m = 5$ и $t_s = 31415$, то нужно удалить цифру в старшем разряде $T[s+1] = 3$ и добавить новую цифру в младший разряд (предположим, это цифра $T[s+5+1] = 2$). В результате мы получаем $t_{s+1} = 10(31415 - 10000 \cdot 3) + 2 = 14152$.

[1]Мы используем запись $\Theta(n-m+1)$ *вместо* $\Theta(n-m)$, поскольку всего имеется $n-m+1$ различных значений, которые может принимать величина s. Слагаемое "+1" важно в асимптотическом смысле, поскольку при $m = n$ для вычисления единственного значения t_s требуется время $\Theta(1)$, а не $\Theta(0)$.

Чтобы удалить из числа t_s цифру старшего разряда, из него вычитается значение $10^{m-1}T[s+1]$; путем умножения на 10 число сдвигается на одну позицию влево, а в результате добавления элемента $T[s+m+1]$ в его младшем разряде появляется нужная цифра. Если предварительно вычислить константу 10^{m-1} (с помощью методов, описанных в разделе 31.6, это можно сделать в течение времени $O(\lg m)$, хотя для данного приложения вполне подойдет обычный метод, требующий времени $O(m)$), то для каждого вычисления результатов выражения (32.1) потребуется фиксированное количество арифметических операций. Таким образом, число p можно вычислить за время $\Theta(m)$, величины $t_0, t_1, \ldots, t_{n-m}$ — за время $\Theta(n-m+1)$, а все вхождения образца $P[1..m]$ в текст $T[1..n]$ можно найти, затратив на фазу предварительной обработки время $\Theta(m)$, а на фазу сравнения — время $\Theta(n-m+1)$.

Единственная сложность, возникающая в этой процедуре, может быть связана с тем, что значения p и t_s могут оказаться слишком большими и с ними будет неудобно работать. Если образец P содержит m символов, то предположение о том, что каждая арифметическая операция с числом p (в которое входит m цифр) занимает "фиксированное время", не отвечает действительности. К счастью, эта проблема имеет простое решение (рис. 32.4): вычислять значения p и t_s по модулю некоторого числа q. Поскольку вычисление величин p, t_0 и рекуррентного соотношения (32.1) можно производить по модулю q, получается, что величина p по модулю q вычисляется за время $\Theta(m)$, а вычисление всех величин t_s по модулю q — за время $\Theta(n-m+1)$. В качестве модуля q обычно выбирается такое простое число, для которого длина величины $10q$ не превышает длины компьютерного слова. Это позволяет производить все необходимые вычисления с помощью арифметических операций с одинарной точностью. В общем случае, если имеется d-символьный алфавит $\{0, 1, \ldots, d-1\}$, значение q выбирается таким образом, чтобы длина величины dq не превышала длины компьютерного слова, и чтобы с рекуррентным соотношением (32.1) было удобно работать по модулю q. После этого рассматриваемое соотношение принимает вид

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q, \qquad (32.2)$$

где $h \equiv d^{m-1} (\mathrm{mod} q)$ —значение, которое приобретает цифра "1", помещенная в старший разряд m-цифрового текстового окна. Работа алгоритма Рабина-Карпа проиллюстрирована на рис. 32.4. Каждый символ здесь представлен десятичной цифрой, а вычисления производятся по модулю 13. В части *а* приведена строка текста. Подстрока длиной 5 символов выделена серым цветом. Находится численное значение выделенной подстроки по модулю 13, в результате чего получается значение 7. В части *б* изображена та же текстовая строка, в которой для всех возможных 5-символьных подстрок вычислены соответствующие им значения по модулю 13. Если предположить, что образец имеет вид $P = 31415$, то нужно найти подстроки, которым соответствуют значения 7 по модулю 13, поскольку

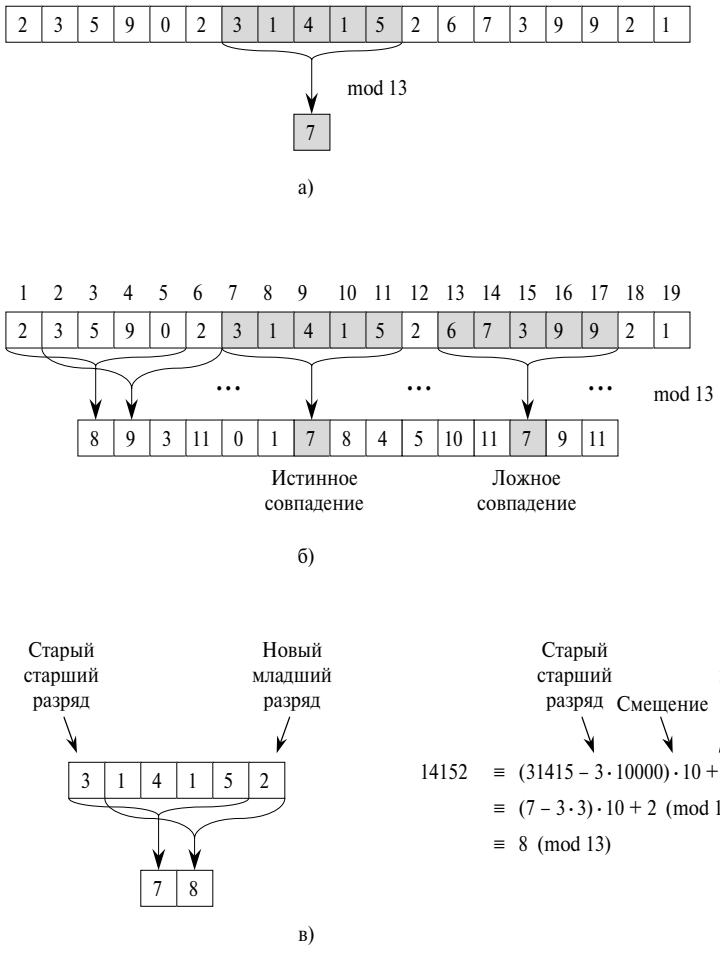


Рис. 32.4. Алгоритм Рабина-Карпа

$31415 \equiv 7 \pmod{13}$. Найдены две такие подстроки; они выделены серым цветом. Первая подстрока, которая начинается в тексте на позиции 7, действительно совпадает с образцом, в то время как вторая, которая начинается на позиции 13, представляет собой т.н. ложное совпадение. В части *в* показано, как в течение фиксированного времени вычислить значение, соответствующее данной подстроке, по значению предыдущей подстроки. Значение, соответствующее первой подстроке, равно 31415. Если отбросить цифру 3 в старшем разряде, произвести сдвиг числа влево (умножение на 10), а затем добавить к полученному результату цифру 2 в младшем разряде, то получим новое значение 14152. Однако все вычисления производятся по модулю 13, поэтому для первой подстроки получится значение 7, а для второй — значение 8.

Итак, как видим, идея работы по модулю q не лишена недостатков, поскольку из $t_s \equiv p \,(\mathrm{mod} q)$ не следует, что $t_s = p$. С другой стороны, если $t_s \not\equiv p \,(\mathrm{mod} q)$, то обязательно выполняется соотношение $t_s \neq p$ и можно сделать вывод, что сдвиг s недопустимый. Таким образом, соотношение $t_s \equiv p \,(\mathrm{mod} q)$ можно использовать в качестве быстрого эвристического теста, позволяющего исключить недопустимые сдвиги s. Все сдвиги, для которых справедливо соотношение $t_s \equiv p \,(\mathrm{mod} q)$, необходимо подвергнуть дополнительному тестированию, чтобы проверить, что действительно ли сдвиг s является допустимым, или это просто ***ложное совпадение*** (spurious hit). Такое тестирование можно осуществить путем явной проверки условия $P\,[1..m] = T\,[s+1..s+m]$. Если значение q достаточно большое, то можно надеяться, что ложные совпадения встречаются довольно редко и стоимость дополнительной проверки окажется низкой.

Сформулированная ниже процедура поясняет описанные выше идеи. В роли входных значений для нее выступает текст T, образец P, разряд d (в качестве значения которого обычно выбирается $|\Sigma|$) и простое число q.

```
RABIN_KARP_MATCHER(T, P, d, q)
 1  n ← length[T]
 2  m ← length[P]
 3  h ← d^(m−1) mod q
 4  p ← 0
 5  t_0 ← 0
 6  for i ← 1 to m                    ▷ Предварительная обработка
 7       do p ← (dp + P[i]) mod q
 8          t_0 ← (dt_0 + T[i]) mod q
 9  for s ← 0 to n − m                ▷ Проверка
10       do if p = t_s
11             then if P[1..m] = T[s+1..s+m]
12                       then print "Образец обнаружен при сдвиге" s
13          if s < n − m
14             then t_{s+1} ← (d(t_s − T[s+1]h) + T[s+m+1]) mod q
```

Опишем работу процедуры RABIN_KARP_MATCHER. Все символы интерпретируются как цифры в системе счисления по основанию d. Индексы переменной t приведены для ясности; программа будет правильно работать и без них. В строке 3 переменной h присваивается начальное значение, равное цифре, расположенной в старшем разряде m-цифрового текстового окна. В строках 4–8 вычисляется значение p, равное $P\,[1..m] \bmod q$, и значение t_0, равное $T\,[1..m] \bmod q$. В цикле **for** в строках 9–14 производятся итерации по всем возможным сдвигам s. При этом сохраняется сформулированный ниже инвариант.

При каждом выполнении строки 10 справедливо соотношение $t_s = T[s+1\,..\,s+m] \bmod q$.

Если в строке 10 выполняется условие $p = t_s$ (“совпадение”), то в строке 11 проверяется справедливость равенства $P\,[1..m] = T\,[s+1..s+m]$, чтобы исключить ложные совпадения. Все обнаруженные допустимые сдвиги выводятся в строке 12. Если $s < n - m$ (это неравенство проверяется в строке 13), то цикл **for** нужно будет выполнить хотя бы еще один раз, поэтому сначала выполняется строка 14, чтобы гарантировать соблюдение инварианта цикла, когда мы снова перейдем к строке 10. В строке 14 на основании значения $t_s \bmod q$ с использованием уравнения (32.2) в течение фиксированного интервала времени вычисляется величина $t_{s+1} \bmod q$.

В процедуре RABIN_KARP_MATCHER на предварительную обработку затрачивается время $\Theta(m)$, а время сравнения в нем в наихудшем случае равно $\Theta((n-m+1)\,m)$, поскольку в алгоритме Рабина-Карпа (как и в простейшем алгоритме поиска подстрок) явно проверяется допустимость каждого сдвига. Если $P = a^m$ и $T = a^n$, то проверка займет время $\Theta((n-m+1)\,m)$, поскольку все $n - m + 1$ возможных сдвигов являются допустимыми.

Во многих приложениях ожидается небольшое количество допустимых сдвигов (возможно, выражающееся некоторой константой c); в таких приложениях математическое ожидание времени работы алгоритма равно сумме величины $O((n-m+1)+cm) = O(n+m)$ и времени, необходимого для обработки ложных совпадений. В основу эвристического анализа можно положить предположение, что приведение значений по модулю q действует как случайное отображение множества Σ^* на множество $\mathbf{Z}_q$. (См. в разделе 11.3.1 обсуждение вопроса об использовании операции деления для хеширования. Сделанное предположение трудно формализовать и доказать, хотя один из многообещающих подходов заключается в предположении о случайном выборе числа q среди целых чисел подходящего размера. В этой книге такая формализация не применяется.) В таком случае можно ожидать, что число ложных совпадений равно $O(n/q)$, потому что вероятность того, что произвольное число t_s будет эквивалентно p по модулю q, можно оценить как $1/q$. Поскольку имеется всего $O(n)$ позиций, в которых проверка в строке 10 дает отрицательный результат, а на обработку каждого совпадения затрачивается время $O(m)$, математическое ожидание времени сравнения в алгоритме Рабина-Карпа равно

$$O(n) + O(m\,(v + n/q))\,,$$

где v — количество допустимых сдвигов. Если $v = O(1)$, а q выбрано так, что $q \geqslant m$, то приведенное выше время выполнения равно $O(n)$. Другими словами, если математическое ожидание количества допустимых сдвигов мало ($O(1)$), а выбранное простое число q превышает длину образца, то можно ожидать,

что для выполнения фазы сравнения процедуре Рабина-Карпа потребуется время $O(n+m)$. Поскольку $m \leqslant n$, то математическое ожидание времени сравнения равно $O(n)$.

Упражнения

32.2-1. Сколько ложных совпадений произойдет в процедуре Рабина-Карпа в случае текста $T = 3141592653589793$, образца поиска $P = 26$, а в качестве модуля q выбрано значение 11?

32.2-2. Как можно было бы обобщить метод Рабина-Карпа для задачи поиска в текстовой строке одного из k заданных образцов? Начните с предположения, что все k образцов имеют одинаковую длину. Затем обобщите решение таким образом, чтобы в нем учитывалась возможность того, что образцы могут быть разной длины.

32.2-3. Покажите, как обобщить метод Рабина-Крапа, чтобы он позволял решать задачу поиска заданного образца размерами $m \times m$ в символьном массиве размером $n \times n$. (Образец можно сдвигать по вертикали и по горизонтали, но его нельзя вращать.)

32.2-4. Алиса располагает копией длинного n-битового файла $A = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$, а Борис — копией n-битового файла $B = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. Алиса и Борис захотели узнать, идентичны ли их файлы. Чтобы не передавать весь файл A или файл B, они используют описанную ниже быструю вероятностную проверку. Совместными усилиями они выбирают простое число $q > 1000n$, а затем случайным образом выбирают целое число x из множества $\{0, 1, \ldots, q-1\}$. После этого Алиса вычисляет значение

$$A(x) = \left(\sum_{i=0}^{n-1} a_i x^i \right) \bmod q,$$

а Борис — соответствующее значение $B(x)$. Докажите, что если $A \neq B$, то имеется не более одного шанса из 1000, что $A(x) = B(x)$, а если файлы одинаковы, то величины $A(x)$ и $B(x)$ также обязательно совпадут. (*Указание*: см. упражнение 31.4-4.)

32.3 Поиск подстрок с помощью конечных автоматов

Многие алгоритмы поиска подстрок начинают с того, что строят конечный автомат, который сканирует строку текста T, отыскивая все вхождения в нее образца P. В этом разделе представлен метод построения такого автомата. Подобные

автоматы для поиска подстрок очень эффективны: они проверяют каждый символ текста *ровно по одному разу*, затрачивая на каждый символ фиксированное количество времени. Поэтому после предварительной обработки образца для построения автомата, время, необходимое для поиска, равно $\Theta(n)$. Однако время построения автомата может оказаться значительным, если алфавит Σ большой. В разделе 32.4 описан остроумный способ решения этой задачи.

В начале этого раздела дадим определение конечного автомата. Затем ознакомимся со специальными автоматами, предназначенными для поиска подстрок, и покажем, как с их помощью можно найти все вхождения образца в текст. Материал включает детальное описание имитации поведения автомата поиска подстрок при обработке текста. Наконец, будет показано, как сконструировать автомат поиска подстрок для заданного входного образца.

Конечные автоматы

Конечный автомат (finite automaton) M — это пятерка $(Q, q_0, A, \Sigma, \delta)$, где

- Q — конечное множество ***состояний*** (states),
- $q_0 \in Q$ — ***начальное состояние*** (start state),
- $A \subseteq Q$ — конечное множество ***допустимых состояний*** (accepting states),
- Σ — конечный ***входной алфавит*** (input alphabet),
- δ — функция, которая отображает множество $Q \times \Sigma$ на множество Q и называется ***функцией переходов*** (transition function) автомата M.

Вначале конечный автомат находится в состоянии q_0 и считывает символы входной строки один за другим. Если автомат находится в состоянии q и считал входной символ a, то автомат переходит из состояния q в состояние $\delta(q, a)$. Если текущее состояние q является членом множества A, то говорят, что машина M ***допускает*** (accepted) считанную строку. Если же $q \notin A$, то входные данные называют ***отвергнутыми*** (rejected). Эти определения проиллюстрированы на рис. 32.5 на простом примере автомата с двумя состояниями. В части *а* этого рисунка приведено табличное представление функции переходов δ. В части *б* изображена эквивалентная диаграмма состояний. Состояние 1 — единственное допустимое состояние (оно показано черным цветом). Переходы представлены ориентированными ребрами. Например, ребро, ведущее из состояния 1 в состояние 0, которое обозначено меткой b, указывает, что $\delta(1, b) = 0$. Рассматриваемый автомат допускает те строки, которые оканчиваются нечетным количеством символов a. Точнее говоря, строка x воспринимается тогда и только тогда, когда $x = yz$, где $y = \varepsilon$ или строка y оканчивается символом b, и $z = a^k$, где k — нечетное. Например, последовательность состояний, которые проходит этот автомат для входной строки $abaaa$ (включая начальное состояние), имеет вид $\langle 0, 1, 0, 1, 0, 1 \rangle$, так что входная

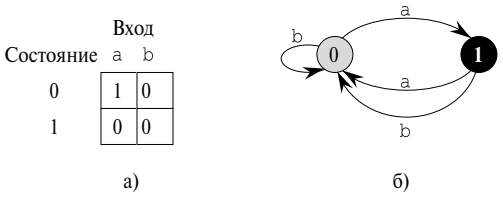


Рис. 32.5. Простой автомат с двумя состояниями: $Q = \{0, 1\}$, $q_0 = 0$, $\Sigma = \{a, b\}$

строка воспринимается. Если входная строка имеет вид $abbaa$, то автомат проходит последовательность состояний $\langle 0, 1, 0, 0, 1, 0\rangle$, поэтому такая входная строка отвергается.

С конечным автоматом M связана функция ϕ, которая называется ***функцией конечного состояния*** (final state function) и которая отображает множество Σ^* на множество Q. Значение $\phi(w)$ представляет собой состояние, в котором оказывается автомат M после прочтения строки w. Таким образом, M допускает строку w тогда и только тогда, когда $\phi(w) \in A$. Функция ϕ определяется следующим рекуррентным соотношением:

$$\phi(\varepsilon) = q_0,$$
$$\phi(wa) = \delta(\phi(w), a) \text{ для } w \in \Sigma^*, a \in \Sigma.$$

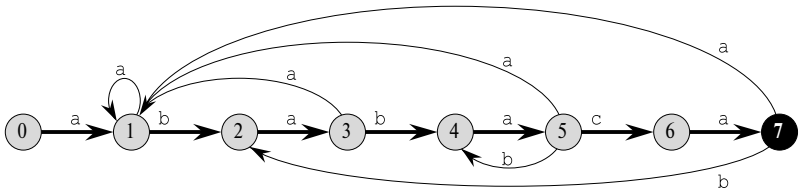
Автоматы поиска подстрок

Для каждого образца P существует свой автомат поиска подстрок; его необходимо сконструировать по образцу на этапе предварительной обработки, чтобы впоследствии его можно было использовать для поиска текстовой строки. Процесс такого конструирования для образца $P = abababaca$ проиллюстрирован на рис. 32.6. С этого момента предполагается, что образец P — это заданная фиксированная строка: для краткости мы будем опускать в обозначениях зависимость от P.

Чтобы создать автомат поиска подстрок, соответствующий заданному образцу $P[1..m]$, сначала определим вспомогательную функцию σ, которая называется ***суффиксной функцией*** (suffix function), отвечающей образцу P. Функция σ является отображением множества Σ^* на множество $\{0, 1, \ldots, m\}$, таким что величина $\sigma(x)$ равна длине максимального префикса P, который является суффиксом строки x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\}.$$

Суффиксная функция σ вполне определена, поскольку пустая строка $P_0 = \varepsilon$ является суффиксом любой строки. В качестве примера рассмотрим образец вида



а)

Состояние	Вход a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

б)

i	–	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	–	a	b	a	b	a	b	a	c	a	b	a
Состояние $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

в)

Рис. 32.6. Процесс конструирования конечного автомата для образца $P = ababaca$

$P = ab$; тогда $\sigma(\varepsilon) = 0$, $\sigma(ccaca) = 1$ и $\sigma(ccab) = 2$. Для образца P длиной m равенство $\sigma(x) = m$ выполняется тогда и только тогда, когда $P \sqsupset x$. Из определения суффиксной функции следует, что если $x \sqsupset y$, то $\sigma(x) \leqslant \sigma(y)$.

Определим автомат поиска подстрок, соответствующий образцу $P[1..m]$, следующим образом.

- Множество состояний Q имеет вид $\{0, 1, \ldots m\}$. В роли начального состояния q_0 выступает состояние 0, а единственным допустимым является состояние m.
- Функция переходов δ определена для любого состояния q и символа a уравнением

$$\delta(q, a) = \sigma(P_q a). \tag{32.3}$$

Поясним, откуда берется соотношение $\delta(q, a) = \sigma(P_q a)$. В качестве инварианта цикла этот автомат поддерживает соотношение

$$\phi(T_i) = \sigma(T_i); \tag{32.4}$$

этот результат доказывается в сформулированной ниже теореме 32.4. На словах это означает, что после сканирования первых i символов текстовой строки T автомат переходит в состояние $\phi(T_i) = q$, где $q = \sigma(T_i)$ — длина самого длинного

суффикса T_i, который одновременно является префиксом образца P. Если следующим сканируемым символом является символ $T[i+1] = a$, то машина должна осуществить переход в состояние $\sigma(T_{i+1}) = \sigma(T_i a)$. Из доказательства этой теоремы видно, что $\sigma(T_i a) = \sigma(P_q a)$. Другими словами, чтобы вычислить количество символов в самом длинном суффиксе $T_i a$, который является префиксом образца P, можно найти самый длинный суффикс $P_q a$, который является префиксом P. На каждом этапе машине нужна информация о длине самого длинного префикса P, который является суффиксом считанной до сих пор строки. Таким образом, если положить $\delta(q, a) = \sigma(P_q a)$, то будет поддерживаться нужный инвариант (32.4). Вскоре это неформальное пояснение будет изложено в более строгой форме.

Например, в автомате поиска подстрок, представленном на рис. 32.6, $\delta(5, b) = 4$. Этот переход осуществляется потому, что если автомат считывает символ b, находясь в состоянии $q = 5$, то $P_q b = ababab$, и самый длинный префикс образца P, совпадающий с суффиксом строки $ababab$ — $P_4 = abab$.

Рассмотрим рис. 32.6 подробнее. В части *а* приведена диаграмма состояний для автомата поиска подстрок, который воспринимает все строки, оканчивающиеся строкой $ababaca$. Состояние 0 — начальное, а состояние 7 (оно выделено черным цветом) — единственное допустимое состояние. Ориентированные ребра, направленные от состояния i в состояние j и обозначенные меткой a, представляют значение функции $\delta(i, a) = j$. Ребра, направленные вправо, которые образуют "хребет" автомата и выделены на рисунке жирными линиями, соответствуют точным совпадениям образца и входных символов. Ребра, направленные влево, соответствуют несовпадениям. Некоторые ребра, соответствующие несовпадениям, не показаны; в соответствии с соглашением, если для некоторого символа $a \in \Sigma$ у состояния i отсутствуют исходящие ребра с меткой a, то $\delta(i, a) = 0$. В части *б* представлена соответствующая функция переходов δ, а также строка образца $P = ababaca$. Элементы, соответствующие совпадениям между образцом и входными символами, выделены серым цветом. В части *в* проиллюстрирована обработка автоматом текста $T = abababacaba$. Под каждым символом текста $T[i]$ указано состояние $\phi(T_i)$, в котором находится автомат после обработки префикса T_i. Образец встречается в тексте один раз, причем совпадающая подстрока оканчивается на девятой позиции.

Чтобы пояснить работу автомата, предназначенного для поиска подстрок, приведем простую, но эффективную программу для моделирования поведения такого автомата (представленного функцией переходов δ) при поиске образца P длиной m во входном тексте $T[1..n]$. Как и в любом другом автомате поиска подстрок, предназначенном для образца длиной m, множество состояний Q имеет вид $\{0, 1, \ldots, m\}$, начальное состояние имеет индекс 0, а единственным допустимым является состояние m.

```
FINITE_AUTOMATON_MATCHER(T, δ, m)
 1  n ← length[T]
 2  q ← 0
 3  for i ← 1 to n
 4      do q ← δ(q, T[i])
 5         if q = m
 6            then print "Образец обнаружен при сдвиге" i − m
```

Поскольку структура процедуры FINITE_AUTOMATON_MATCHER представляет собой простой цикл, время поиска совпадений с его помощью в тексте длиной n равна $\Theta(n)$. Однако сюда не входит время предварительной обработки, которое необходимо для вычисления функции переходов δ. К этой задаче мы обратимся позже, после доказательства корректности работы процедуры FINITE_AUTOMATON_MATCHER.

Рассмотрим, как автомат обрабатывает входной текст $T[1..n]$. Докажем, что после сканирования символа $T[i]$ автомат окажется в состоянии $\sigma(T_i)$. Поскольку соотношение $\sigma(T_i) = m$ справедливо тогда и только тогда, когда $P \sqsupset T_i$, машина окажется в допустимом состоянии тогда и только тогда, когда она только что считала образец P. Чтобы доказать этот результат, воспользуемся двумя приведенными ниже леммами, в которых идет речь о суффиксной функции σ.

Лемма 32.2 (Неравенство суффиксной функции). Для любой строки x и символа a выполняется неравенство $\sigma(xa) \leqslant \sigma(x) + 1$.

Доказательство. Введем обозначение $r = \sigma(xa)$ (рис. 32.7). Если $r = 0$, то неравенство $\sigma(xa) = r \leqslant \sigma(x) + 1$ тривиальным образом следует из того, что величина $\sigma(x)$ неотрицательна. Теперь предположим, что $r > 0$. Тогда $P_r \sqsupset xa$ согласно определению функции σ. Если в конце строк P_r и xa отбросить по символу a, то мы получим $P_{r-1} \sqsupset x$. Следовательно, справедливо неравенство $r - 1 \leqslant \sigma(x)$, так как $\sigma(x)$ — максимальное значение k, при котором выполняется соотношение $P_k \sqsupset x$, и $\sigma(xa) = r \leqslant \sigma(x) + 1$. ■

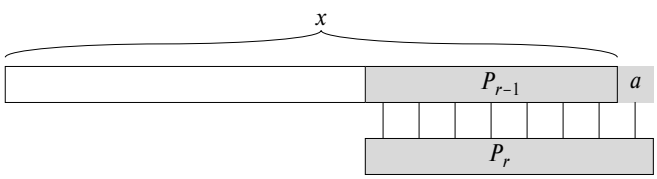


Рис. 32.7. Иллюстрация к доказательству леммы 32.2; из рисунка видно, что выполняется неравенство $r \leqslant \sigma(x) + 1$, где $r = \sigma(xa)$

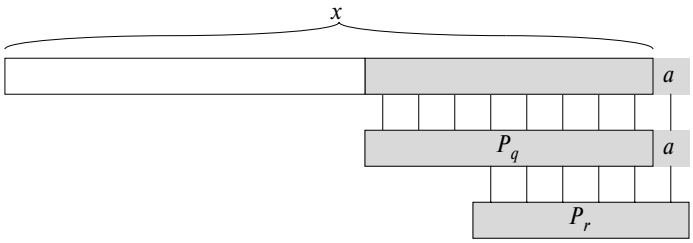


Рис. 32.8. Иллюстрация к доказательству леммы 32.3; из рисунка видно, что выполняется соотношение $r = \sigma(P_q a)$, где $q = \sigma(x)$ и $r = \sigma(xa)$

Лемма 32.3 (Лемма о рекурсии суффиксной функции). Если для произвольных строки x и символа a $q = \sigma(x)$, то $\sigma(xa) = \sigma(P_q a)$.

Доказательство. Согласно определению функции σ, $P_q \sqsupset x$. Как видно из рис. 32.8, наряду с ним выполняется соотношение $P_q a \sqsupset xa$. Если ввести обозначение $r = \sigma(xa)$, то, согласно лемме 32.2, справедливо неравенство $r \leqslant q + 1$. Поскольку $P_q a \sqsupset xa$, $P_r \sqsupset xa$ и $|P_r| \leqslant |P_q a|$, из леммы 32.1 следует, что $P_r \sqsupset P_q a$. Поэтому справедливо неравенство $r \leqslant \sigma(P_q a)$, т.е. $\sigma(xa) \leqslant \sigma(P_q a)$. Но, кроме того, $\sigma(P_q a) \leqslant \sigma(xa)$, поскольку $P_q a \sqsupset xa$. Таким образом, $\sigma(xa) = \sigma(P_q a)$. ■

Теперь все готово для доказательства основной теоремы, характеризующей поведение автомата поиска подстрок при обработке входного текста. Как уже упоминалось, в этой теореме показано, что автомат на каждом шагу просто отслеживает, какой самый длинный префикс образца совпадает с суффиксом части текста, считанной до сих пор. Другими словами, в автомате поддерживается инвариант (32.4).

Теорема 32.4. Если ϕ — функция конечного состояния, соответствующая автомату поиска подстрок с заданным образцом P, а $T[1..n]$ — входной текст этого автомата, то

$$\phi(T_i) = \sigma(T_i)$$

для $i = 0, 1, \ldots, n$.

Доказательство. Докажем теорему по индукции по i. Если $i = 0$, то справедливость теоремы очевидна, поскольку $T_0 = \varepsilon$. Таким образом, $\phi(T_0) = 0 = \sigma(T_0)$.

Теперь предположим, что выполняется равенство $\phi(T_i) = \sigma(T_i)$, и докажем, что $\phi(T_{i+1}) = \sigma(T_{i+1})$. Обозначим величину $\phi(T_i)$ как q, а величину $T[i+1]$ —

через a. Тогда можно написать цепочку следующих соотношений:

$$\begin{aligned}
\phi(T_{i+1}) &= \phi(T_i a) = && \text{(по определению величин } T_{i+1} \text{ и } a\text{)} \\
&= \delta(\phi(T_i), a) = && \text{(по определению } \phi\text{)} \\
&= \delta(q, a) = && \text{(по определению } q\text{)} \\
&= \sigma(P_q a) = && \text{(по определению } \delta \text{ (32.3))} \\
&= \sigma(T_i a) = && \text{(согласно лемме 32.3 и гипотезе индукции)} \\
&= \sigma(T_{i+1}) && \text{(по определению } T_{i+1}\text{).}
\end{aligned}$$

■

В соответствии с теоремой 32.4, если автомат попадает в состояние q в строке 4, то q — наибольшее значение, такое что $P_q \sqsupset T_i$. Таким образом, в строке 5 равенство $q = m$ выполняется тогда и только тогда, когда только что был считан образец P. Итак, мы приходим к выводу, что процедура FINITE_AUTOMATON_MATCHER работает корректно.

Вычисление функции переходов

Приведенная ниже процедура вычисляет функцию переходов δ по заданному образцу $P[1..m]$.

```
COMPUTE_TRANSITION_FUNCTION(P, Σ)
1  m ← length[P]
2  for q ← 0 to m
3       do for (для) каждого символа a ∈ Σ
4            do k ← min(m + 1, q + 2)
5               repeat k ← k − 1
6                 until P_k ⊐ P_q a
7               δ(q, a) ← k
8  return δ
```

В этой процедуре функция $\delta(q, a)$ вычисляется непосредственно, исходя из ее определения. Во вложенных циклах, которые начинаются в строках 2 и 3, рассматриваются все состояния q и символы a, а в строках 4–7 функции $\delta(q, a)$ присваиваются максимальные значения k, при которых справедливо соотношение $P_k \sqsupset P_q a$.

Время работы процедуры FINITE_AUTOMATON_MATCHER равно $O\left(m^3 |\Sigma|\right)$, поскольку внешние циклы дают вклад, соответствующий умножению на $m|\Sigma|$, внутренний цикл **repeat** может повториться не более $m + 1$ раз, а для выполнения проверки в строке 6 может понадобиться сравнить вплоть до m символов. Существуют процедуры, работающие намного быстрее; время, необходимое для вычисления функции δ по заданному образцу P, путем применения некоторых величин,

остроумно вычисленных для этого образца (см. упражнение 32.4-6), можно улучшить до величины $O(m|\Sigma|)$. С помощью такой усовершенствованной процедуры вычисления функции δ можно найти все вхождения образца длиной m в текст длиной n для алфавита Σ, затратив на предварительную обработку время $O(m|\Sigma|)$, а на сравнение — время $\Theta(n)$.

Упражнения

32.3-1. Сконструируйте автомат поиска подстрок для образца $P = aabab$ и проиллюстрируйте его работу при обработке текста $T = aaababaabaababaab$.

32.3-2. Изобразите диаграмму состояний для автомата поиска подстрок, если образец имеет вид $ababbabbababbababbabb$, а алфавит — $\Sigma = \{a, b\}$.

32.3-3. Образец P называют ***строкой с уникальными префиксами*** (nonoverlappable), если из соотношения $P_k \sqsupset P_q$ следует, что $k = 0$ или $k = q$. Как выглядит диаграмма переходов автомата для поиска подстроки с уникальными префиксами?

⋆ 32.3-4. Заданы два образца P и P'. Опишите процесс построения конечного автомата, позволяющего найти все вхождения *любого из* образцов. Попытайтесь свести к минимуму количество состояний такого автомата.

32.3-5. Задан образец P, содержащий пробельные символы (см. упражнение 32.1-4). Опишите процесс построения конечного автомата, позволяющего найти все вхождения образца P в текст T, затратив при этом на сравнение время $O(n)$, где $n = |T|$.

⋆ 32.4 Алгоритм Кнута-Морриса-Пратта

Теперь представим алгоритм сравнения строк, который был предложен Кнутом (Knuth), Моррисом (Morris) и Праттом (Pratt), и время работы которого линейно зависит от объема входных данных. В этом алгоритме удается избежать вычисления функции переходов δ, а благодаря использованию вспомогательной функции $\pi[1..m]$, которая вычисляется по заданному образцу за время $\Theta(m)$, время сравнения в этом алгоритме равно $\Theta(n)$. Массив π позволяет эффективно (в амортизированном смысле) вычислять функцию δ “на лету”, т.е. по мере необходимости. Грубо говоря, для любого состояния $q = 0, 1, \ldots, m$ и любого символа $a \in \Sigma$ величина $\pi[q]$ содержит информацию, которая не зависит от символа a и необходима для вычисления величины $\delta(q, a)$ (пояснения по поводу этого замечания будут даны ниже). Поскольку массив π содержит только m элементов (в то время как в массиве δ их $\Theta(m|\Sigma|)$), вычисляя на этапе предварительной обработки функцию π вместо функции δ, удается уменьшить время предварительной обработки образца в $|\Sigma|$ раз.

Префиксная функция для образца

Префиксная функция π, предназначенная для какого-нибудь образца, инкапсулирует сведения о том, в какой мере образец совпадает сам с собой после сдвигов. Эта информация позволяет избежать ненужных проверок в простейшем алгоритме поиска подстрок или предвычисления функции δ при использовании конечных автоматов.

Рассмотрим работу обычного алгоритма сопоставления. На рис. 32.9*а* показан некоторый сдвиг s шаблона с образцом $P = ababaca$ вдоль текста T. В этом примере $q = 5$ символов успешно совпали (они выделены серым цветом и соединены вертикальными линиями), а 6-й символ образца отличается от соответствующего символа текста. Сведения о количестве совпавших символов q позволяют сделать вывод о том, какие символы содержатся в соответствующих местах текста. Эти знания позволяют сразу же определить, что некоторые сдвиги будут недопустимыми. В представленном на рисунке примере сдвиг $s + 1$ точно недопустим, поскольку первый символ образца (a) находился бы напротив символа текста, для которого известно, что он совпадает со вторым символом образца (b). Однако из части *б* рисунка, на которой показан сдвиг $s' = s + 2$, видно, что первые три символа образца совпадают с тремя последними просмотренными символами текста, так что такой сдвиг априори нельзя отбросить как недопустимый. Полезную информацию, позволяющую сделать такой вывод, можно получить путем сравнения образца с самим собой. В части *в* рисунка изображен самый длинный префикс образца P, который также является собственным суффиксом строк P_5 и P_3. Эти данные предварительно вычисляются и заносятся в массив π, так что $\pi[5] = 3$. Если первые q символов совпали при сдвиге s, то следующий сдвиг, который может оказаться допустимым, равен $s' = s + (q - \pi[q])$.

Словом, в общем случае полезно знать ответ на сформулированный ниже вопрос.

> Пусть символы $P[1..q]$ образца совпадают с символами текста $T[s + + 1..s + q]$. Чему равен наименьший сдвиг $s' > s$, такой что
>
> $$P[1..k] = T\left[s' + 1..s' + k\right], \tag{32.5}$$
>
> где $s' + k = s + q$?

Такой сдвиг s' — первый после сдвига s, недопустимость которого не обязательно следует из наших знаний о подстроке $T[s + 1..s + q]$. В лучшем случае $s' = s + q$, и все сдвиги $s + 1, s + 2, \ldots, s + q - 1$ немедленно отбрасываются. В любом случае для нового сдвига s' нет необходимости сравнивать первые k символов образца P и соответствующие символы текста T, поскольку их совпадение гарантированно благодаря уравнению (32.5).

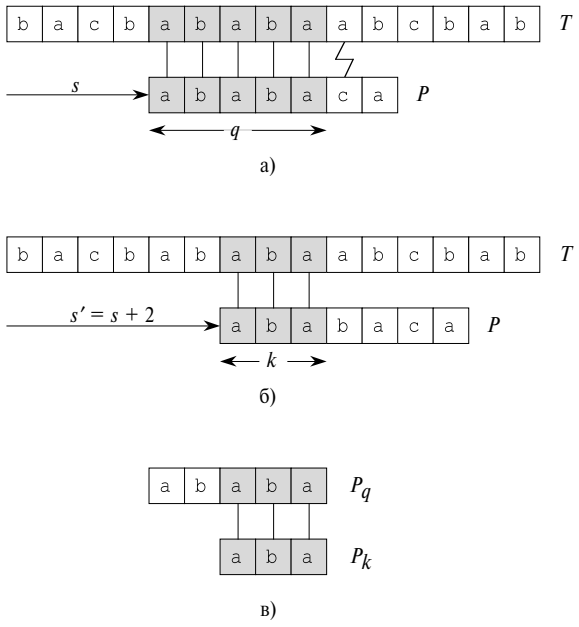


Рис. 32.9. Префиксная функция π

Необходимую информацию об образце можно получить, сдвигая его вдоль самого себя (см. рис.32.9*c*). Поскольку $T[s'+1..s'+k]$ — известная часть текста, она является суффиксом строки P_q. Поэтому уравнение (32.5) можно рассматривать как запрос о максимальном значении $k < q$, таком что $P_k \sqsupset P_q$. Тогда следующий потенциально допустимый сдвиг равен $s' = s + (q - k)$. Оказывается, что удобнее хранить количество k совпадающих при новом сдвиге s' символов, чем, например, величину $s' - s$. С помощью этой информации можно ускорить и простейший алгоритм поиска подстроки, и работу конечного автомата.

Теперь дадим формальное определение. ***Префиксной функцией*** (prefix function) заданного образца $P[1..m]$ называется функция $\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$, такая что

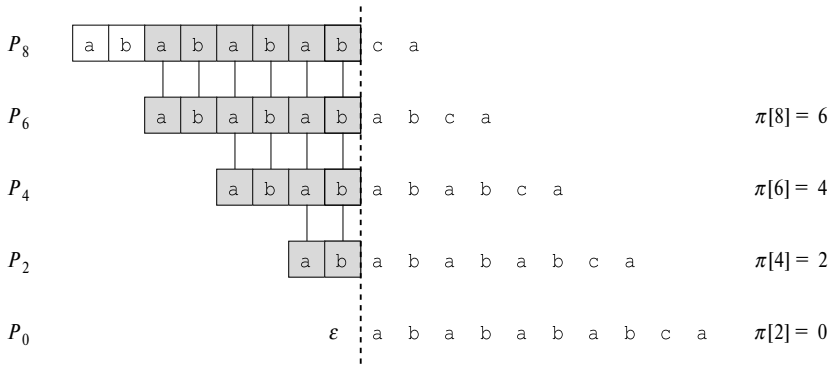
$$\pi[q] = \max\{k : k < q \text{ и } P_k \sqsupset P_q\}.$$

Другими словами, $\pi[q]$ равно длине наибольшего префикса образца P, который является истинным суффиксом строки P_q. В качестве другого примера на рис. 32.10*a* приведена полная префиксная функция π для образца $abababaabca$.

Алгоритм поиска подстрок Кнута-Морриса-Пратта приведен ниже в виде псевдокода процедуры KMP_MATCHER. В процедуре KMP_MATCHER вызывается вспомогательная процедура COMPUTE_PREFIX_FUNCTION, в которой вычисляется функция π.

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

а)



б)

Рис. 32.10. Иллюстрация к лемме 32.5 для образца $P = abababa bca$ и $q = 8$

KMP_MATCHER(T, P)

```
 1  n ← length[T]
 2  m ← length[P]
 3  π ← COMPUTE_PREFIX_FUNCTION(P)
 4  q ← 0                              ▷ Число совпавших символов
 5  for i ← 1 to n                     ▷ Сканирование текста слева направо
 6      do while q > 0 и P[q + 1] ≠ T[i]
 7            do q ← π[q]              ▷ Следующий символ не совпадает
 8         if P[q + 1] = T[i]
 9           then q ← q + 1           ▷ Следующий символ совпадает
10         if q = m                   ▷ Совпали ли все символы образца P?
11           then print "Образец обнаружен при сдвиге" i − m
12                q ← π[q]            ▷ Поиск следующего совпадения
```

```
COMPUTE_PREFIX_FUNCTION(P)
 1  m ← length[P]
 2  π[1] ← 0
 3  k ← 0
 4  for q ← 2 to m
 5       do while k > 0 и P[k + 1] ≠ P[q]
 6                do k ← π[k]
 7          if P[k + 1] = P[q]
 8            then k ← k + 1
 9          π[q] ← k
10  return π
```

Сначала проанализируем время работы этой процедуры. Доказательство ее корректности окажется более сложным.

Анализ времени работы

С помощью метода потенциалов амортизационного анализа (см. раздел 17.3) можно показать, что время работы процедуры COMPUTE_PREFIX_FUNCTION равно $\Theta(m)$. Потенциал величины k связывается с текущим ее значением в алгоритме. Как видно из строки 3, начальное значение этого потенциала равно 0. В строке 6 значение величины k уменьшается при каждом его вычислении, поскольку $\pi[k] < k$. Однако в силу неравенства $\pi[k] \geqslant 0$, которое справедливо при всех k, значение этой переменной никогда не бывает отрицательным. Единственная другая строка, которая тоже оказывает влияние на значение переменной k, — строка 8. Благодаря ее наличию при каждом выполнении тела цикла **for** значение переменной k увеличивается не более чем на 1. Поскольку перед входом в цикл выполняется неравенство $k < q$ и поскольку значение переменной q увеличивается в каждой итерации цикла **for**, справедливость неравенства $k < q$ сохраняется (подтверждая тот факт, что соблюдается также неравенство $\pi[q] < q$ в строке 9). Каждое выполнение тела цикла **while** в строке 6 можно оплатить соответствующим уменьшением потенциальной функции, поскольку $\pi[k] < k$. В строке 8 потенциальная функция возрастает не более чем на 1, поэтому амортизированная стоимость тела цикла в строках 5–9 равна $O(1)$. Так как количество итераций внешнего цикла равно $\Theta(m)$, и поскольку конечное значение потенциальной функции по величине не меньше, чем ее начальное значение, полное фактическое время работы процедуры COMPUTE_PREFIX_FUNCTION в наихудшем случае равно $\Theta(m)$.

Аналогичный амортизационный анализ, в котором в качестве потенциальной функции используется значение величины q, показывает, что время выполнения сравнений в процедуре KMP_MATCHER равно $\Theta(n)$.

Благодаря использованию функции π вместо функции δ, которая используется в процедуре FINITE_AUTOMATON_MATCHER, время предварительной обработки образца уменьшается от $O(m|\Sigma|)$ до $\Theta(m)$, в то время как оценка реального времени поиска остается равной $\Theta(n)$.

Корректность вычисления префиксной функции

Начнем с рассмотрения важной леммы, в которой показано, что путем итерации префиксной функции π можно перечислить все префиксы P_k, которые являются истинными суффиксами заданного префикса P_q. Введем обозначение

$$\pi^*[q] = \left\{\pi[q], \pi^{(2)}[q], \pi^{(3)}[q], \ldots, \pi^{(t)}[q]\right\},$$

где величина $\pi^{(i)}[q]$ обозначает i-ю итерацию префиксной функции, т.е. $\pi^{(0)}[q] = q$ и при $i \geqslant 1$ $\pi^{(i)}[q] = \pi\left[\pi^{(i-1)}[q]\right]$. Кроме того, понятно, что последовательность $\pi^*[q]$ обрывается, когда в ней будет достигнуто значение $\pi^{(t)}[q] = 0$. Приведенная ниже лемма, проиллюстрированная на рис. 32.10, характеризует последовательность $\pi^*[q]$.

Лемма 32.5 (Лемма об итерации префиксной функции). Пусть P — образец длиной m с префиксной функцией π. Тогда для всех $q = 1, 2, \ldots, m$ имеем $\pi^*[q] = \{k : k < q \text{ и } P_k \sqsupset P_q\}$.

Доказательство. Сначала докажем, что

$$\text{из } i \in \pi^*[q] \text{ следует } P_i \sqsupset P_q. \tag{32.6}$$

Если $i \in \pi^*[q]$, то $i = \pi^{(u)}[q]$ для некоторого $u > 0$. Докажем (32.6) по индукции по u. При $u = 1$ $i = \pi[q]$ и сформулированное выше утверждение следует из того, что $i < q$ и $P_{\pi[q]} \sqsupset P_q$. Воспользовавшись соотношениями $\pi[i] < i$ и $P_{\pi[i]} \sqsupset P_i$, а также транзитивностью операций $<$ и $\sqsupset$, можно установить справедливость нашего утверждения для всех i из $\pi^*[q]$. Следовательно, $\pi^*[q] \subseteq \{k : k < q \text{ и } P_k \sqsupset P_q\}$.

То, что $\{k : k < q \text{ и } P_k \sqsupset P_q\} \subseteq \pi^*[q]$, мы докажем методом “от противного”. Предположим, что в множестве $\{k : k < q \text{ и } P_k \sqsupset P_q\} - \pi^*[q]$ содержатся целые числа и что j — наибольшее из них. Поскольку $\pi[q]$ — наибольшее значение множества $\{k : k < q \text{ и } P_k \sqsupset P_q\}$, и в силу того, что $\pi[q] \in \pi^*[q]$, должно выполняться неравенство $j < \pi[q]$. Обозначим через j' наименьший целый элемент множества $\pi^*[q]$, превышающий j. (Если в множестве $\pi^*[q]$ не содержится других значений, превышающих j, можно выбрать $j' = \pi[q]$.) В силу того, что $j \in \{k : k < q \text{ и } P_k \sqsupset P_q\}$, $P_j \sqsupset P_q$, а кроме того, $P'_j \sqsupset P_q$, поскольку $j' \in \pi^*[q]$. Таким образом, согласно лемме 32.1, справедливо соотношение $P_j \sqsupset P'_j$ и j — наибольшее из значений, меньших значения j' и обладающих этим свойством. Поэтому должно выполняться равенство $\pi[j'] = j$ и, поскольку $j' \in \pi^*[q]$, должно выполняться соотношение $j \in \pi^*[q]$. Это противоречие и доказывает лемму. ■

На рис. 32.10 лемма 32.5 проиллюстрирована для шаблона $P = abababacba$ и $q = 8$. Поскольку $\pi[8] = 6$, $\pi[6] = 4$, $\pi[4] = 2$ и $\pi[2] = 0$, путем итерации функции π получим $\pi^*[8] = \{6, 4, 2, 0\}$. В части *б* рис. 32.10 показаны последовательные сдвиги шаблона с образцом P вправо. Обратите внимание, как после каждого сдвига некоторый префикс P_k образца P совпадает с некоторым собственным суффиксом строки P_8; это происходит при $k = 6, 4, 2, 0$. На этом рисунке в первой строке приведен образец P, а пунктирная вертикальная линия обозначает конец строки P_8. В последовательных строках изображены все сдвиги образца P, при которых некоторый префикс P_k образца P совпадает с некоторым суффиксом строки P_8. Совпадающие символы выделены серым цветом. Вертикальные линии соединяют совпадающие символы. Таким образом, $\{k : k < q \text{ и } P_k \sqsupset P_q\} = \{6, 4, 2, 0\}$. В лемме утверждается, что для всех q $\pi^*[q] = \{k : k < q \text{ и } P_k \sqsupset P_q\}$.

Алгоритм COMPUTE_PREFIX_FUNCTION по порядку вычисляет $\pi[q]$ для $q = 1, 2, \ldots, m$. Корректность вычисления значения $\pi[1] = 0$ во второй строке этой процедуры не вызывает сомнений, поскольку при всех q выполняется неравенство $\pi[q] < q$. Приведенная ниже лемма и следствие из нее будут использованы для доказательства того факта, что в процедуре COMPUTE_PREFIX_FUNCTION функция $\pi[q]$ вычисляется корректно при всех $q > 1$.

Лемма 32.6. Пусть P — образец длиной m, а π — префиксная функция этого образца. Тогда $\pi[q] - 1 \in \pi^*[q-1]$ для всех $q = 1, 2, \ldots, m$, для которых $\pi[q] > 0$.

Доказательство. Если $r = \pi[q] > 0$, то $r < q$ и $P_r \sqsupset P_q$; таким образом, $r - 1 < q - 1$ и $P_{r-1} \sqsupset P_{q-1}$ (отбрасывая последний символ в строках P_r и P_q). Поэтому, согласно лемме 32.5, $\pi[q] - 1 = r - 1 \in \pi^*[q-1]$. ■

Для $q = 2, 3, \ldots, m$ определим подмножество $E_{q-1} \subseteq \pi^*[q-1]$ следующим образом:

$$\begin{aligned} E_{q-1} &= \{k \in \pi^*[q-1] : P[k+1] = P[q]\} = \\ &= \{k : k < q-1 \text{ и } P_k \sqsupset P_{q-1} \text{ и } P[k+1] = P[q]\} = \\ &= \{k : k < q-1 \text{ и } P_{k+1} \sqsupset P_q\}, \end{aligned}$$

где предпоследнее равенство следует из леммы 32.5. Итак, множество E_{q-1} состоит из таких значений $k < q-1$, что $P_k \sqsupset P_{q-1}$ и $P_{k+1} \sqsupset P_q$ (последнее следует из того, что $P[k+1] = P[q]$). Таким образом, множество E_{q-1} состоит из тех значений $k \in \pi^*[q-1]$, для которых строку P_k можно расширить до строки P_{k+1}, получив в результате истинный суффикс строки P_q.

Следствие 32.7. Пусть P — образец длиной m, а π — префиксная функция этого образца. Тогда при $q = 2, 3, \ldots, m$

$$\pi[q] = \begin{cases} 0 & \text{если } E_{q-1} = \emptyset, \\ 1 + \max\{k \in E_{q-1}\} & \text{если } E_{q-1} \neq \emptyset. \end{cases}$$

Доказательство. Если множество E_{q-1} пустое, то не существует значений $k \in \in \pi^*[q-1]$ (включая $k = 0$), для которых строку P_k можно расширить до строки P_{k+1} и получить собственный суффикс P_q. Следовательно, $\pi[q] = 0$.

Если множество E_{q-1} не пустое, то для каждого значения $k \in E_{q-1}$ справедливы соотношения $k + 1 < q$ и $P_{k+1} \sqsupset P_q$. Следовательно, из определения $\pi[q]$ мы имеем

$$\pi[q] \geqslant 1 + \max\{k \in E_{q-1}\}. \tag{32.7}$$

Заметим, что $\pi[q] > 0$. Обозначим $r = \pi[q] - 1$, так что $r + 1 = \pi[q]$. Поскольку $r + 1 > 0$, имеем $P[r+1] = P[q]$. Кроме того, согласно лемме 32.6, $r \in \pi^*[q-1]$. Таким образом, $r \in E_{q-1}$, так что $r \leqslant \max\{k \in E_{q-1}\}$ или, что эквивалентно,

$$\pi[q] \leqslant 1 + \max\{k \in E_{q-1}\}. \tag{32.8}$$

Объединение уравнений (32.7) и (32.8) завершает доказательство. ■

Теперь завершим доказательство того, что процедура COMPUTE_PREFIX_FUNCTION корректно вычисляет функцию π. В этой процедуре в начале каждой итерации цикла **for** в строках 4–9 выполняется равенство $k = \pi[q-1]$. Это условие обеспечивается строками 2 и 3 во время первого вхождения в цикл и остается справедливым в каждой успешной итерации благодаря строке 9. В строках 5–8 значение переменной k изменяется таким образом, что становится корректным значение $\pi[q]$. В цикле в строках 5–6 выполняется поиск по всем значениям $k \in \in \pi^*[q-1]$, пока не будет найдено такое значение, для которого $P[k+1] = P[q]$. В этот момент k является наибольшим значением в множестве E_{q-1}, поэтому, согласно следствию 32.7, величину $\pi[q]$ можно положить равной $k + 1$. Если же искомое значение k не найдено, в строке 7 выполняется присваивание $k = = 0$. Если $P[1] = P[q]$, то мы должны присвоить значение 1 как k, так и $\pi[q]$; в противном случае переменную k следует оставить неизменной, а величине $\pi[q]$ присвоить значение 0. В любом случае в строках 7–9 величинам k и $\pi[q]$ присваиваются правильные значения. На этом доказательство корректности процедуры COMPUTE_PREFIX_FUNCTION можно считать завершенным.

Корректность алгоритма Кнута-Морриса-Пратта

Процедуру KMP_MATCHER можно рассматривать как видоизмененную реализацию процедуры FINITE_AUTOMATON_MATCHER. В частности, мы докажем,

что код в строках 6–9 процедуры KMP_MATCHER эквивалентен строке 4 процедуры FINITE_AUTOMATON_MATCHER, в которой переменной q присваивается значение $\delta(q, T[i])$. Однако вместо того, чтобы использовать сохраненную величину $\delta(q, T[i])$, она вычисляется по мере необходимости с помощью функции π. Поскольку ранее было показано, что процедура KMP_MATCHER имитирует поведение процедуры FINITE_AUTOMATON_MATCHER, корректность процедуры KMP_MATCHER следует из корректности процедуры FINITE_AUTOMATON_MATCHER (и вскоре станет понятно, почему в процедуре KMP_MATCHER необходима строка 12).

Корректность процедуры KMP_MATCHER следует из утверждения, что должно выполняться либо соотношение $\delta(q, T[i]) = 0$, либо соотношение $\delta(q, T[i]) - 1 \in \in \pi^*[q]$. Чтобы проверить это утверждение, обозначим $k = \delta(q, T[i])$. Тогда из определений функций δ и σ следует, что $P_k \sqsupset P_q T[i]$. Следовательно, либо $k = = 0$, либо $k \geqslant 1$ и (отбрасывая последний символ из P_k и $P_q T[i]$) $P_{k-1} \sqsupset P_q$ (в этом случае $k - 1 \in \pi^*[q]$). Таким образом, либо $k = 0$, либо $k - 1 \in \pi^*[q]$, что и доказывает сформулированное выше утверждение.

Доказанное утверждение используется следующим образом. Обозначим через q' значение, которое принимает величина q в строке 6, и воспользуемся уравнением $\pi^*[q] = \{k : k < q \text{ и } P_k \sqsupset P_q\}$ из леммы 32.5 для обоснования итерации $q \leftarrow \pi[q]$, перечисляющей элементы множества $\{k : P_k \sqsupset P'_q\}$. В строках 6–9 путем проверки элементов множества $\pi^*[q']$ в убывающем порядке определяется величина $\delta(q', T[i])$. Указанное утверждение используется в коде для того, чтобы начать работу с $q = \phi(T_{i-1}) = s(T_{i-1})$ и выполнять итерации $q \leftarrow \pi[q]$ до тех пор, пока не будет найдено значение q, такое что $q = 0$ или $P[q+1] = = T[i]$. В первом случае $\delta(q', T[i] = 0)$, а во втором — значение переменной q равно максимальному элементу множества E'_q, так что $\delta(q', T[i]) = q+1$ согласно следствию 32.7.

Строка 12 в процедуре KMP_MATCHER необходима для того, чтобы избежать возможного обращения к элементу $P[m+1]$ в строке 6 после того, как будет обнаружено вхождение образца P. (Согласно указанию, приведенному в упражнении 32.4-6, аргумент, что $q = \sigma(T_{i-1})$, остается справедливым во время следующего выполнения строки 6: для любого $a \in \Sigma$ $\delta(m, a) = \delta(\pi[m], a)$ или, что эквивалентно, $\sigma(Pa) = \sigma\left(P_{\pi[m]}a\right)$.) Остальная часть обоснования корректности алгоритма Кнута-Морриса-Пратта следует из корректности процедуры FINITE_AUTOMATON_MATCHER, поскольку теперь должно быть понятно, что процедура KMP_MATCHER просто имитирует ее поведение.

Упражнения

32.4-1. Вычислите префиксную функцию для образца $ababbabbabbababbabb$ (алфавит имеет вид $\Sigma = \{a, b\}$).

32.4-2. Найдите верхнюю границу размера $\pi^*[q]$ как функцию от величины q. Приведите пример, показывающий, что ваша оценка не может быть улучшена.

32.4-3. Объясните, как найти вхождения образца P в текст T, зная функцию π для PT (т.е. строки длиной $m+n$, полученной в результате конкатенации строк P и T).

32.4-4. Покажите, как улучшить процедуру KMP_MATCHER, заменив функцию π в строке 7 (но не в строке 12) функцией π', рекурсивно определяемой для $q = 1, 2, \ldots, m$ следующим образом:

$$\pi'[q] = \begin{cases} 0 & \text{если } \pi[q] = 0, \\ \pi'[\pi[q]] & \text{если } \pi[q] \neq 0 \text{ и } P[\pi[q]+1] = P[q+1], \\ \pi[q] & \text{если } \pi[q] \neq 0 \text{ и } P[\pi[q]+1] \neq P[q+1]. \end{cases}$$

Объясните, почему модифицированный алгоритм работает корректно, а также в чем состоит смысл этого улучшения.

32.4-5. Разработайте алгоритм, который бы позволил в течение линейного времени определить, является ли текстовая строка T циклической перестановкой другой строки T'. Например, строки arc и car являются циклическими перестановками друг друга.

⋆ 32.4-6. Разработайте эффективный алгоритм вычисления функции переходов δ для конечного автомата поиска заданного образца P. Время работы алгоритма должно быть равно $O(m|\Sigma|)$. (*Указание*: докажите, что $\delta(q, a) = \delta(\pi[q], a)$, если $q = m$ или $P[q+1] \neq a$.)

Задачи

32-1. **Поиск подстрок на основе коэффициентов повторения**

Обозначим как y^i строку, полученную в результате i-кратной конкатенации строки y с самой собой. Например, $(ab)^3 = ababab$. Говорят, что ***коэффициент повторения*** (repetition factor) строки $x \in \Sigma^*$ равен r, если $x = y^r$ для некоторой строки $y \in \Sigma^*$ и значения $r > 0$. Через $\rho(x)$ обозначим наибольший из коэффициентов повторения x.

а) Разработайте эффективный алгоритм, принимающий в качестве входных данных образец $P[1..m]$ и вычисляющий значение $\rho(P_i)$ для $i = 1, 2, \ldots, m$. Чему равно время работы этого алгоритма?

б) Определим для произвольного образца $P[1..m]$ величину $\rho^*(P)$ как $\max_{1 \leqslant i \leqslant m} \rho(P_i)$. Докажите, что если образец P выбирается случайным образом из множества всех бинарных строк длиной m, то математическое ожидание $\rho^*(P)$ равно $O(1)$.

в) Покажите, что приведенный ниже алгоритм поиска подстрок корректно находит все вхождения образца P в текст $T[1..n]$ в течение времени $O(\rho^*(P)\, n + m)$.

```
REPETITION_MATCHER(P, T)
 1  m ← length[P]
 2  n ← length[T]
 3  k ← 1 + ρ*(P)
 4  q ← 0
 5  s ← 0
 6  while s ≤ n − m
 7        do if T[s + q + 1] = P[q + 1]
 8              then q ← q + 1
 9                   if q = m
10                      then print "Образец обнаружен при сдвиге" s
11           if q = m или T[s + q + 1] ≠ P[q + 1]
12              then s ← s + max(1, ⌈q/k⌉)
13                   q ← 0
```

Этот алгоритм был предложен Галилом (Galil) и Сейферасом (Seiferas). Развивая эти идеи, они получили алгоритм поиска подстрок с линейным временем работы, использующий всего $O(1)$ памяти, кроме необходимой для хранения строк P и T.

Заключительные замечания

Связь поиска подстрок с теорией конечных автоматов обсуждается в книге Ахо (Aho), Хопкрофта (Hopcroft) и Ульмана (Ullman) [5]. Алгоритм Кнута-Морриса-Пратта [187] разработан Кнутом (Knuth) и Праттом (Pratt) и независимо Моррисом (Morris); результаты своих исследований они опубликовали совместно. Алгоритм Рабина-Карпа был предложен Рабином (Rabin) и Карпом (Karp) [175]. Галил (Galil) и Сейферас (Seiferas) [107] разработали интересный детерминистический алгоритм поиска подстрок с линейным временем работы, в котором используется лишь $O(1)$ памяти сверх необходимой для хранения образца и текста.

ГЛАВА 33

Вычислительная геометрия

Вычислительная геометрия — это раздел теории вычислительных систем, изучающий алгоритмы, предназначенные для решения геометрических задач. В современных инженерных и математических расчетах вычислительная геометрия, в числе других областей знаний, применяется в машинной графике, в робототехнике, при разработке СБИС, при автоматизированном проектировании и в статистике. Роль входных данных в задачах вычислительной геометрии обычно играет описание множества таких геометрических объектов, как точки, отрезки или вершины многоугольника в порядке обхода против часовой стрелки. На выходе часто дается ответ на такие запросы об этих объектах, как наличие пересекающихся линий. Могут также выводиться параметры новых геометрических объектов, например, выпуклой оболочки множества точек (это минимальный выпуклый многоугольник, содержащий данное множество).

В этой главе мы ознакомимся с несколькими алгоритмами вычислительной геометрии в двух измерениях, т.е. на плоскости. Каждый входной объект представлен множеством точек $\{p_1, p_2, \ldots\}$, где каждая точка $p_i = (x_i, y_i)$ образована парой действительных чисел $x_i, y_i \in \mathbf{R}$. Например, n-угольник P представлен последовательностью вершин $\langle p_0, p_1, \ldots, p_{n-1} \rangle$ в порядке обхода границы многоугольника. Вычислительная геометрия может работать в трех измерениях, и даже в большем их числе, но визуализация подобных задач и их решений может оказаться очень трудной. Однако даже в двух измерениях можно ознакомиться с хорошими примерами применения методов вычислительной геометрии.

В разделе 33.1 показано, как быстро и точно ответить на такие основные вопросы о расположении отрезков: если два отрезка имеют общую конечную точку, то как следует перемещаться при переходе от одного из них к другому — по часовой стрелке или против часовой стрелки, в каком направлении следует сво-

рачивать при переходе от одного такого отрезка к другому, а также пересекаются ли два отрезка. В разделе 33.2 представлен метод, известный под названием "выметание", или "метод движущейся прямой". Он используется при разработке алгоритма со временем работы $O(n \lg n)$, определяющего, имеются ли пересечения среди n отрезков. В разделе 33.3 приведены два алгоритма "выметания по кругу", предназначенные для вычисления выпуклой оболочки множества n точек (наименьшего содержащего их выпуклого многоугольника): сканирование по Грэхему (Graham's scan), время работы которого равно $O(n \lg n)$, и обход по Джарвису (Jarvis's march), время выполнения которого равно $O(nh)$, где h — количество вершин в оболочке. Наконец, в разделе 33.4 приводится алгоритм декомпозиции со временем работы $O(n \lg n)$, предназначенный для поиска пары ближайших точек в заданном на плоскости множестве из n точек.

33.1 Свойства отрезков

В некоторых представленных в этой главе алгоритмах требуется ответить на вопросы о свойствах отрезков. ***Выпуклой комбинацией*** (convex combination) двух различных точек $p_1 = (x_1, y_1)$ и $p_2 = (x_2, y_2)$ называется любая точка $p_3 = (x_3, y_3)$, такая что для некоторого значения α, принадлежащего интервалу $0 \leqslant \alpha \leqslant 1$, выполняются равенства $x_3 = \alpha x_1 + (1 - \alpha) x_2$ и $y_3 = \alpha y_1 + (1 - \alpha) y_2$. Также пишут, что $p_3 = \alpha p_1 + (1 - \alpha) p_2$. Интуитивно понятно, что в роли p_3 может выступать любая точка, которая принадлежит прямой, соединяющей точки p_1 и p_2, и находится между этими точками. Если заданы две различные точки p_1 и p_2, то ***отрезком прямой*** (line segment) $\overline{p_1 p_2}$ называется множество выпуклых комбинаций p_1 и p_2. Точки p_1 и p_2 называются ***конечными точками*** (endpoints) отрезка $\overline{p_1 p_2}$. Иногда играет роль порядок следования точек p_1 и p_2, и тогда говорят о ***направленном отрезке*** (directed segment) $\overrightarrow{p_1 p_2}$. Если точка p_1 совпадает с ***началом координат*** (origin), т.е. имеет координаты $(0, 0)$, то направленный отрезок $\overrightarrow{p_1 p_2}$ можно рассматривать как ***вектор*** (vector) p_2.

В этом разделе исследуются перечисленные ниже вопросы.

1. Если заданы два направленных отрезка $\overrightarrow{p_0 p_1}$ и $\overrightarrow{p_0 p_2}$, то находится ли отрезок $\overrightarrow{p_0 p_1}$ в направлении по часовой стрелке от отрезка $\overrightarrow{p_0 p_2}$ относительно их общей точки p_0?
2. Если заданы два направленных отрезка $\overline{p_0 p_1}$ и $\overline{p_1 p_2}$, то в какую сторону следует свернуть в точке p_1 при переходе от отрезка $\overline{p_0 p_1}$ к отрезку $\overline{p_1 p_2}$?
3. Пересекаются ли отрезки $\overline{p_1 p_2}$ и $\overline{p_3 p_4}$?

На рассматриваемые точки не накладывается никаких ограничений.

На каждый из этих вопросов можно ответить в течение времени $O(1)$, что не должно вызывать удивления, поскольку объем входных данных, которыми выражается каждый из этих вопросов, равен $O(1)$. Кроме того, в наших методах

используются только операции сложения, вычитания, умножения и сравнения. Не понадобится ни деление, ни вычисление тригонометрических функций, а ведь обе эти операции могут оказаться дорогостоящими и приводить к проблемам, связанным с ошибками округления. Например, метод "в лоб" при определении того, пересекаются ли два отрезка, — найти для каждого из них уравнение прямой в виде $y = mx + b$ (где m — угловой коэффициент, а b — координата пересечения с осью y), вычислить точку пересечения этих прямых и проверить, принадлежит ли эта точка обоим отрезкам. В таком методе при вычислении координат точки пересечения используется деление. Если отрезки почти параллельны, этот метод очень чувствителен к точности операции деления на реальных компьютерах. Изложенный в данном разделе метод, в котором удается избежать операции деления, намного точнее.

Векторное произведение

Вычисление векторного произведения составляет основу методов работы с отрезками. Рассмотрим векторы p_1 и p_2, показанные на рис. 33.1*а*. ***Векторное произведение*** (cross product) $p_1 \times p_2$ можно интерпретировать как значение (со знаком) площади параллелограмма, образованного точками $(0, 0)$, p_1, p_2 и $p_1 + p_2 = = (x_1 + x_2, y_1 + y_2)$. Эквивалентное, но более полезное определение векторного произведения — определитель матрицы[1]:

$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 = -p_2 \times p_1.$$

Если величина $p_1 \times p_2$ положительна, то вектор p_1 находится по часовой стрелке от вектора p_2 относительно начала координат $(0, 0)$; если же векторное произведение отрицательно, то вектор p_1 находится в направлении против часовой стрелки от вектора p_2. (См. упражнение 33.1-1.) На рис. 33.1*б* показаны области, расположенные по и против часовой стрелки относительно вектора p. Граничный случай возникает, когда векторное произведение равно нулю. В этом случае векторы ***коллинеарны*** (collinear), т.е. они направлены в одном и том же или в противоположных направлениях.

Чтобы определить, находится ли направленный отрезок $\overrightarrow{p_0 p_1}$ по часовой стрелке от направленного отрезка $\overrightarrow{p_0 p_2}$ относительно их общей точки p_0, достаточно использовать ее как начало координат. Сначала обозначим величину $p_1 - p_0$ как вектор $p_1' = (x_1', y_1')$, где $x_1' = x_1 - x_0$ и $y_1' = y_1 - y_0$, и введем аналогичные

[1] Фактически векторное произведение — трехмерная концепция. Это вектор, перпендикулярный обоим векторам p_1 и p_2, направление которого определяется "правилом правой руки", а величина равна $|x_1 y_2 - x_2 y_1|$. Однако в этой главе удобнее трактовать векторное произведение как величину $x_1 y_2 - x_2 y_1$.

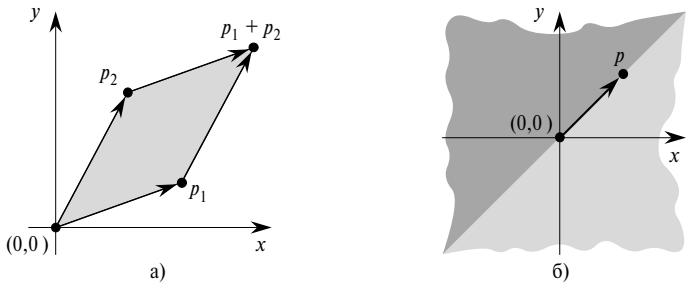


Рис. 33.1. Геометрический смысл векторного произведения

обозначения для величины $p_2 - p_1$. Затем вычислим векторное произведение

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0).$$

Если это векторное произведение положительно, то переход от направленного отрезка $\overrightarrow{p_0p_1}$ к отрезку $\overrightarrow{p_0p_2}$ осуществляется против часовой стрелки; в противном случае он осуществляется по часовой стрелке.

Поворот последовательных отрезков

Следующий вопрос заключается в том, куда сворачивают два последовательных отрезка $\overline{p_0p_1}$ и $\overline{p_1p_2}$ в точке p_1 — влево или вправо. Можно привести эквивалентную формулировку этого вопроса —определить знак угла $\angle p_0p_1p_2$. Векторное произведение позволяет ответить на этот вопрос, не вычисляя величину угла. Как видно из рис. 33.2, производится проверка, находится ли направленный отрезок $\overrightarrow{p_0p_2}$ по часовой стрелке или против часовой стрелки относительно направленного отрезка $\overrightarrow{p_0p_1}$. Для этого вычисляется векторное произведение $(p_2 - p_0) \times (p_1 - p_0)$. Если эта величина отрицательная, то переход от направленного отрезка $\overrightarrow{p_0p_1}$ к направленному отрезку $\overrightarrow{p_0p_2}$ осуществляется против часовой стрелки, и в точке p_1 образуется левый поворот. Положительное значение векторного произведения указывает на ориентацию по часовой стрелке и на правый поворот. Обе эти ситуа-

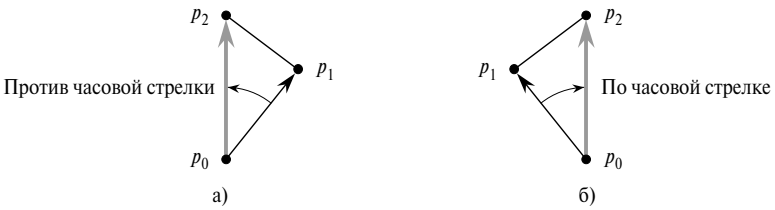


Рис. 33.2. Использование векторного произведения для определения направления поворота последовательных отрезков

ции проиллюстрированы на рис. 33.2. В части *а* показан случай, соответствующий левому повороту, а в части *б* — случай, соответствующий правому повороту. Нулевое векторное произведение означает, что точки p_0, p_1 и p_2 коллинеарны.

Определение того, пересекаются ли два отрезка

Чтобы определить, пересекаются ли два отрезка, следует проверить, пресекает ли каждый из них прямую, содержащую другой отрезок. Отрезок $\overrightarrow{p_1p_2}$ ***пересекает*** (straddles) прямую, если конечные точки отрезка принадлежат разным полуплоскостям, на которые прямая разбивает плоскость. В граничном случае точка p_1 или точка p_2 (или обе эти точки) лежит на прямой. Два отрезка пересекаются тогда и только тогда, когда выполняется одно из сформулированных ниже условий (или оба эти условия).

1. Каждый отрезок пересекает прямую, на которой лежит другой отрезок.
2. Конечная точка одного из отрезков лежит на другом отрезке (граничный случай).

Идея метода реализована в приведенных ниже процедурах. Процедура SEGMENTS_INTERSECT возвращает значение TRUE, если отрезки $\overline{p_1p_2}$ и $\overline{p_3p_4}$ пересекаются, и значение FALSE в противном случае. В этой процедуре вызываются вспомогательные процедуры DIRECTION и ON_SEGMENT. В первой из них с помощью описанного выше векторного произведения определяется относительное расположение отрезков, а во второй — лежит ли на отрезке точка, если известно, что она коллинеарна этому отрезку.

```
SEGMENTS_INTERSECT(p1, p2, p3, p4)
 1  d1 ← DIRECTION(p3, p4, p1)
 2  d2 ← DIRECTION(p3, p4, p2)
 3  d3 ← DIRECTION(p1, p2, p3)
 4  d4 ← DIRECTION(p1, p2, p4)
 5  if ((d1 > 0 и d2 < 0) или (d1 < 0 и d2 > 0)) и
          ((d3 > 0 и d4 < 0) или (d3 < 0 и d4 > 0))
 6     then return TRUE
 7  elseif d1 = 0 и ON_SEGMENT(p3, p4, p1)
 8     then return TRUE
 9  elseif d2 = 0 и ON_SEGMENT(p3, p4, p2)
10     then return TRUE
11  elseif d3 = 0 и ON_SEGMENT(p1, p2, p3)
12     then return TRUE
13  elseif d4 = 0 и ON_SEGMENT(p1, p2, p4)
14     then return TRUE
15  else return FALSE
```

```
DIRECTION(p_i, p_j, p_k)
1  return (p_k − p_i) × (p_j − p_i)

ON_SEGMENT(p_i, p_j, p_k)
1  if min(x_i, x_j) ≤ x_k ≤ max(x_i, x_j) и min(y_i, y_j) ≤ y_k ≤ max(y_i, y_j)
2    then return TRUE
3    else return FALSE
```

Процедура SEGMENTS_INTERSECT работает следующим образом. В строках 1–4 вычисляется относительное расположение d_i каждой конечной точки p_i относительно другого отрезка. Если все величины, характеризующие относительное расположение, отличны от нуля, то легко определить, пересекаются ли отрезки $\overline{p_1p_2}$ и $\overline{p_3p_4}$. Это выполняется следующим образом. Отрезок $\overline{p_1p_2}$ пересекает прямую, содержащую отрезок $\overline{p_3p_4}$, если направленные отрезки $\overrightarrow{p_3p_1}$ и $\overrightarrow{p_3p_2}$ имеют противоположные направления относительно $\overrightarrow{p_3p_4}$. В этом случае знаки величин d_1 и d_2 разные. Аналогично, отрезок $\overline{p_3p_4}$ пересекает прямую, содержащую отрезок $\overline{p_1p_2}$, если знаки величин d_3 и d_4 разные. Если проверка в строке 5 подтверждается, то отрезки пересекаются, и процедура SEGMENTS_INTERSECT возвращает значение TRUE. Этот случай показан на рис. 33.3*а*. В противном случае отрезки не пересекают прямые друг друга, хотя и возможны граничные случаи. Если ни одна из величин, характеризующих взаимную ориентацию отрезков, не равна нулю, то это не граничный случай. При этом не выполняется ни одно из условий на равенство нулю в строках 7–13, и процедура SEGMENTS_INTERSECT возвращает в строке 15 значение FALSE. Этот случай проиллюстрирован на рис. 33.3*б*. Здесь отрезок $\overline{p_3p_4}$ пересекает прямую, содержащую отрезок $\overline{p_1p_2}$, но отрезок $\overline{p_1p_2}$ не пересекает прямую, содержащую отрезок $\overline{p_3p_4}$. При этом знаки векторных произведений $(p_1 - p_3) \times (p_4 - p_3)$ и $(p_2 - p_3) \times (p_4 - p_3)$ совпадают.

Граничный случай возникает, если какая-либо относительная ориентация d_k равна 0. Это говорит о том, что точка p_k коллинеарна с другим отрезком. Данная точка принадлежит другому отрезку тогда и только тогда, когда она находится между его конечными точками. Процедура ON_SEGMENT позволяет определить, расположена ли точка p_k между конечными точками другого отрезка $\overline{p_ip_j}$. Эта процедура вызывается в строках 7–13, и в ней предполагается, что точка p_k коллинеарна отрезку $\overline{p_ip_j}$. В частях *в* и *г* рисунка 33.3 проиллюстрирован случай, когда один из отрезков коллинеарен с конечной точкой другого отрезка. В части *в* точка p_3 коллинеарна с отрезком $\overline{p_1p_2}$ и находится между его конечными точками. В этом случае процедура SEGMENTS_INTERSECT возвращает в строке 12 значение TRUE. В части *г* точка p_3 коллинеарна с отрезком $\overline{p_1p_2}$, но не лежит между его конечными точками. Процедура SEGMENTS_INTERSECT возвращает в строке 15 значение FALSE (отрезки не пересекаются).

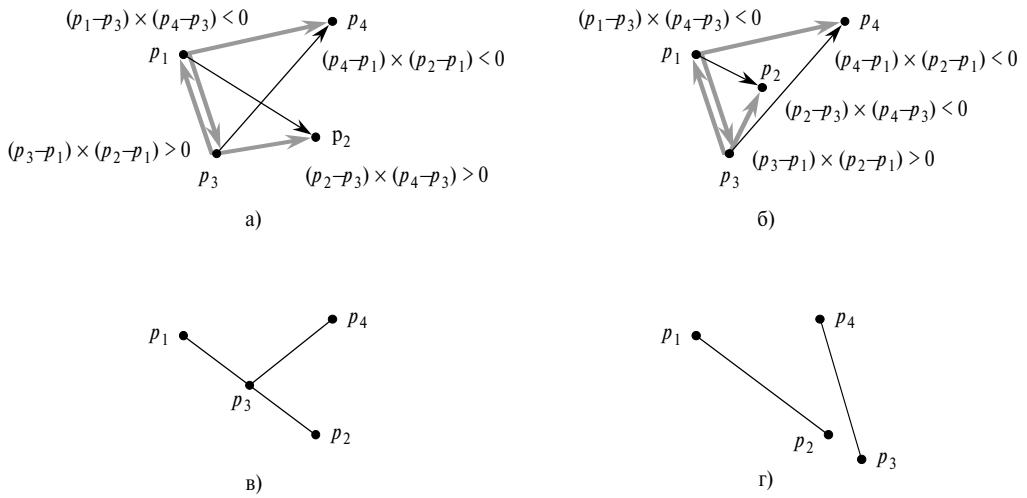


Рис. 33.3. Частные случаи в процедуре SEGMENTS_INTERSECT

Другие применения векторного произведения

В последующих разделах этой главы описаны другие приложения векторного произведения. В разделе 33.3 ставится задача сортировки множества точек по характеризующему их расположение полярным углам относительно заданного начала координат. В упражнении 33.1-3 предлагается показать, что с помощью векторного произведения можно осуществить сравнение в процедуре сортировки. В разделе 33.2 с помощью красно-черных деревьев поддерживается упорядочение отрезков по вертикали. Вместо того, чтобы явным образом поддерживать ключевые значения, в коде, реализующем красно-черное дерево, их сравнение будет заменено векторным произведением. Это позволит определить, какой из двух отрезков, пересекающих заданную прямую, находится над другим.

Упражнения

33.1-1. Докажите, что если произведение $p_1 \times p_2$ положительно, то переход от вектора p_2 к вектору p_1 относительно начала координат $(0, 0)$ осуществляется по часовой стрелке, а если это векторное произведение отрицательно, то переход осуществляется против часовой стрелки.

33.1-2. Профессор предположил, что в строке 1 процедуры ON_SEGMENT достаточно протестировать только измерение x. Покажите, что профессор ошибается.

33.1-3. ***Полярным углом*** (polar angle) точки p_1 относительно начала координат p_0 называется угол между вектором $p_1 - p_0$ и полярной осью. Например,

полярный угол точки $(3, 5)$ относительно точки $(2, 4)$ — это угол вектора $(1, 1)$, составляющий 45° или $\pi/4$ радиан. Полярный угол точки $(3, 3)$ относительно точки $(2, 4)$ — это угол вектора $(1, -1)$, составляющий угол 315° или $7\pi/4$ радиан. Напишите псевдокод, сортирующий последовательность n точек $\langle p_1, p_2, \ldots, p_n\rangle$ по их полярному углу относительно заданной точки p_0. Процедура должна выполняться в течение времени $O(n \lg n)$, и сравнение углов в ней следует выполнять с помощью векторного произведения.

33.1-4. Покажите, как за время $O\left(n^2 \lg n\right)$ определить, содержатся ли в множестве n три коллинеарные точки.

33.1-5. ***Многоугольник*** (polygon) — это кусочно-линейная замкнутая кривая на плоскости. Другими словами, это образованная последовательностью отрезков кривая, начало которой совпадает с ее концом. Эти отрезки называются ***сторонами*** (sides) многоугольника. Точка, соединяющая две последовательных стороны, называется ***вершиной*** (vertex) многоугольника. Если многоугольник ***простой*** (simple), в нем нет самопересечений. (В общем случае предполагается, что мы имеем дело с простыми многоугольниками.) Множество точек плоскости, ограниченное простым многоугольником, образует ***внутреннюю область*** (interior) многоугольника, множество точек, принадлежащих самому многоугольнику, образует его ***границу*** (boundary), а множество точек, окружающих многоугольник, образует его ***внешнюю область*** (exterior). Простой многоугольник называется ***выпуклым*** (convex), если отрезок, соединяющий любые две его граничные или внутренние точки, лежат на границе или во внутренней части этого многоугольника.

Профессор предложил метод, позволяющий определить, являются ли n точек $\langle p_0, p_1, \ldots, p_n\rangle$ последовательными вершинами выпуклого многоугольника. Предложенный алгоритм выводит положительный ответ, если в множестве $\{\angle p_i p_{i+1} p_{i+2} : i = 0, 1, \ldots, n-1\}$, где увеличение индексов выполняется по модулю n, не содержатся одновременно и левые, и правые повороты. В противном случае выводится отрицательный ответ. Покажите, что несмотря на то, что время работы этой процедуры выражается линейной функцией, она не всегда дает правильный ответ. Модифицируйте предложенный профессором метод, чтобы он всегда давал правильный ответ в течение линейного времени.

33.1-6. Пусть задана точка $p_0 = (x_0, y_0)$. ***Правым горизонтальным лучом*** (right horizontal ray) точки p_0 называется множество точек $\{p_i = (x_i, y_i) : x_i \geqslant x_0$ и $y_i = y_0\}$, т.е. множество точек, расположенных строго справа от точки p_0, вместе с самой этой точкой. Покажите, как в течение времени $O(1)$ определить, пересекает ли данный правый горизонтальный

луч из точки p_0 отрезок $\overline{p_1p_2}$. Сведите эту задачу к другой, в которой определяется, пересекаются ли два отрезка.

33.1-7. Один из методов, позволяющих определить, находится ли заданная точка во внутренней области простого, но не обязательного выпуклого многоугольника P, заключается в следующем. Из точки p_0 проводится произвольный луч и определяется, сколько раз он пересекает границу многоугольника P. Если количество пересечений нечетно и сама точка p_0 не лежит на границе многоугольника P, то она лежит во внутренней его части. Покажите, как в течение времени $\Theta(n)$ определить, находится ли точка p_0 во внутренней части n-угольника P. (*Указание*: воспользуйтесь результатами упражнения 33.1-6. Убедитесь в правильности этого алгоритма в том случае, когда луч пересекает границу многоугольника в его вершине, и если луч перекрывается его стороной.)

33.1-8. Покажите, как в течение времени $\Theta(n)$ найти площадь простого, но не обязательно выпуклого n-угольника. (Определения, которые относятся к многоугольникам, можно найти в упражнении 33.1-5.)

33.2 Определение наличия пересекающихся отрезков

В этом разделе представлен алгоритм, позволяющий определить, пересекаются ли какие-нибудь два из отрезков некоторого множества. В этом алгоритме используется метод, известный под названием “выметание”, который часто встречается в алгоритмах вычислительной геометрии. Кроме того, как показано в упражнениях, приведенных в конце этого раздела, с помощью данного алгоритма или его вариации можно решать и другие задачи вычислительной геометрии.

Время работы этого алгоритма равно $O(n \lg n)$, где n — количество заданных отрезков. В нем лишь определяется, существуют пересечения или нет, но не выводятся данные обо всех этих пересечениях. (Согласно результатам упражнения 33.2-1, для поиска *всех* пересечений в множестве, состоящем из n отрезков, в наихудшем случае понадобится время $\Omega(n^2)$.)

В методе ***выметания*** (sweeping) по заданному множеству геометрических объектов проводится воображаемая вертикальная ***выметающая прямая*** (sweep line), которая обычно движется слева направо. Измерение, вдоль которого двигается выметающая прямая (в данном случае это измерение x), трактуется как время. Выметание — это способ упорядочения геометрических объектов путем размещения их параметров в динамической структуре данных, что позволяет воспользоваться взаимоотношениями между этими объектами. В приведенном в этом разделе алгоритме, устанавливающем наличие пересечений отрезков, рассматриваются

все конечные точки отрезков в порядке их расположения слева направо, и для каждой новой точки проверяется наличие пересечений.

Чтобы описать алгоритм и доказать его способность корректно определить, пересекаются ли какие-либо из n отрезков, сделаем два упрощающих предположения. Во-первых, предположим, что ни один из входных отрезков не является вертикальным. Во-вторых, — что никакие три входных отрезка не пересекаются в одной точке. В упражнениях 33.2-8 и 33.2-9 предлагается показать, что алгоритм достаточно надежен для того, чтобы после небольших изменений работать даже в тех случаях, когда сделанные предположения не выполняются. Часто оказывается, что отказ от подобных упрощений и учет граничных условий становится самым сложным этапом программирования алгоритмов вычислительной геометрии и доказательства их корректности.

Упорядочение отрезков

Поскольку предполагается, что вертикальные отрезки отсутствуют, каждый входной отрезок пересекает данную вертикальную выметающую прямую в одной точке. Таким образом, отрезки, пересекающие вертикальную выметающую прямую, можно упорядочить по координате y точки пересечения.

Рассмотрим это подробнее. Пусть имеется два отрезка s_1 и s_2. Говорят, что они ***сравнимы*** (comparable) в координате x, если вертикальная выметающая прямая с координатой x пересекает оба этих отрезка. Говорят также, что отрезок s_1 расположен ***над*** (above) отрезком s_2 в x (записывается $s_1 >_x s_2$), если отрезки s_1 и s_2 сравнимы в координате x и точка пересечения отрезка s_1 с выметающей прямой в координате x находится выше, чем точка пересечения отрезка s_2 с этой выметающей прямой. Например, в ситуации, проиллюстрированной на рис. 33.4*а*, выполняются соотношения $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$ и $b >_u c$. Отрезок d не сравним ни с каким другим отрезком.

Для произвольной заданной координаты x отношение "$>_x$" полностью упорядочивает (см. раздел Б.2) отрезки, пересекающие выметающую прямую в ко-

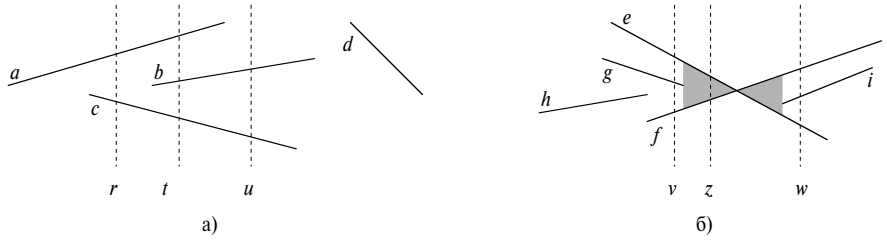


Рис. 33.4. Упорядочение отрезков с помощью разных вертикальных выметающих прямых

ординате x. Однако порядок может зависеть от x, а отрезки по мере изменения x могут попадать в число сравнимых или выходить из него. Отрезок попадает в число сравнимых, если выметающая прямая проходит его левую конечную точку, и выходит из него, когда выметающая прямая проходит его правую конечную точку.

Что происходит, если выметающая прямая проходит через точку пересечения двух отрезков? Как видно из рис. 33.4*б*, меняется порядок этих отрезков. В показанной на рисунке ситуации выметающие прямые v и w находятся слева и справа, соответственно, от точки пересечения отрезков e и f, и справедливы соотношения $e >_v f$ и $f >_w e$. В силу предположения о том, что никакие три отрезка не пересекаются в одной и той же точке, должна существовать некоторая вертикальная выметающая прямая x, для которой пересекающиеся отрезки e и f являются *последовательными* (т.е. между ними нет других отрезков) в полном упорядочении и связаны между собой соотношением $>_x$. Для любой выметающей прямой, проходящей через затененную область на рис. 33.4*б*, например, для прямой z отрезки e и f являются последовательными.

Перемещение выметающей прямой

В выметающих алгоритмах обычно обрабатываются два набора данных.

1. ***Состояние относительно выметающей прямой*** (sweep-line status) описывает соотношения между объектами, пересекаемыми выметающей прямой.
2. ***Таблица точек-событий*** (event-point schedule) — это последовательность координат x, упорядоченных слева направо, в которой определяются точки останова выметающей прямой. Каждая такая точка останова называется ***точкой-событием*** (event point). Состояние относительно выметающей прямой определяется только в точках-событиях.

В некоторых алгоритмах (например, в том, который предлагается разработать в упражнении 33.2-7) таблица точек-событий строится динамически, в ходе работы алгоритма. Однако в алгоритме, о котором сейчас пойдет речь, точки-события определяются статически, исходя исключительно из простых свойств входных данных. В частности, в роли точки-события выступает каждая конечная точка отрезка. Конечные точки отрезков сортируются в порядке возрастания координат x (т.е. слева направо). (Если две или более точек ***ковертикальны*** (covertical), т.е. их координаты x совпадают, этот узел можно "разрубить", поместив все ковертикальные левые конечные точки перед ковертикальными правыми конечными точками. Если в множестве ковертикальных левых точек есть несколько левых, первыми среди них будут те, которые имеют меньшее значение координаты y, и то же самое выполняется в множестве ковертикальных правых конечных точек.) Отрезок помещается в набор состояний относительно выметающей прямой,

когда пересекается его левая конечная точка; этот отрезок удаляется из набора состояний относительно выметающей прямой, когда пересекается его правая конечная точка. Если два отрезка впервые становятся последовательными в полном упорядочении, выполняется проверка, пересекаются ли они.

Состояние относительно выметающей прямой является полностью упорядоченным множеством T, в котором необходимо реализовать такие операции:

- INSERT(T, s): вставка отрезка s в множество T;
- DELETE(T, s): удаление отрезка s из множества T;
- ABOVE(T, s): возвращает отрезок, расположенный непосредственно над отрезком s множества T;
- BELOW(T, s): возвращает отрезок, расположенный непосредственно под отрезком s множества T.

Если всего имеется n входных отрезков, каждую из перечисленных выше операций с помощью красно-черных деревьев можно выполнить в течение времени $O(\lg n)$. Напомним, что операции над красно-черными деревьями, описанные в главе 13, включают в себя сравнение ключей. Однако в нашем случае сравнение ключей можно заменить векторным произведением, позволяющем определить относительный порядок двух отрезков (см. упражнение 33.2-2).

Псевдокод, выявляющий пересечение отрезков

В приведенном ниже алгоритме в качестве входных данных выступает множество S, состоящее из n отрезков. На выходе возвращается булево значение TRUE, если любая пара отрезков из множества S пересекается, и значение FALSE в противном случае. Полное упорядочение T реализуется с помощью красно-черного дерева.

```
ANY_SEGMENTS_INTERSECT(S)
 1  T ← 0
 2  Сортировка конечных точек отрезков из множества S слева
        направо, разрешение совпадений путем помещения левых
        конечных точек перед правыми и путем помещения точек
        с меньшими координатами y перед теми, у которых
        координата y больше
 3  for  каждой точки p из отсортированного списка конечных точек
 4       do if  p — левая конечная точка отрезка s
 5             then INSERT(T, s)
 6                  if (ABOVE(T, s) существует и пересекает отрезок s)
                           или (BELOW(T, s) существует и пересекает отрезок s)
 7                    then return TRUE
```

```
 8        if  p — правая конечная точка отрезка s
 9          then if существуют ABOVE(T, s) и BELOW(T, s),
                        и ABOVE(T, s) пересекает BELOW(T, s)
10                 then return TRUE
11               DELETE(T, s)
12  return FALSE
```

Работа этого алгоритма проиллюстрирована на рис. 33.5. Каждая пунктирная линия является выметающей прямой, проходящей через одну из точек-событий. Под каждой из них приведено полностью упорядоченное множество T в момент окончания итерации цикла **for**, соответствующей данному событию. В момент удаления отрезка c обнаружено пересечение отрезков d и b. В строке 1 инициализируется пустое множество полностью упорядоченных отрезков. В строке 2 путем сортировки $2n$ конечных точек отрезков слева направо создается таблица точек-событий, а ковертикальные точки обрабатываются так, как описано выше. Заметим, что строку 2 можно выполнить путем лексикографической сортировки конечных точек (x, e, y), где x и y — обычные координаты, а значение переменной e принимает значение 0 для левых конечных точек и значение 1 — для правых конечных точек.

При каждой итерации цикла **for** в строках 3–11 обрабатывается одно событие p. Если p — левая конечная точка отрезка s, то этот отрезок добавляется в строке 5 в полностью упорядоченное множество. В строках 6–7 возвращается значение TRUE, если отрезок s пересекает какой-нибудь из соседних отрезков полностью упорядоченного множества, определенного выметающей прямой, которая проходит через точку p. (Если точка p находится на другом отрезке s', то имеет место граничный случай. При этом лишь требуется, чтобы отрезки s и s' были после-

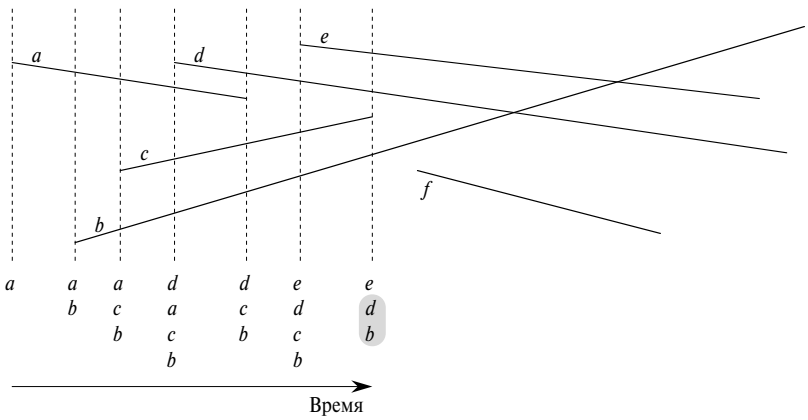


Рис. 33.5. Выполнение процедуры ANY_SEGMENTS_INTERSECT

довательными в множестве T.) Если p — правая конечная точка отрезка s, то этот отрезок подлежит удалению из полностью упорядоченного множества. В строках 9–10 возвращается значение TRUE, если пересекаются ближайшие к отрезку s отрезки из полностью упорядоченного множества, определенного с помощью выметающей прямой, которая проходит через точку p. После удаления из полностью упорядоченного множества отрезка s ближайшие к нему отрезки становятся последовательными в этом множестве. Если эти отрезки не пересекаются, отрезок s в строке 11 удаляется из полностью упорядоченного множества. Наконец, если во всех $2n$ точках-событиях не обнаружено никаких пересечений, в строке 12 возвращается значение FALSE.

Корректность

Чтобы показать, что процедура ANY_SEGMENTS_INTERSECT работает корректно, докажем, что при вызове ANY_SEGMENTS_INTERSECT(S) значение TRUE возвращается тогда и только тогда, когда какие-либо отрезки из S пересекаются.

Очевидно, что процедура ANY_SEGMENTS_INTERSECT возвращает значение TRUE (в строках 7 и 10) только в том случае, когда в ней будет обнаружено пересечение между двумя входными отрезками. Таким образом, если возвращается значение TRUE, то пересечение существует.

Следует также показать обратное: если пересечение существует, то процедура ANY_SEGMENTS_INTERSECT возвращает значение TRUE. Предположим, что существует по крайней мере одно пересечение. Пусть p — самая левая точка пересечения (если таких точек несколько, то выбирается та из них, у которой наименьшая координата y), а a и b — отрезки, пересекающиеся в этой точке. Поскольку слева от точки p нет никаких пересечений, порядок, заданный в множестве T, остается правильным во всех точках, которые находятся слева от точки p. Так как никакие три отрезка не пересекаются в одной и той же точке, существует выметающая прямая z, относительно которой отрезки a и b расположены последовательно в полностью упорядоченном множестве[2]. Более того, прямая z проходит через точку p или слева от нее. На выметающей прямой z находится конечная точка q какого-нибудь отрезка, которая выступает в роли события, при котором отрезки a и b становятся последовательными в полностью упорядоченном множестве. Если на этой выметающей прямой лежит точка p, то $q = p$. Если же точка p не принадлежит выметающей прямой z, то точка q находится слева от точки p. В любом случае порядок, установленный в множестве T, является

[2]Если позволить трем отрезкам пересекаться в одной точке, появится возможность существования промежуточного отрезка c, пересекающего отрезки a и b в точке p. Т.е. для всех выметающих прямых w, расположенных слева от точки p, для которых $a <_w b$, могут выполняться соотношения $a <_w c$ и $c <_w b$. В упражнении 33.2-8 предлагается показать, что процедура ANY_SEGMENTS_INTERSECT работает корректно, даже если три отрезка пересекаются в одной и той же точке.

правильным непосредственно перед точкой q. (Вот где используется лексикографический порядок, в котором алгоритм обрабатывает конечные точки. Поскольку p — нижайшая из самых левых точек пересечения, то даже если она лежит на выметающей прямой z и на этой прямой есть еще одна точка пересечения p', точка-событие $q = p$ обрабатывается перед тем, как эта другая точка пересечения сможет повлиять на порядок отрезков в упорядоченном множестве T. Кроме того, если p — левая конечная точка одного отрезка (скажем, отрезка a) и правая конечная точка другого отрезка (скажем, отрезка b), то в силу того, что левые точки-события расположены перед правыми, отрезок b уже будет находиться в множестве T, когда в это множество попадет отрезок a.) Каждая точка-событие q либо обрабатывается процедурой ANY_SEGMENTS_INTERSECT, либо нет.

Если точка q обрабатывается процедурой ANY_SEGMENTS_INTERSECT, возможны всего два варианта выполняемых действий.

1. Отрезок a либо отрезок b помещается в множество T, и в этом множестве один из них расположен над или под другим. То, какой из этих случаев имеет место, определяется в строках 4–7.
2. Отрезки a и b уже содержатся в множестве T, а расположенный между ними отрезок удаляется из этого множества, в результате чего отрезки a и b становятся последовательными. Этот случай выявляется в строках 8–11.

При реализации любой из перечисленных выше возможностей выявляется точка пересечения p, и процедура ANY_SEGMENTS_INTERSECT возвращает значение TRUE.

Если точка q не обрабатывается процедурой ANY_SEGMENTS_INTERSECT, это означает, что произошел выход из процедуры до обработки всех точек-событий. Это может произойти только в том случае, когда в этой процедуре уже была обнаружена точка пересечения и процедурой возвращено значение TRUE.

Таким образом, если отрезки пересекаются, то процедура ANY_SEGMENTS_INTERSECT возвращает значение TRUE. Как мы уже убедились, если процедура ANY_SEGMENTS_INTERSECT возвращает значение TRUE, отрезки пересекаются. Поэтому рассматриваемая процедура всегда выдает правильный ответ.

Время работы

Если в множестве S содержится n отрезков, то процедура ANY_SEGMENTS_INTERSECT выполняется в течение времени $O(n \lg n)$. Выполнение строки 1 занимает время $O(1)$. Строка 2 с помощью сортировки слиянием или пирамидальной сортировки выполняется в течение времени $O(n \lg n)$. Поскольку всего имеется $2n$ точек-событий, цикл **for** в строках 3–11 повторяется не более $2n$ раз. На каждую итерацию требуется время $O(\lg n)$, поскольку выполнение каждой операции в красно-черном дереве занимает время $O(\lg n)$, и с помощью метода, описанного

в разделе 33.1, каждый тест на пересечение удается выполнить за время $O(1)$. Таким образом, полное время равно $O(n \lg n)$.

Упражнения

33.2-1. Покажите, что в множестве, состоящем из n отрезков, может быть $\Theta\left(n^2\right)$ пересечений.

33.2-2. Пусть отрезки a и b сравнимы в координате x. Покажите, как за время $O(1)$ определить, какое из соотношений выполняется — $a >_x b$ или $b >_x a$. Предполагается, что вертикальные отрезки отсутствуют. (*Указание*: если отрезки a и b не пересекаются, достаточно просто воспользоваться векторным произведением. Если же эти отрезки пересекаются, что выявляется путем вычисления векторных произведений, то и в этом случае можно ограничиться только операциями сложения, вычитания и умножения, и избежать деления. Если отрезки a и b пересекаются, достаточно просто вывести сообщение, что обнаружено пересечение.)

33.2-3. Профессор предложил модифицировать процедуру ANY_SEGMENTS_INTERSECT таким образом, чтобы она не прекращала работу после того, как будет обнаружено пересечение, а выводила пересекающиеся отрезки и переходила к выполнению следующей итерации цикла **for**. Профессор назвал получившуюся в результате процедуру PRINT_INTERSECTING_SEGMENTS и заявил, что она выводит все пересечения, следующие слева направо в том порядке, в котором они располагаются в множестве отрезков. Покажите, что профессор ошибается в двух аспектах. Для этого приведите пример множества отрезков, для которых первая точка пересечения, обнаруженная процедурой PRINT_INTERSECTING_SEGMENTS, не является самой левой, а также пример множества отрезков, для которых эта процедура выявляет не все пересечения.

33.2-4. Разработайте алгоритм, позволяющий в течение времени $O(n \lg n)$ определить, является ли n-угольник простым.

33.2-5. Разработайте алгоритм, позволяющий в течение времени $O(n \lg n)$ определить, пересекаются ли два простых многоугольника, суммарное количество вершин в которых равно n.

33.2-6. ***Круг*** (disk) состоит из окружности и ее внутренней области, и его можно представить при помощи центра и радиуса. Два круга пересекаются, если у них есть хоть одна общая точка. Приведите алгоритм, позволяющий в течение времени $O(n \lg n)$ определить, пересекаются ли какие-либо два круга из множества, состоящего из n кругов.

33.2-7. Пусть задано множество, состоящее из n отрезков, между которыми имеется k пересечений. Покажите, как вывести данные по всем пересечениям в течение времени $O((n+k)\lg n)$.

33.2-8. Покажите, что процедура ANY_SEGMENTS_INTERSECT работает корректно даже в том случае, если в одной и той же точке пересекается три или больше отрезков.

33.2-9. Покажите, что процедура ANY_SEGMENTS_INTERSECT работает корректно даже в том случае, если в числе ее входных отрезков есть вертикальные. При этом нижние конечные точки вертикальных отрезков обрабатываются как левые конечные точки, а верхние — как правые конечные точки. Как изменится ответ в упражнении 33.2-2, если допускается наличие вертикальных отрезков?

33.3 Построение выпуклой оболочки

Выпуклой оболочкой (convex hull) множества точек Q называется наименьший выпуклый многоугольник P, такой что каждая точка из Q находится либо на границе многоугольника P, либо в его внутренней области. (Точное определение выпуклого многоугольника можно найти в упражнении 33.1-5.) Обозначим выпуклую оболочку множества Q как $\mathrm{CH}(Q)$. Интуитивно можно представлять каждую точку множества Q в виде торчащего из доски гвоздя; тогда выпуклая оболочка будет иметь форму, полученную в результате наматывания на гвозди тугой резиновой нити. Пример множества точек и их выпуклой оболочки приведен на рис. 33.6.

В этом разделе представлены два алгоритма, позволяющие построить выпуклую оболочку множества, состоящего из n точек. Оба алгоритма выводят верши-

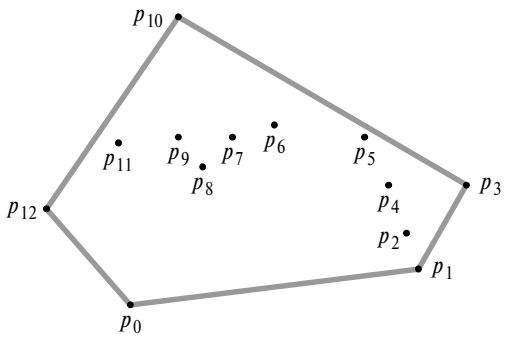


Рис. 33.6. Множество точек $Q = \{p_0, p_1, \ldots, p_{12}\}$ и их выпуклая оболочка $\mathrm{CH}(Q)$

ны выпуклой оболочки в порядке обхода против часовой стрелки. Время работы первого из них, известного как сканирование по Грэхему, равно $O(n \lg n)$. Второй алгоритм, который называется обходом по Джарвису, выполняется в течение времени $O(nh)$, где h — количество вершин выпуклой оболочки. Как видно из рис. 33.6, каждая вершина $\mathrm{CH}(Q)$ — это точка из множества Q. Это свойство используется в обоих алгоритмах. В них принимается решение, какие точки из множества Q выступают в роли вершин выпуклой оболочки, а какие следует отбросить.

Фактически имеется несколько методов построения выпуклой оболочки в течение времени $O(n \lg n)$. И при сканировании по Грэхему, и при обходе по Джарвису используется метод, который называется "выметанием по кругу". Вершины обрабатываются в порядке возрастания их полярных углов, которые они образуют с некоторой базисной вершиной. В число других методов входят те, что перечислены ниже.

- В ***методе последовательного счета*** (incremental method) точки сортируются слева направо, в результате чего получается последовательность $\langle p_1, p_2, \ldots, p_n \rangle$. На i-м этапе выпуклая оболочка $i-1$ самых левых точек $\mathrm{CH}(\{p_1, p_2, \ldots, p_{i-1}\})$ обновляется в соответствии с положением i-й слева точки, после чего формируется оболочка $\mathrm{CH}(\{p_1, p_2, \ldots, p_i\})$. В упражнении 33.3-6 предлагается показать, что этот метод можно реализовать так, чтобы время его работы было равно $O(n \lg n)$.
- В ***методе декомпозиции*** (devide-and-conque method) множество n точек за время $\Theta(n)$ разбивается на два подмножества, в одном из которых содержится $\lceil n/2 \rceil$ самых левых точек, а во втором — $\lfloor n/2 \rfloor$ самых правых точек. Затем рекурсивно строятся выпуклые оболочки этих подмножеств, которые затем объединяются с помощью одного остроумного метода в течение времени $O(n)$. Время работы этого метода описывается уже знакомым рекуррентным соотношением $T(n) = 2T(n/2) + O(n)$, решение которого равно $O(n \lg n)$.
- ***Метод отсечения и поиска*** (prune-and-search method) подобен алгоритму, описанному в разделе 9.3, время работы которого в наихудшем случае ведет себя линейно. В нем строится верхняя часть (или "верхняя цепь") выпуклой оболочки путем повторного отбрасывания фиксированной части оставшихся точек до тех пор, пока не останется только верхняя цепь выпуклой оболочки. Затем то же самое выполняется с нижней цепью. В асимптотическом пределе этот метод самый быстрый: если выпуклая оболочка содержит h вершин, время его работы равно $O(n \lg h)$.

Построение выпуклой оболочки множества точек — задача, интересная сама по себе. Кроме того, алгоритмы, предназначенные для решения некоторых других

задач вычислительной геометрии, начинают работу с построения выпуклой оболочки. Например, рассмотрим двумерную задачу о ***поиске пары самых дальних точек*** (farthest-pair problem): в заданном на плоскости множестве n точек требуется найти две, расстояние между которыми является максимальным. В упражнении 33.3-3 предлагается доказать, что обе эти точки должны быть вершинами выпуклой оболочки. Пару самых дальних вершин выпуклого n-угольника можно найти в течение времени $O(n)$, хотя здесь мы не станем доказывать это утверждение. Таким образом, путем построения выпуклой оболочки n входных точек за время $O(n \lg n)$ и последующего поиска пары самых дальних точек среди вершин полученного в результате выпуклого многоугольника, можно найти пару самых дальних точек из произвольного множества n точек за время $O(n \lg n)$.

Сканирование по Грэхему

В методе ***сканирования по Грэхему*** (Graham's scan) задача о выпуклой оболочке решается с помощью стека S, сформированного из точек-кандидатов. Все точки входного множества Q заносятся в стек, а потом точки, не являющиеся вершинами $\mathrm{CH}(Q)$, со временем удаляются из него. По завершении работы алгоритма в стеке S остаются только вершины оболочки $\mathrm{CH}(Q)$ в порядке их обхода против часовой стрелки.

В качестве входных данных процедуры GRAHAM_SCAN выступает множество точек Q, где $|Q| \geqslant 3$. В ней вызывается функция TOP(S), которая возвращает точку, находящуюся на вершине стека S, не изменяя при этом его содержимое. Кроме того, используется также функция NEXT_TO_TOP(S), которая возвращает точку, расположенную в стеке S на одну позицию ниже от верхней точки; стек S при этом не изменяется. Вскоре будет показано, что стек S, возвращаемый процедурой GRAHAM_SCAN, содержит только вершины $\mathrm{CH}(Q)$, причем расположенные в порядке обхода против часовой стрелки, если просматривать их в стеке снизу вверх.

```
GRAHAM_SCAN(Q)
1  Пусть p_0 — точка из множества Q с минимальной координатой y
       или самая левая из таких точек при наличии совпадений
2  Пусть ⟨p_1, p_2, ..., p_m⟩ — остальные точки множества Q,
       отсортированные в порядке возрастания полярного угла,
       измеряемого против часовой стрелки относительно точки p_0
       (если полярные углы нескольких точек совпадают, то из множества
       удаляются все эти точки, кроме одной, самой дальней от точки p_0)
3  PUSH(p_0, S)
4  PUSH(p_1, S)
5  PUSH(p_2, S)
6  for i ← 3 to m
```

```
 7      do while (пока) угол, образованный точками
                  NEXT_TO_TOP(S), TOP(S) и p_i, образует
                  не левый поворот (при движении по ломаной,
                  образованной этими точками, мы движемся
                  прямо или вправо)
 8          do POP(S)
 9       PUSH(p_i, S)
10  return S
```

Работа процедуры GRAHAM_SCAN проиллюстрирована на рис. 33.7. Выпуклая оболочка, содержащаяся на данный момент в стеке S, на каждом этапе показана серым цветом. В строке 1 процедуры выбирается точка p_0 с минимальной координатой y; при наличии нескольких таких точек выбирается крайняя левая. Поскольку в множестве Q отсутствуют точки, расположенные ниже точки p_0, и все другие точки с такой же координатой y расположены справа от точки p_0, p_0 — вершина оболочки $\mathrm{CH}(Q)$. В строке 2 остальные точки множества Q сортируются в порядке возрастания их полярных углов относительно точки p_0. При этом используется тот же метод, что и в упражнении 33.1-3, т.е. сравнение векторных произведений. Если полярные углы двух или нескольких точек относительно точки p_0 совпадают, все такие точки, кроме самой удаленной от точки p_0, являются выпуклыми комбинациями точки p_0 и самой удаленной от нее из числа таких точек, поэтому все они исключаются из рассмотрения. Обозначим через m количество оставшихся точек, отличных от точки p_0. Величина полярного угла каждой из точек множества Q, измеряемого в радианах относительно точки p_0, принадлежит полуоткрытому интервалу $[0, \pi)$. Поскольку точки отсортированы по величине возрастания их полярных углов, они расположены относительно точки p_0 в порядке обхода против часовой стрелки. Обозначим эту упорядоченную последовательность точек как $\langle p_1, p_2, \ldots, p_m \rangle$. Заметим, что точки p_1 и p_m являются вершинами оболочки $\mathrm{CH}(Q)$ (см. упражнение 33.3-1). На рис. 33.7*а* изображены точки из рис. 33.6, последовательно пронумерованные в порядке возрастания полярных углов относительно точки p_0.

В остальной части процедуры используется стек S. В строках 3–5 стек инициализируется, и в него заносятся снизу вверх три точки p_0, p_1 и p_2. На рис. 33.7*а* показано начальное состояние стека S. В каждой итерации цикла **for** в строках 6–9 обрабатывается очередная точка подпоследовательности $\langle p_3, p_4, \ldots, p_m \rangle$. Цель этой обработки — сделать так, чтобы в результате в стеке S в порядке размещения снизу вверх содержались вершины оболочки $\mathrm{CH}(\{p_0, p_1, \ldots, p_i\})$ в порядке обхода против часовой стрелки. В описанном в строках 7–8 цикле **while** точки удаляются из стека, если будет обнаружено, что они не являются вершинами выпуклой оболочки. При обходе выпуклой оболочки против часовой стрелки в каждой вершине должен производиться поворот влево. Каждый раз, когда в цикле

while встречается вершина, в которой не производится поворот влево, такая вершина снимается со стека. (Выполняется именно проверка того, что поворот — не левый, а не проверка того, что поворот правый. Такая проверка исключает наличие развернутого угла в полученной в результате выпуклой оболочке. Развернутых углов быть не должно, поскольку ни одна из вершин выпуклого многоугольника не может быть выпуклой комбинацией других вершин этого многоугольника.) После снятия со стека всех вершин, в которых при переходе к точке p_i не производится левый поворот, в стек заносится точка p_i. На рис. 33.7*б–л* показано состояние стека S после каждой итерации цикла **for**. По окончании работы в строке 10 процедуры GRAHAM_SCAN возвращается стек S. На рис. 33.7*м* показана полученная выпуклая оболочка.

В сформулированной ниже теореме формально доказывается корректность процедуры GRAHAM_SCAN.

Теорема 33.1 (Корректность сканирования по Грэхему). Если процедура GRAHAM_SCAN обрабатывает множество точек Q, где $|Q| \geqslant 3$, то по завершении этой процедуры стек S будет содержать (в направлении снизу вверх) только вершины оболочки $\mathrm{CH}(Q)$ в порядке обхода против часовой стрелки.

Доказательство. После выполнения строки 2 в нашем распоряжении имеется последовательность точек $\langle p_1, p_2, \ldots, p_m \rangle$. Определим подмножество точек $Q_i = \{p_0, p_1, \ldots, p_i\}$ при $i = 2, 3, \ldots, m$. Множество точек $Q - Q_m$ образуют те из них, что были удалены из-за того, что их полярный угол относительно точки p_0 совпадает с полярным углом некоторой точки из множества Q_m. Эти точки не принадлежат выпуклой оболочке $\mathrm{CH}(Q)$, так что $\mathrm{CH}(Q_m) = \mathrm{CH}(Q)$. Таким образом, достаточно показать, что по завершении процедуры GRAHAM_SCAN стек S состоит из вершин оболочки $\mathrm{CH}(Q_m)$ в порядке обхода против часовой стрелки, если эти точки просматриваются в стеке снизу вверх. Заметим, что точно так же, как точки p_0, p_1, p_m являются вершинами оболочки $\mathrm{CH}(Q)$, точки p_0, p_1, p_i являются вершинами оболочки $\mathrm{CH}(Q_i)$.

В доказательстве используется сформулированный ниже инвариант цикла.

> В начале каждой итерации цикла **for** в строках 6–9 стек S состоит (снизу вверх) только из вершин оболочки $\mathrm{CH}(Q_{i-1})$ в порядке их обхода против часовой стрелки.

Инициализация. При первом выполнении строки 6 инвариант поддерживается, поскольку в этот момент стек S состоит только из вершин $Q_2 = Q_{i-1}$, и это множество трех вершин формирует свою собственную выпуклую оболочку. Кроме того, если просматривать точки снизу вверх, то они будут расположены в порядке обхода против часовой стрелки.

Сохранение. При входе в новую итерацию цикла **for** вверху стека S находится точка p_{i-1}, помещенная туда в конце предыдущей итерации (или перед

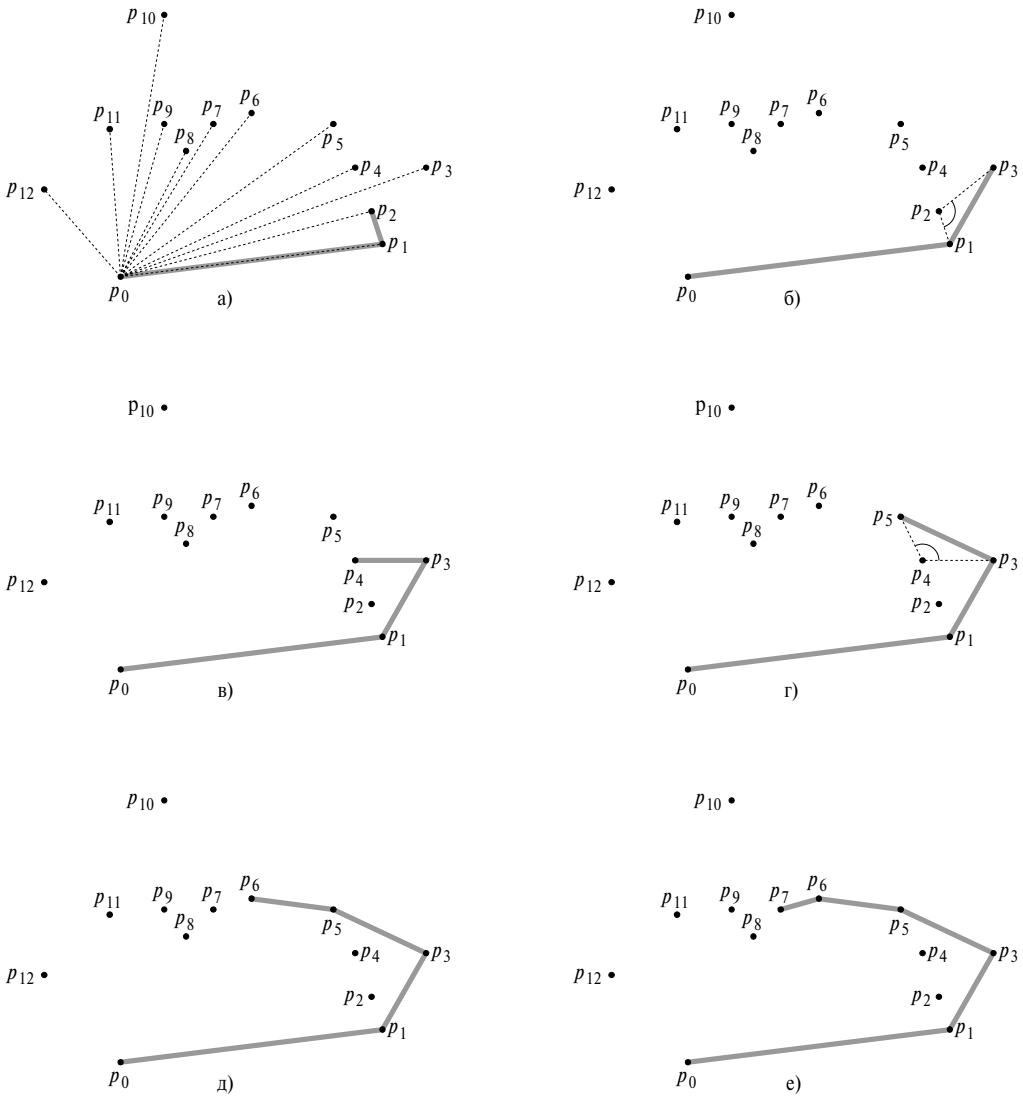
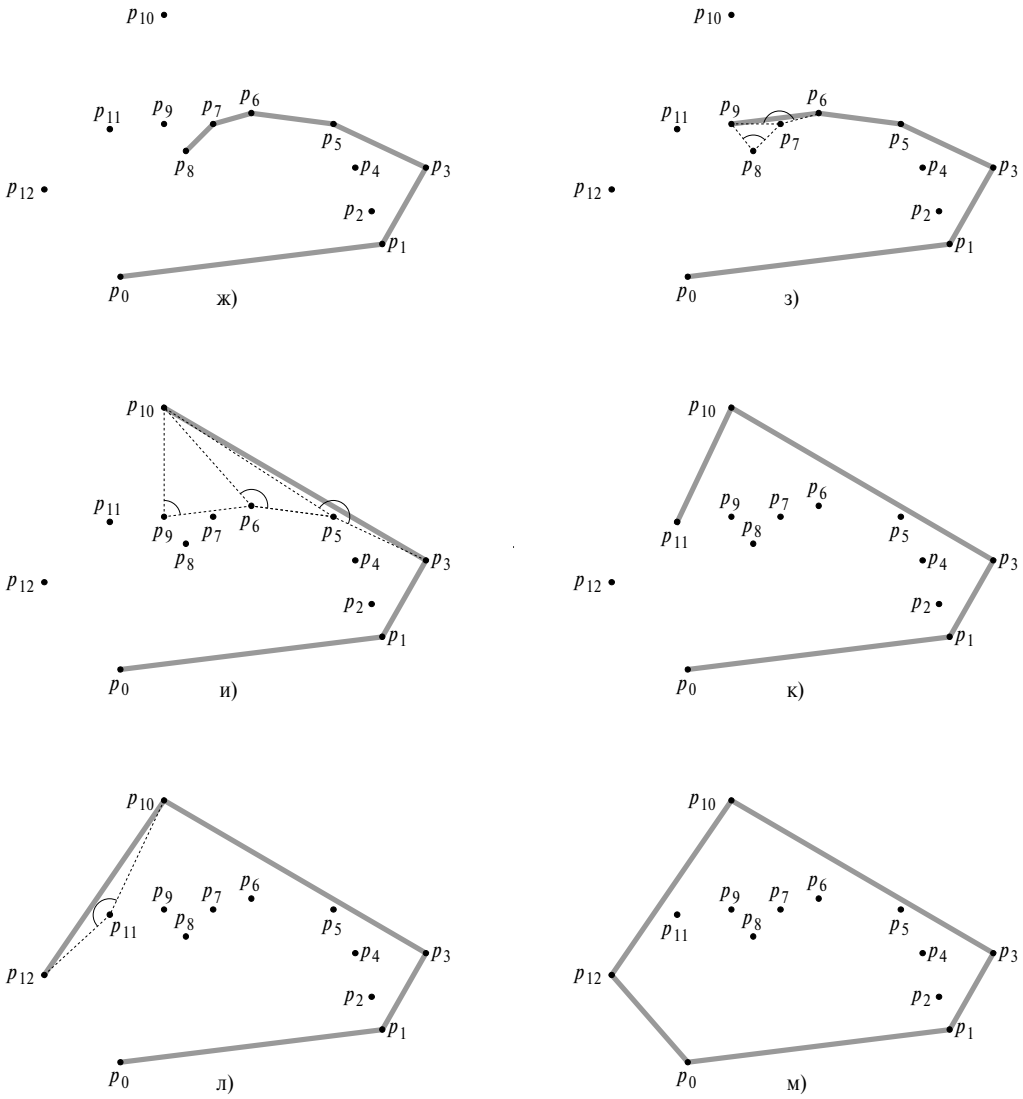


Рис. 33.7. Обработка процедурой GRAHAM_SCAN множества точек Q, показанного на рис. 33.6

первой итерацией, когда $i = 3$). Пусть p_j — верхняя точка стека S после выполнения строк 7–8 цикла **while**, но перед тем, как в строке 9 в стек будет помещена точка p_i. Пусть также p_k — точка, расположенная в стеке S непосредственно под точкой p_j. В тот момент, когда точка p_j находится наверху стека S, а точка p_i еще туда не добавлена, стек содержит те же точки, что и после j-й итерации цикла **for**. Поэтому, согласно инварианту



цикла, в этот момент стек S содержит только вершины $\mathrm{CH}(Q_i)$ в порядке их обхода против часовой стрелки, если просматривать их снизу вверх.

Продолжим размышлять о состоянии стека S непосредственно перед тем, как в него будет добавлена точка p_i. Обратимся к рис. 33.8*а*. Поскольку полярный угол точки p_i относительно точки p_0 больше, чем полярный угол точки p_j, и поскольку угол $\angle p_k p_j p_i$ сворачивает влево (в противном случае точка p_j была бы снята со стека), после добавления в стек S точки p_i (до этого там были только вершины $\mathrm{CH}(Q_j)$) в нем будут содержаться вершины

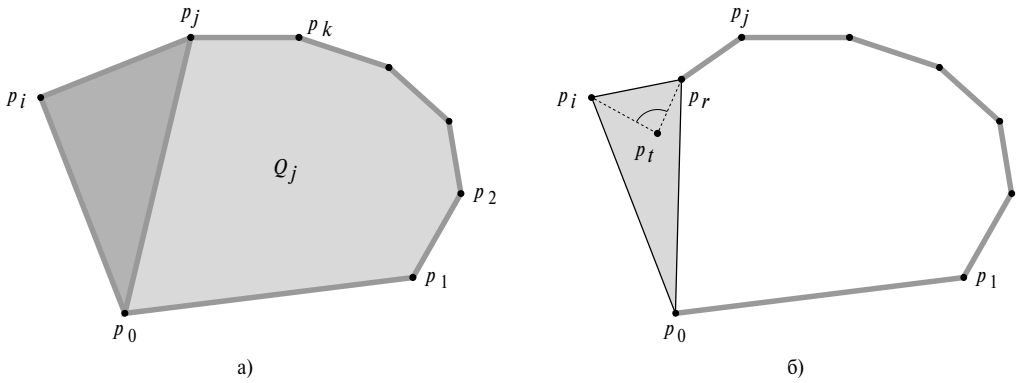


Рис. 33.8. Доказательство корректности процедуры GRAHAM_SCAN

$\mathrm{CH}(Q_j \cup \{p_i\})$. При этом они будут расположены в порядке обхода против часовой стрелки, если просматривать их снизу вверх.

Теперь покажем, что множество вершин $\mathrm{CH}(Q_j \cup \{p_i\})$ совпадает с множеством точек $\mathrm{CH}(Q_i)$. Рассмотрим произвольную точку p_t, снятую со стека во время выполнения i-й итерации цикла **for**, и пусть p_r — точка, расположенная в стеке S непосредственно под точкой p_t перед снятием со стека последней (этой точкой p_r может быть точка p_j). Угол $\angle p_r p_t p_i$ не сворачивает влево, и полярный угол точки p_t относительно точки p_0 больше полярного угла точки p_r. Как видно из рис. 33.8*б*, точка p_t должна располагаться либо внутри треугольника, образованного точками p_0, p_r и p_i, либо на стороне этого треугольника (но не совпадать с его вершиной). Очевидно, что поскольку точка p_t находится внутри треугольника, образованного тремя другими точками множества Q_i, она не может быть вершиной $\mathrm{CH}(Q_i)$. Так как p_t не является вершиной $\mathrm{CH}(Q_i)$, можно записать соотношение

$$\mathrm{CH}(Q_i - \{p_t\}) = \mathrm{CH}(Q_i)\,. \tag{33.1}$$

Пусть P_i — множество точек, снятых со стека во время выполнения i-й итерации цикла **for**. Поскольку равенство (33.1) применимо ко всем точкам множества P_i, в результате его многократного применения можно показать, что выполняется равенство $\mathrm{CH}(Q_i - P_i) = \mathrm{CH}(Q_i)$. Однако $Q_i - P_i = Q_j \cup \{p_i\}$, поэтому мы приходим к заключению, что $\mathrm{CH}(Q_j \cup \{p_i\}) = \mathrm{CH}(Q_i - P_i) = \mathrm{CH}(Q_i)$.

Мы показали, что сразу после вытеснения из стека S точки p_i в нем содержатся только вершины $\mathrm{CH}(Q_i)$ в порядке их обхода против часовой стрелки, если просматривать их в стеке снизу вверх. Последующее увеличение на единицу значения переменной i приведет к сохранению инварианта цикла в очередной итерации.

Завершение. По завершении цикла выполняется равенство $i = m+1$, поэтому из инварианта цикла следует, что стек S состоит только из вершин $\mathrm{CH}(Q_m)$, т.е. из вершин $\mathrm{CH}(Q)$. Эти вершины расположены в порядке обхода против часовой стрелки, если они просматриваются в стеке снизу вверх. На этом доказательство завершается. ■

Теперь покажем, что время работы процедуры GRAHAM_SCAN равно $O(n \lg n)$, где $n = |Q|$. Выполнение строки 1 занимает время $\Theta(n)$. Для сортировки полярных углов методом слияния или пирамидальной сортировки, а также сравнения углов с помощью векторного произведения, как описано в разделе 33.1, в строке 2 требуется время $O(n \lg n)$. (Все точки с одним и тем же значением полярного угла, кроме самой удаленной, можно удалить в течение времени $O(n)$.) Выполнение строк 3–5 занимает время $O(1)$. В силу неравенства $m \leqslant n-1$ цикл **for** в строках 6–9 выполняется не более $n-3$ раз. Так как выполнение процедуры PUSH занимает время $O(1)$, на выполнение каждой итерации требуется время $O(1)$, если не принимать во внимание время, затраченное на выполнение цикла **while** в строках 7–8. Таким образом, общее время выполнения цикла **for** равно $O(n)$, за исключением времени выполнения вложенного в него цикла **while**.

Покажем с помощью группового анализа, что для выполнения всего цикла **while** потребуется время $O(n)$. При $i = 0, 1, \ldots, m$ каждая точка p_i попадает в стек S ровно по одному разу. Как и при анализе процедуры MULTIPOP в разделе 17.1, нетрудно заметить, что каждой операции PUSH соответствует не более одной операции POP. По крайней мере три точки (p_0, p_1 и p_m) никогда не снимаются со стека, поэтому всего выполняется не более $m-2$ операций POP. В каждой итерации цикла **while** выполняется одна операция POP, следовательно, всего этих итераций не более $m-2$. Так как проверка в строке 7 занимает время $O(1)$, каждый вызов POP также требует $O(1)$ времени, и в силу неравенства $m \leqslant n-1$, полное время, которое требуется для выполнения цикла **while**, равно $O(n)$. Таким образом, время выполнения процедуры GRAHAM_SCAN составляет $O(n \lg n)$.

Обход по Джарвису

При построении выпуклой оболочки множества точек Q путем ***обхода по Джарвису*** (Jarvis's march) используется метод, известный как ***упаковка пакета*** (package wrapping) или ***упаковка подарка*** (gift wrapping). Время выполнения алгоритма равно $O(nh)$, где h — количество вершин $\mathrm{CH}(Q)$. В том случае, когда h равно $o(\lg n)$, обход по Джарвису работает быстрее, чем сканирование по Грэхему.

Интуитивно обход по Джарвису моделирует обертывание плотного куска бумаги вокруг множества Q. Начнем с того, что прикрепим конец бумаги к нижайшей точке множества, т.е. к той же точке p_0, с которой начинается сканирование по Грэхему. Эта точка является вершиной выпуклой оболочки. Натянем бумагу вправо,

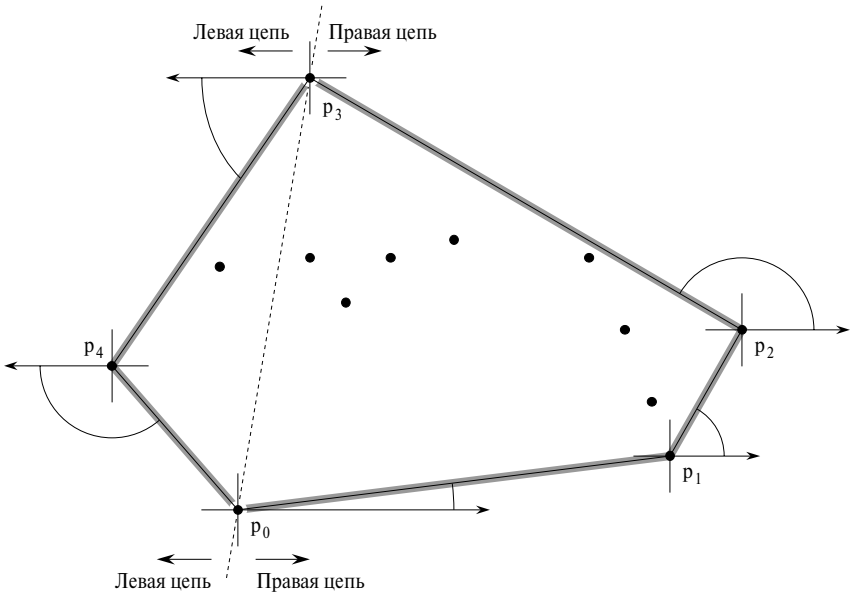


Рис. 33.9. Работа алгоритма обхода по Джарвису

чтобы она не провисала, после чего переместим ее вверх до тех пор, пока она не коснется какой-то точки. Эта точка также должна быть вершиной выпуклой оболочки. Сохраняя бумагу натянутой, продолжим наматывать ее на множество вершин, пока не вернемся к исходной точке p_0.

Если говорить более формально, то при обходе по Джарвису строится последовательность $H = \langle p_0, p_1, \ldots, p_{h-1} \rangle$ вершин $\mathrm{CH}(Q)$. Начнем с точки p_0. Как видно из рис. 33.9, следующая вершина выпуклой оболочки p_1 имеет наименьший полярный угол относительно точки p_0. (При наличии совпадений выбирается точка, самая удаленная от точки p_0.) Аналогично, точка p_2 имеет наименьший полярный угол относительно точки p_1 и т.д. По достижении самой высокой вершины, скажем, p_k (совпадения разрешаются путем выбора самой удаленной из таких вершин), оказывается построенной (рис. 33.9) ***правая цепь*** (right chain) оболочки $\mathrm{CH}(Q)$. Чтобы сконструировать ***левую цепь*** (left chain), начнем с точки p_k и выберем в качестве p_{k+1} точку с наименьшим полярным углом относительно точки p_k, но *относительно отрицательного направления оси x*. Продолжаем выполнять эту процедуру, отсчитывая полярный угол от отрицательного направления оси x, пока не вернемся к исходной точке p_0.

Обход по Джарвису можно было бы реализовать путем единообразного обхода вокруг выпуклой оболочки, т.е. не прибегая к отдельному построению правой и левой цепей. В такой реализации обычно отслеживается угол последней стороны выпуклой оболочки и накладывается требование, чтобы последовательность

углов, образованных сторонами оболочки с положительным направлением оси x строго возрастала (в интервале от 0 до 2π радиан). Преимущество конструирования отдельных цепей заключается в том, что исключается необходимость явно вычислять углы; для сравнения углов достаточно методов, описанных в разделе 33.1.

При надлежащей реализации время выполнения обхода по Джарвису равно $O(nh)$. Для каждой из h вершин оболочки $\mathrm{CH}(Q)$ находится вершина с минимальным полярным углом. Каждое сравнение полярных углов длится в течение времени $O(1)$, если используются методы, описанные в разделе 33.1. Как показано в разделе 9.1, если каждая операция сравнения выполняется в течение времени $O(1)$, то выбор минимального из n значений занимает время $O(n)$. Таким образом, обход по Джарвису длится в течение времени $O(nh)$.

Упражнения

33.3-1. Докажите, что в процедуре GRAHAM_SCAN точки p_1 и p_m должны быть вершинами $\mathrm{CH}(Q)$.

33.3-2. Рассмотрим модель вычислений, которая поддерживает сложение, сравнение и умножение и в которой существует нижняя граница времени сортировки n чисел, равная $\Omega(n \lg n)$. Докажите, что в такой модели $\Omega(n \lg n)$ — нижняя граница времени вычисления вершин выпуклой оболочки множества n точек в порядке их обхода.

33.3-3. Докажите, что в заданном множестве точек Q две самые удаленные друг от друга точки должны быть вершинами $\mathrm{CH}(Q)$.

33.3-4. Для заданного многоугольника P и точки q на его границе ***тенью*** (shadow) точки q называется множество точек r, для которых отрезок $\overline{qr}$ полностью находится на границе или во внутренней области многоугольника P. Многоугольник P называется ***звездообразным*** (star-shaped), если в его внутренней области существует точка p, которая находится в тени любой точки, лежащей на границе этого многоугольника. Множество всех таких точек называется ***ядром*** (kernel) многоугольника P. На рис. 33.10 приведены примеры двух многоугольников. В части *а* показан звездообразный многоугольник. Отрезок, соединяющий точку p с любой из принадлежащих границе многоугольника точек q пересекает эту границу только в точке q. В части *б* рисунка приведен многоугольник, не являющийся звездообразным. Выделенная серым цветом левая область является тенью точки q, а выделенная правая область — тенью точки q'. Поскольку эти области не перекрываются, ядро является пустым. Пусть звездообразный n-угольник P задан своими вершинами, перечисленными в порядке обхода против часовой стрелки. Покажите, как построить $\mathrm{CH}(P)$ за время $O(n)$.

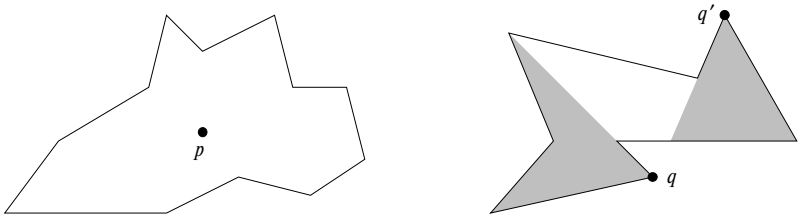


Рис. 33.10. Определение звездообразного многоугольника, о котором идет речь в упражнении 33.3-4

33.3-5. В ***оперативной задаче о выпуклой оболочке*** (on-line convex-hull problem) параметры каждой из n точек из заданного множества Q становятся известны по одной точке за раз. После получения сведений об очередной точке строится выпуклая оболочка тех точек, о которых стало известно к настоящему времени. Очевидно, можно было бы применять сканирование по Грэхему к каждой из точек, затратив при этом полное время, равное $O\left(n^2 \lg n\right)$. Покажите, как решить оперативную задачу о выпуклой оболочке за время $O\left(n^2\right)$.

⋆ 33.3-6. Покажите, как реализовать инкрементный метод построения выпуклой оболочки таким образом, чтобы время его работы было равно $O\left(n \lg n\right)$.

33.4 Поиск пары ближайших точек

Теперь рассмотрим задачу о поиске в множестве Q, состоящем из $n \geqslant 2$ точек, пары ближайших друг к другу точек. Термин “ближайший” относится к обычному евклидову расстоянию: расстояние между точками $p_1 = (x_1, y_1)$ и $p_2 = (x_2, y_2)$ равно $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Две точки в множестве Q могут совпадать; в этом случае расстояние между ними равно нулю. Эта задача находит применение, например, в системах управления транспортом. В системе управления воздушным или морским транспортом может понадобиться узнать, какие из двух транспортных средств находятся друг к другу ближе всего, чтобы предотвратить возможное столкновение между ними.

В алгоритме, который работает “в лоб”, просто перебираются все $\binom{n}{2} = \Theta\left(n^2\right)$ пар точек. В данном разделе описывается решение этой задачи с помощью алгоритма разбиения. Время решения этим методом определяется знакомым рекуррентным соотношением $T\left(n\right) = 2T\left(n/2\right) + O\left(n\right)$. Таким образом, время работы этого алгоритма равно $O\left(n \lg n\right)$.

Алгоритм декомпозиции

В каждом рекурсивном вызове описываемого алгоритма в качестве входных данных используется подмножество $P \subseteq Q$ и массивы X и Y, в каждом из которых содержатся все точки входного подмножества P. Точки в массиве X отсортированы в порядке монотонного возрастания их координаты x. Аналогично, массив Y отсортирован в порядке монотонного возрастания координаты y. Заметим, что достигнуть границы $O(n \lg n)$ не удастся, если сортировка будет производиться в каждом рекурсивном вызове; если бы мы поступали таким образом, то время работы такого метода определялось бы рекуррентным соотношением $T(n) = 2T(n/2) + O(n \lg n)$, решение которого равно $T(n) = O\left(n \lg^2 n\right)$. (Это можно показать с помощью версии основного метода, описанной в упражнении 4.4-2.) Немного позже станет понятно, как с помощью "предварительной сортировки" поддерживать отсортированность, не прибегая к процедуре сортировке в каждом рекурсивном вызове.

В рекурсивном вызове со входными данными P, X и Y сначала проверяется, выполняется ли условие $|P| \leqslant 3$. Если оно справедливо, то в вызове просто применяется упомянутый выше метод решения "в лоб": сравниваются между собой все $\binom{|P|}{2}$ пар точек, и возвращается пара точек, расположенных друг к другу ближе других. Если $|P| > 3$, то производится рекурсивный вызов в соответствии с описанной ниже парадигмой "разделяй и властвуй".

Разделение. Находится вертикальная прямая l, которая делит множество точек P на два множества P_L и P_R, таких что $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$, все точки множества P_L находятся слева от прямой l или на этой прямой, а все точки множества P_R находятся справа от прямой l или на этой прямой. Массив X разбивается на массивы X_L и X_R, содержащие точки множеств P_L и P_R соответственно, отсортированные в порядке монотонного возрастания координаты x. Аналогично, массив Y разбивается на массивы Y_L и Y_R, содержащие соответственно точки множеств P_L и P_R, отсортированные в порядке монотонного возрастания координаты y.

Покорение. После разбиения множества P на подмножества P_L и P_R производится два рекурсивных вызова: один для поиска пары ближайших точек в множестве P_L, а другой для поиска пары ближайших точек в множестве P_R. В качестве входных данных в первом вызове выступает подмножество P_L и массивы X_L и Y_L; во втором вызове на вход подается множество P_R, а также массивы X_R и Y_R. Обозначим расстояния между ближайшими точками в множествах P_L и P_R через δ_L и δ_R соответственно; введем также обозначение $\delta = \min(\delta_L, \delta_R)$.

Комбинирование. Ближайшая пара либо находится друг от друга на расстоянии δ, найденном в одном из рекуррентных вызовов, либо образована точками, одна из которых принадлежит множеству P_L, а вторая — множеству

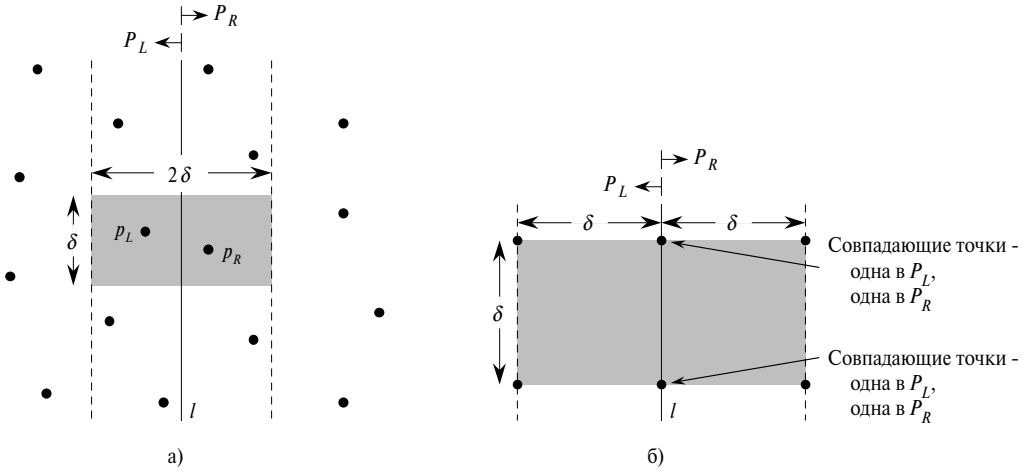


Рис. 33.11. Основные концепции, которые используются при доказательстве достаточности проверки 7 точек массива Y' при поиске пары ближайших точек

P_R. В алгоритме определяется, существует ли пара таких точек, расстояние между которыми меньше δ. Заметим, что если существует такая "пограничная" пара точек, расстояние между которыми меньше δ, то обе они не могут находиться дальше от прямой l, чем на расстоянии δ. Таким образом, как видно из рис. 33.11*a*, обе эти точки должны лежать в пределах вертикальной полосы шириной 2δ, в центре которой находится прямая l. Для поиска такой пары (если она существует) в алгоритме производятся описанные ниже действия.

1. Создается массив Y' путем удаления из массива Y всех точек, не попадающих в полосу шириной 2δ. Как и в массиве Y, в массиве Y' точки отсортированы в порядке возрастания координаты y.
2. Для каждой точки p из массива Y' алгоритм пытается найти в этом же массиве точки, которые находятся в δ-окрестности от точки p. Как мы вскоре увидим, достаточно рассмотреть лишь 7 точек, расположенных в массиве Y' после точки p. В алгоритме вычисляется расстояние от точки p до каждой из этих семи точек, и отслеживается, какое из расстояний $\delta\prime$ между всеми парами точек в массиве Y' является наименьшим.
3. Если выполняется неравенство $\delta' < \delta$, то в вертикальной полосе действительно содержится пара точек, которые находятся друг от друга ближе, чем те, что были найдены в результате рекурсивных вызовов. В этом случае возвращается эта пара точек и соответствующее ей расстояние $\delta\prime$. В противном случае возвращается пара ближайших точек

и расстояние между ними δ, найденное в результате рекурсивных вызовов.

В приведенном выше описании опущены некоторые детали реализации, необходимые для сокращения времени работы до величины $O(n \lg n)$. После доказательства корректности алгоритма будет показано, как реализовать этот алгоритм так, чтобы достичь желаемой границы для времени работы.

Корректность

Корректность этого алгоритма, предназначенного для поиска пары ближайших точек, очевидна, за исключением двух аспектов. Во-первых, мы уверены, что в результате рекурсивного спуска до случая, когда $|P| \leqslant 3$, никогда не придется решать вспомогательную задачу, в которой задана только одна точка. Второй аспект заключается в том, что понадобится проверить всего 7 точек, расположенных в массиве Y' ниже каждой точки p; далее будет приведено доказательство этого свойства.

Предположим, что на некотором уровне рекурсии пару ближайших точек образуют точки $p_L \in P_L$ и $p_R \in P_R$. Таким образом, расстояние δ' между этими точками строго меньше δ. Точка p_L должна находиться слева от прямой l на расстоянии, не превышающем величину δ, или на этой прямой. Аналогично, точка p_R находится справа от прямой l на расстоянии, не превышающем величину δ, или на этой прямой. Кроме того, расстояние по вертикали между точками p_L и p_R не превышает δ. Таким образом, как видно из рис. 33.11*а*, точки p_L и p_R находятся внутри прямоугольника $\delta \times 2\delta$, через центр которого проходит прямая l. (В этот прямоугольник могут также попасть другие точки.)

Теперь покажем, что прямоугольник размерами $\delta \times 2\delta$ может содержать не более восьми точек. Рассмотрим квадрат $\delta \times \delta$, который составляет левую половину этого прямоугольника. Поскольку расстояние между любой парой точек из множества P_L не меньше δ, в этом квадрате может находиться не более четырех точек (рис. 33.11*б*). Аналогично, в квадрате размерами $\delta \times \delta$, составляющем правую половину означенного прямоугольника, тоже может быть не более четырех точек. Таким образом, в прямоугольнике $\delta \times 2\delta$ содержится не более восьми точек. (Заметим, что поскольку точки на прямой l могут входить либо в множество P_L, либо в множество P_R, на этой прямой может лежать до четырех точек. Этот предел достигается, если имеется две пары совпадающих точек, причем в каждой паре одна из точек принадлежит множеству P_L, а другая — множеству P_R. Кроме того, одна из этих пар находится на пересечении прямой l и верхней стороны прямоугольника, а вторая — на пересечении прямой l и нижней стороны прямоугольника.)

Поскольку в прямоугольнике указанных выше размеров может содержаться не более восьми точек множества P, очевидно, что достаточно проверить 7 точек,

расположенных в массиве Y' после каждой точки. Продолжая придерживаться предположения, что пара ближайших друг к другу точек состоит из точек p_L и p_R, без потери общности предположим, что в массиве Y' точка p_L находится перед точкой p_R. В этом случае точка p_R является одной из семи точек, следующих после точки p_L, даже если точка p_L встречается в массиве Y' так рано, насколько это возможно, а точка p_R — настолько поздно, насколько это возможно. Таким образом, корректность алгоритма, предназначенного для поиска пары ближайших точек, доказана.

Реализация и время работы алгоритма

Как уже упоминалось, наша цель заключается в том, чтобы показать, что время работы представленного алгоритма описывается рекуррентным соотношением $T(n) = 2T(n/2) + O(n)$, где $T(n)$ — время работы алгоритма для n точек. Основная сложность — это добиться, чтобы массивы X_L, X_R, Y_L и Y_R, которые передаются в рекурсивных вызовах, были отсортированы по соответствующей координате и чтобы массив Y' был отсортирован по координате y. (Заметим, что если массив X, который передается в рекурсивном вызове, уже отсортирован, то разбиение множества P на подмножества P_L и P_R легко выполнить в течение времени, линейного по количеству элементов.)

Главное заключается в том, что в каждом вызове следует сформировать отсортированное подмножество отсортированного массива. Например, пусть в каком-то отдельно взятом рекурсивном вызове задано подмножество P и массив Y, отсортированный по координате y. В процессе разбиения множества P на подмножества P_L и P_R нужно образовать массивы Y_L и Y_R, которые должны быть отсортированы по координате y. Более того, эти массивы необходимо сформировать в течение линейного времени. Этот метод можно рассматривать как антипод процедуры MERGE, описанной в разделе 2.3.1: отсортированный массив разбивается на два отсортированных массива. Эта идея реализована в приведенном ниже псевдокоде.

```
1  length[Y_L] ← length[Y_R] ← 0
2  for i ← 1 to length[Y]
3       do if Y[i] ∈ P_L
4            then length[Y_L] ← length[Y_L] + 1
5                 Y_L[length[Y_L]] ← Y[i]
6            else length[Y_R] ← length[Y_R] + 1
7                 Y_R[length[Y_R]] ← Y[i]
```

Точки массива Y просто обрабатываются в порядке их расположения в этом массиве. Если точка $Y[i]$ принадлежит множеству P_L, она добавляется в конец массива

Y_L; в противном случае она добавляется в конец массива Y_R. С помощью аналогичного псевдокода можно сформировать массивы X_L, X_R и Y'.

Единственный оставшийся вопрос — как сделать так, чтобы точки были отсортированы с самого начала. Для этого следует провести их ***предварительную сортировку*** (presorting); т.е. это делается один раз *перед* первым рекурсивным вызовом. Отсортированные массивы передаются в первом рекурсивном вызове, а затем они будут дробиться в рекурсивных вызовах до тех пор, пока это будет необходимо. Предварительная сортировка добавит ко времени работы алгоритма величину $O(n \lg n)$, но позволит выполнять каждый шаг рекурсии в течение линейного времени. Таким образом, если обозначить через $T(n)$ время обработки каждого шага рекурсии, а через $T'(n)$ — общее время работы всего алгоритма, то получим равенство $T'(n) = T(n) + O(n \lg n)$ и соотношение

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{если } n > 3, \\ O(1) & \text{если } n \leqslant 3. \end{cases}$$

Таким образом, $T(n) = O(n \lg n)$ и $T'(n) = O(n \lg n)$.

Упражнения

33.4-1. Профессор предложил схему, позволяющую ограничиться в алгоритме поиска пары ближайших точек пятью точками, которые находятся в массиве Y' после каждой из точек. Его идея заключается в том, чтобы точки, которые лежат на прямой l, всегда заносились в множество P_L. Тогда на этой прямой не может быть пар совпадающих точек, одна из которых принадлежит множеству P_L, а другая — множеству P_R. Таким образом, прямоугольник размерами $\delta \times 2\delta$ может содержать не более шести точек. Какая ошибка содержится в предложенной профессором схеме?

33.4-2. Покажите, как без ухудшения асимптотического времени работы алгоритма убедиться в том, что передаваемое в самом первом рекурсивном вызове множество не содержит совпадающих точек. Докажите, что в этом случае достаточно проверять по 5 точек, расположенных после каждой точки в массиве Y'.

33.4-3. Расстояние между двумя точками можно определить и не в евклидовом смысле, а по-другому. Так, ***L_m-расстояние*** (L_m-distance) между точками p_1 и p_2 на плоскости задается выражением $(|x_1 - x_2|^m + |y_1 - y_2|^m)^{1/m}$. Поэтому евклидово расстояние — это L_2-расстояние. Модифицируйте алгоритм поиска пары ближайших точек для L_1-расстояния, известного как ***манхэттенское расстояние*** (Manhattan distance).

33.4-4. Для точек p_1 и p_2, заданных на плоскости, L_∞-расстоянием между ними называется величина $\max(|x_1 - x_2|, |y_1 - y_2|)$. Модифицируйте алгоритм поиска пары ближайших точек для L_∞-расстояния.

33.4-5. Предложите, как изменить алгоритм поиска пары ближайших точек, чтобы избежать в нем предварительной сортировки массива Y, но чтобы время его работы по-прежнему осталось равным $O(n \lg n)$. (*Указание*: объедините отсортированные массивы Y_L и Y_R в отсортированный массив Y.)

Задачи

33-1. **Выпуклые слои**

Для заданного на плоскости множества точек Q ***выпуклые слои*** (convex layers) определяются следующим образом. Первый выпуклый слой множества Q состоит из вершин $\mathrm{CH}(Q)$. Уберем эти точки и снова найдем выпуклую оболочку — ее вершины образуют второй выпуклый слой. Отбросим их и возьмем выпуклую оболочку остатка — ее вершины образуют третий выпуклый слой и т.д. Строго выпуклые слои определяются индуктивно следующим образом. Первый выпуклый слой множества Q состоит из вершин $\mathrm{CH}(Q)$. Определим при $i > 1$ множество Q_i, состоящее из точек множества Q, из которого удалили все точки выпуклых слоев $1, 2, \ldots, i-1$. Тогда i-м выпуклым слоем множества Q является $\mathrm{CH}(Q_i)$, если $Q_i \neq \emptyset$; в противном случае он считается неопределенным.

а) Разработайте алгоритм, который находил бы выпуклые слои множества из n точек в течение времени $O(n^2)$.

б) Докажите, что для построения выпуклых слоев в множестве из n точек по любой вычислительной модели, в которой сортировка n действительных чисел занимает время $\Omega(n \lg n)$, требуется время $\Omega(n \lg n)$.

33-2. **Максимальные слои**

Пусть Q — множество n точек на плоскости. Говорят, что точка (x, y) ***доминирует*** (dominates) над точкой (x', y'), если выполняются неравенства $x \geqslant x'$ и $y \geqslant y'$. Точка множества Q, над которой не доминирует никакая другая точка этого множества, называется ***максимальной*** (maximal). Заметим, что множество Q может содержать большое количество максимальных точек, которые можно объединить в ***максимальные слои*** (maximal layers) следующим образом. В первый максимальный слой L_1 образует подмножество максимальных точек множества Q. При $i > 1$

i-м максимальным слоем L_i является подмножество максимальных точек множества $Q - \bigcup_{j=1}^{i-1} L_j$.

Предположим, что множество Q содержит k непустых максимальных слоев, и пусть y_i — координата y самой левой точки в слое L_i ($i = 1, 2, \ldots, k$). Мы предполагаем, что координаты x или y никаких двух точек множества Q не совпадают.

а) Покажите, что $y_1 > y_2 > \cdots > y_k$.

Рассмотрим точку (x, y), которая находится левее всех точек множества Q и координата y которой отличается от координаты y любой другой точки этого множества. Введем обозначение $Q' = Q \cup \{(x, y)\}$.

б) Пусть j — минимальный индекс, такой что $y_j < y$; если такого индекса нет ($y < y_k$), то $j = k + 1$. Покажите, что максимальные слои множества Q' обладают следующими свойствами.

- Если $j \leqslant k$, то максимальные слои множества Q' совпадают с максимальными слоями множества Q за исключением слоя L_j, который в качестве новой крайней левой точки включает точку (x, y).
- Если $j = k + 1$, то первые k максимальных слоя множества Q' совпадают с максимальными слоями множества Q, но, кроме того, множество Q' имеет непустой $(k + 1)$-й максимальный слой $L_{k+1} = \{(x, y)\}$.

в) Разработайте алгоритм, позволяющий в течение времени $O(n \lg n)$ построить максимальные слои множества Q, состоящего из n точек. (*Указание*: проведите вертикальную выметающую прямую справа налево.)

г) Возникнут ли какие-либо сложности, если разрешить входным точкам иметь одинаковые координаты x или y? Предложите способ решения таких проблем.

33-3. **Охотники за привидениями и привидения**

Группа, состоящая из n охотников за привидениями, борется с n привидениями. Каждый охотник вооружен протонной установкой, уничтожающей привидение протонным пучком. Поток протонов распространяется по прямой и прекращает свой путь, когда попадает в привидение. Охотники придерживаются такой стратегии. Каждый из них выбирает среди привидений свою цель, в результате чего образуется n пар "охотник–привидение". После этого каждый охотник выпускает пучок протонов по своей жертве. Известно, что пересечение пучков *очень* опасно, поэтому охотники должны выбирать привидения так, чтобы избежать пересечений.

Предположим, что положение каждого охотника и каждого привидения задано фиксированной точкой на плоскости и что никакие три точки не коллинеарны.

а) Докажите, что всегда существует прямая, проходящая через одного охотника и одно привидение, для которой количество охотников, попавших в одну из полуплоскостей относительно этой прямой, равно количеству привидений в этой же полуплоскости. Опишите, как найти такую прямую за время $O(n \lg n)$.

б) Разработайте алгоритм, позволяющий в течение времени $O\left(n^2 \lg n\right)$ разбить охотников и привидения на пары таким образом, чтобы не было пересечения пучков.

33-4. **Сбор палочек**

Профессор занимается исследованием множества из n палочек, сложенных в кучу в некоторой конфигурации. Положение каждой палочки задается ее конечными точками, а каждая конечная точка представляет собой упорядоченную тройку координат (x, y, z). Вертикальных палочек нет. Профессор хочет собрать по одной все палочки, соблюдая условие, согласно которому он может снимать палочку только тогда, когда на ней не лежат никакие другие палочки.

а) Разработайте процедуру, которая бы для двух заданных палочек a и b определяла, находится ли палочка a над палочкой b, под ней или палочка a не связана ни одним из этих соотношений с палочкой b.

б) Разработайте эффективный алгоритм, позволяющий определить, можно ли собрать все палочки, и в случае положительного ответа предлагающий допустимую последовательность, в которой следует собирать палочки.

33-5. **Разреженно-оболочечные распределения**

Рассмотрим задачу, в которой нужно построить выпуклую оболочку множества заданных на плоскости точек, нанесенных в соответствии с каким-то известным случайным распределением. Для некоторых распределений математическое ожидание размера (количества точек) выпуклой оболочки такого множества, состоящего из n точек, равно $O\left(n^{1-\varepsilon}\right)$, где $\varepsilon > 0$ — некоторая положительная константа. Назовем такое распределение ***разреженно-оболочечным*** (sparse-hulled). К разреженно-оболочечным распределениям относятся следующие.

- Точки равномерно распределены в круге единичного радиуса. Математическое ожидание размера выпуклой оболочки равно $\Theta\left(n^{1/3}\right)$.

- Точки равномерно распределены внутри выпуклого k-угольника, где k — произвольная константа. Математическое ожидание размера выпуклой оболочки равно $\Theta(\lg n)$.
- Точки наносятся в соответствии с двумерным нормальным распределением. Математическое ожидание размера выпуклой оболочки равно $\Theta\left(\sqrt{\lg n}\right)$.

а) Даны два выпуклых многоугольника с n_1 и n_2 вершинами соответственно. Покажите, как построить выпуклую оболочку всех $n_1 + n_2$ точек в течение времени $O(n_1 + n_2)$. (Многоугольники могут пересекаться.)

б) Покажите, что можно разработать алгоритм, который отыскивает выпуклую оболочку n независимых точек, подчиненных некоторому разреженно-оболочечному распределению, за время $O(n)$. (*Указание*: рекурсивно постройте выпуклую оболочку для двух половин множества, а затем объедините полученные результаты.)

Заключительные замечания

В этой главе мы лишь вкратце обсудили тему алгоритмов и методов вычислительной геометрии. Среди серьезных изданий по вычислительной геометрии можно выделить книги Препараты (Preparata) и Шамоса (Shamos) [247], а также Эдельсбруннера (Edelsbrunner) [83] и О'Рурка (O'Rourke) [235].

Несмотря на то, что геометрия изучается с античных времен, развитие алгоритмов для геометрических задач — относительно новое направление. Препарата и Шамос отмечают, что самое раннее упоминание о сложности задачи было сделано Лемоном (E. Lemoine) в 1902 году. Изучая евклидовы построения, в которых используется только циркуль и линейка, все действия он свел к набору пяти примитивов: размещение ножки циркуля в заданной точке, размещение ножки циркуля на заданной прямой, построение окружности, совмещение края линейки с заданной точкой и наконец построение прямой. Лемона интересовало количество примитивов, необходимых для построения заданной конструкции; это число он назвал "простотой" данной конструкции.

Описанный в разделе 33.2 алгоритм, в котором определяется, пересекаются ли какие-либо отрезки, предложен Шамосом и Гоем (Hoey) [275].

Изначальная версия сканирования по Грэхему была представлена Грэхемом (Graham) [130]. Алгоритм оборачивания предложен Джарвисом (Jarvis) [165]. Яо (Yao) [318] с помощью модели дерева решений доказал, что нижняя граница времени работы алгоритма, предназначенного для построения выпуклой оболочки, равна $\Omega(n \lg n)$. С учетом количества вершин h выпуклой оболочки

в асимптотическом пределе оптимальным является алгоритм отсечения и поиска, разработанный Киркпатриком (Kirkpatrick) и Зайделем (Seidel) [180], время работы которого равно $O(n \lg h)$.

Алгоритм разбиения со временем работы $O(n \lg n)$, предназначенный для поиска пары ближайших точек, предложен Препаратой и опубликован в книге Препараты и Шамоса [247]. Препарата и Шамос также показали, что в модели дерева решений этот алгоритм асимптотически оптимален.

ГЛАВА 34

NP-полнота

Почти все изученные нами до сих пор алгоритмы имеют ***полиномиальное время работы*** (polynomial-time algorithms): для входных данных размера n их время работы в наихудшем случае равно $O\left(n^k\right)$, где k — некоторая константа. Возникает естественный вопрос: *все* ли задачи можно решить в течение полиномиального времени? Ответ отрицательный. В качестве примера можно привести знаменитую "задачу останова", предложенную Тьюрингом (Turing). Эту задачу невозможно решить ни на одном компьютере, каким бы количеством времени мы не располагали. Существуют также задачи, которые можно решить, но не удается сделать это за время $O\left(n^k\right)$, где k — некоторая константа. Вообще говоря, о задачах, разрешимых с помощью алгоритмов с полиномиальным временем работы, возникает представление как о легко разрешимых или простых, а о задачах, время работы которых превосходит полиномиальное, — как о трудно разрешимых или сложных.

Однако тема этой главы — интересный класс задач, которые называются "NP-полными". Их статус пока что неизвестен. Для решения NP-полных задач до настоящего времени не разработано алгоритмов с полиномиальным временем работы, но и не доказано, что для какой-то из них таких алгоритмов не существует. Этот так называемый вопрос $P \neq NP$ с момента своей постановки в 1971 году стал одним из самых трудных в теории вычислительных систем.

Особо интригующим аспектом NP-полных задач является то, что некоторые из них на первый взгляд аналогичны задачам, для решения которых существуют алгоритмы с полиномиальным временем работы. В каждой из описанных ниже пар задач одна из них разрешима в течение полиномиального времени, а другая

является NP-полной. При этом различие между задачами кажется совершенно незначительным.

Поиск самых коротких и самых длинных простых путей. В главе 24 мы убедились, что даже при отрицательных весах ребер *кратчайшие* пути от отдельно взятого источника в ориентированном графе $G = (V, E)$ можно найти в течение времени $O(VE)$. Однако поиск *самого длинного* пути между двумя вершинами оказывается сложным. Задача по определению того, содержит ли граф простой путь, количество ребер в котором не меньше заданного числа, является NP-полной.

Эйлеров и Гамильтонов циклы. ***Эйлеров цикл*** (Euler tour) связанного ориентированного графа $G = (V, E)$ — это цикл, в котором переход по каждому *ребру* G осуществляется ровно один раз, хотя допускается неоднократное посещение некоторых вершин. В соответствии с результатами задачи 22-3, определить наличие Эйлерова цикла (а также найти составляющие его ребра) можно в течение времени $O(E)$. ***Гамильтонов цикл*** (hamiltonian cycle) ориентированного графа $G = (V, E)$ — это простой цикл, содержащий все *вершины* из множества V. Задача по определению того, содержится ли в ориентированном графе Гамильтонов цикл, является NP-полной. (Далее в этой главе будет доказано, что задача по определению того, содержится ли в *неориентированном* графе Гамильтонов цикл, также является NP-полной.)

2-CNF- и 3-CNF-выполнимость. Булева формула содержит переменные, принимающие значения 0 и 1, булевы операторы, такие как $\wedge$ (И), $\vee$ (ИЛИ) и $\neg$ (НЕ), а также скобки. Булева формула называется ***выполнимой*** (satisfiable), если входящим в ее состав переменным можно присвоить такие значения 0 и 1, чтобы в результате вычисления формулы получилось значение 1. Далее в этой главе даются более формализованные определения всех терминов, а пока, говоря неформально, булева формула представлена в ***k-конъюнктивной нормальной форме*** (k-conjunctive normal form), или k-CNF, если она имеет вид конъюнкции (И) взятых в скобки выражений, являющихся дизъюнкцией (ИЛИ) ровно k переменных или их отрицаний (НЕ). Например, формула $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ представлена в 2-CNF. (Для нее существует выполнимый набор $x_1 = 1$, $x_2 = 0$, $x_3 = 1$.) Можно сформулировать алгоритм с полиномиальным временем работы, позволяющий определить, является ли 2-CNF формула выполнимой. Однако, как будет показано далее в этой главе, определение того, является ли 3-CNF формула выполнимой — это NP-полная задача.

NP-полнота и классы P и NP

В этой главе мы будем ссылаться на три класса задач: P, NP и NPC (класс NP-полных задач). В этом разделе они описываются неформально, а их формальное определение будет дано позже.

Класс P состоит из задач, разрешимых в течение полиномиального времени работы. Точнее говоря — это задачи, которые можно решить за время $O\left(n^k\right)$, где k — некоторая константа, а n — размер входных данных задачи. Большинство задач, рассмотренных в предыдущих главах, принадлежат классу P.

Класс NP состоит из задач, которые поддаются *проверке* в течение полиномиального времени. Имеется в виду, что если мы каким-то образом получаем "сертификат" решения, то в течение времени, полиномиальным образом зависящего от размера входных данных задачи, можно проверить корректность такого решения. Например, в задаче о гамильтоновом цикле с заданным ориентированным графом $G = (V, E)$ сертификат имел бы вид последовательности $\left\langle v_1, v_2, \ldots, v_{|V|}\right\rangle$ из $|V|$ вершин. В течение полиномиального времени легко проверить, что $(v_i, v_{i+1}) \in E$ при $i = 1, 2, \ldots, |V| - 1$ и что $\left(v_{|V|}, v_1\right) \in E$. Приведем другой пример: в задаче о 3-CNF-выполнимости в сертификате должно быть указано, какие значения следует присвоить переменным. В течение полиномиального времени легко проверить, удовлетворяет ли такое присваивание булевой формуле.

Любая задача класса P принадлежит классу NP, поскольку принадлежность задачи классу P означает, что ее решение можно получить в течение полиномиального времени, даже не располагая сертификатом. Это замечание будет формализовано ниже в данной главе, а пока что можно считать, что $\mathrm{P} \subseteq \mathrm{NP}$. Остается открытым вопрос, является ли P строгим подмножеством NP.

Неформально задача принадлежит классу NPC (такие задачи называются ***NP-полными*** (NP-complete)), если она принадлежит классу NP и является такой же "сложной", как и любая задача из класса NP. Ниже в этой главе будет дано формальное определение того, что означает выражение "задача такая же по сложности, как любая задача из класса NP". А пока что мы примем без доказательства положение, что если *любую* NP-полную задачу можно решить в течение полиномиального времени, то для *каждой* задачи из класса NP существует алгоритм с полиномиальным временем работы. Большинство ученых, занимающихся теорией вычислительных систем, считают NP-полные задачи очень трудноразрешимыми, потому что при огромном разнообразии изучавшихся до настоящего времени NP-полных задач ни для одной из них пока так и не найдено решение в виде алгоритма с полиномиальным временем работы. Таким образом, было бы крайне удивительно, если бы все они оказались разрешимыми в течение полиномиального времени.

Чтобы стать квалифицированным разработчиком алгоритмов, необходимо понимать основы теории NP-полноты. Если установлено, что задача NP-полная, это

служит достаточно надежным указанием на то, что она трудноразрешимая. Как инженер, вы эффективнее потратите время, если займетесь разработкой приближенного алгоритма (см. главу 35) или решением легкого особого случая, вместо того, чтобы искать быстрый алгоритм, выдающий точное решение задачи. Более того, многие естественно возникающие интересные задачи, которые на первый взгляд не сложнее задачи сортировки, поиска в графе или определения потока в сети, фактически являются NP-полными. Таким образом, важно ознакомиться с этим замечательным классом задач.

Как показать, что задача является NP-полной

Методы, позволяющие доказать, что та или иная задача является NP-полной, отличаются от методов, которые использовались в большей части этой книги для разработки и анализа алгоритмов. Имеется фундаментальная причина такого различия: чтобы показать, что задача является NP-полной, делается утверждение о том, насколько она сложна (или, по крайней мере, насколько она сложна в нашем представлении), а не о том, насколько она проста. Не предпринимается попыток доказать, что существуют эффективные алгоритмы. Скорее, мы пытаемся доказать, что эффективных алгоритмов, вероятнее всего, не существует. В этом смысле доказательство NP-полноты несколько напоминает представленное в разделе 8.1 доказательство того, что нижняя граница любого алгоритма, работающего по методу сравнений, равна $\Omega(n \lg n)$. Однако методы, использующиеся для доказательства NP-полноты, отличаются от методов с применением дерева решений, описанных в разделе 8.1.

При доказательстве NP-полноты задачи используются три основные концепции, описанные ниже.

Задачи принятия решения и задачи оптимизации

Многие представляющие интерес задачи являются ***задачами оптимизации*** (optimization problems), каждому допустимому ("законному") решению которых можно сопоставить некоторое значение и для которых нужно найти допустимое решение с наилучшим значением. Например, в задаче, получившей название SHORTEST_PATH, задается неориентированный граф G, а также вершины u и v, и нужно найти путь из вершины u к вершине v, в котором содержится наименьшее количество ребер. (Другими словами, SHORTEST_PATH — это задача поиска кратчайшего пути между парой вершин невзвешенного неориентированного графа.) Однако NP-полнота непосредственно применима не к задачам оптимизации, а к ***задачам принятия решения*** (decision problems), в которых ответ может быть положительным или отрицательным (говоря более формально, принимать значения "1" или "0").

Хотя при доказательстве NP-полноты задачи приходится ограничиваться задачами принятия решения, между ними и задачами оптимизации существует удобная взаимосвязь. Наложив ограничение на оптимизируемое значение, поставленную задачу оптимизации можно свести к соответствующей задаче принятия решения. Например, задаче SHORTEST_PATH соответствует задача принятия решения под названием PATH: существует ли для заданных исходных данных, в число которых входит направленный граф G, вершины u и v, и целое число k, путь из вершины u к вершине v, состоящий не более чем из k ребер.

Взаимосвязь между задачей оптимизации и соответствующей ей задачей принятия решения полезна для нас, если мы пытаемся показать, что задача оптимизации является "сложной". Причина этого заключается в том, что задача принятия решения в некотором смысле "проще" или, по крайней мере, "не сложнее". Например, задачу PATH можно решить, решив задачу SHORTEST_PATH, а затем сравнивая количество ребер в найденном кратчайшем пути со значением параметра k в задаче принятия решения. Другими словами, если задача оптимизации простая, соответствующая ей задача принятия решения тоже простая. Формулируя это утверждение так, чтобы оно имело большее отношение к NP-полноте, можно сказать, что если удается засвидетельствовать сложность задачи принятия решения, это означает, что соответствующая задача оптимизации тоже сложная. Таким образом, хотя и приходится ограничиваться рассмотрением задач принятия решения, теория NP-полноты зачастую имеет следствия и для задач оптимизации.

Приведение

Сделанное выше замечание о том, что одна задача не сложнее или не легче другой, применимо, даже если обе задачи являются задачами принятия решения. Эта идея используется почти во всех доказательствах NP-полноты. Это делается следующим образом. Рассмотрим задачу принятия решения, скажем, задачу A, которую хотелось бы решить в течение полиномиального времени. Назовем входные данные отдельно взятой задачи ***экземпляром*** (instance) этой задачи. Например, экземпляром задачи PATH является некоторый граф G, некоторые его вершины u и v, а также некоторое целое число k. А теперь предположим, что существует другая задача принятия решения, скажем, B, для которой заранее известно, как решить ее в течение полиномиального времени. Наконец, предположим, что имеется процедура с приведенными ниже характеристиками, преобразующая любой экземпляр α задачи A в некоторый экземпляр β задачи B.

1. Это преобразование занимает полиномиальное время.
2. Ответы являются идентичными, т.е. в экземпляре α ответ "да" выдается тогда и только тогда, когда в экземпляре β тоже выдается ответ "да".

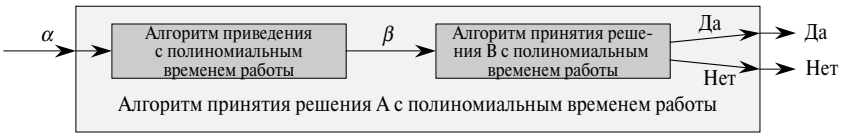


Рис. 34.1. Решение задачи принятия решения A в течение полиномиального времени с помощью алгоритма приведения с полиномиальным временем работы и известного алгоритма с полиномиальным временем работы, предназначенного для решения другой задачи B

Назовем такую процедуру ***алгоритмом приведения*** (reduction algorithm) с полиномиальным временем. Как видно из рис. 34.1, эта процедура предоставляет описанный ниже способ решения задачи A в течение полиномиального времени.

1. Заданный экземпляр α задачи A с помощью алгоритма приведения с полиномиальным временем преобразуется в экземпляр β задачи B.
2. Запускается алгоритм, решающий экземпляр β задачи принятия решения B в течение полиномиального времени.
3. Ответ для экземпляра β используется в качестве ответа для экземпляра α.

Поскольку для выполнения каждого из перечисленных выше этапов требуется полиномиальное время, это относится и ко всему процессу в целом, что дает способ решения экземпляра α задачи в течение полиномиального времени. Другими словами, путем сведения задачи A к задаче B “простота” задачи B используется для доказательства “простоты” задачи A.

Если вспомнить, что при доказательстве NP-полноты требуется показать, насколько сложной является задача, а не насколько она простая, доказательство NP-полноты с помощью приведения с полиномиальным временем работы выполняется обратно описанному выше способу. Продвинемся в разработке этой идеи еще на шаг и посмотрим, как с помощью приведения с полиномиальным временем показать, что для конкретной задачи B не существует алгоритмов с полиномиальным временем работы. Предположим, что имеется задача принятия решения A, относительно которой заранее известно, что для нее не существует алгоритма с полиномиальным временем работы. (Пока что мы не станем обременять себя размышлениями о том, как сформулировать такую задачу A.) Предположим также, что имеется преобразование, позволяющее в течение полиномиального времени преобразовать экземпляры задачи A в экземпляры задачи B. Теперь с помощью простого доказательства “от противного” можно показать, что для решения задачи B не существует алгоритмов с полиномиальным временем работы. Предположим обратное, т.е. что существует решение задачи B в виде алгоритма с полиномиальным временем работы. Тогда, воспользовавшись методом, проиллюстрированным на рис. 34.1, можно получить способ решения задачи A в течение полиномиаль-

ного времени, а это противоречит предположению о том, что таких алгоритмов для задачи A не существует.

Если речь идет о NP-полноте, то здесь нельзя предположить, что для задачи A вообще не существует алгоритмов с полиномиальным временем работы. Однако методология аналогична в том отношении, что доказательство NP-полноты задачи B основывается на предположении о NP-полноте задачи A.

Первая NP-полная задача

Поскольку метод приведения базируется на том, что для какой-то задачи заранее известна ее NP-полнота, то для доказательства NP-полноты различных задач нам понадобится "первая" NP-полная задача. В качестве таковой мы воспользуемся задачей, в которой задана булева комбинационная схема, состоящая из логических элементов И, ИЛИ и НЕ. В задаче спрашивается, существует ли для этой схемы такой набор входных булевых величин, для которого будет выдано значение 1. Доказательство NP-полноты этой первой задачи будет представлено в разделе 34.3.

Краткое содержание главы

В этой главе изучаются аспекты NP-полноты, которые непосредственно основываются на анализе алгоритмов. В разделе 34.1 формализуется понятие "задачи" и определяется класс сложности P как класс задач принятия решения, разрешимых в течение полиномиального времени. Также станет понятно, как эти понятия укладываются в рамки теории формальных языков. В разделе 34.2 определяется класс NP, к которому относятся задачи принятия решения, правильность решения которых можно проверить в течение полиномиального времени. Здесь также формально ставится вопрос $\mathrm{P} \neq \mathrm{NP}$.

В разделе 34.3 показано, как с помощью приведения с полиномиальным временем работы изучаются взаимоотношения между задачами. Здесь дано определение NP-полноты и представлен набросок доказательства того, что задача о выполнимости схемы является NP-полной. Имея одну NP-полную задачу, в разделе 34.4 мы увидим, как можно существенно проще доказать NP-полноту других задач с помощью приведения. Эта методология проиллюстрирована на примере двух задач на выполнимость формул, для которых доказывается NP-полнота. В разделе 34.5 NP-полнота демонстрируется для широкого круга других задач.

34.1 Полиномиальное время

Начнем исследование NP-полноты с формализации понятия задач, разрешимых в течение полиномиального времени. Эти задачи считаются легко разрешимыми,

но не из математических, а из философских соображений. В поддержку этого мнения можно привести три аргумента.

Во-первых, хотя и разумно считать трудноразрешимой задачу, для решения которой требуется время $\Theta\left(n^{100}\right)$, на практике крайне редко встречаются задачи, время решения которых выражается полиномом такой высокой степени. Для практических задач, которые решаются за полиномиальное время, показатель степени обычно намного меньше. Опыт показывает, что если для задачи становится известен алгоритм с полиномиальным временем работы, то зачастую впоследствии разрабатывается и более эффективный алгоритм. Даже если самый лучший из известных на сегодняшний день алгоритмов решения задачи характеризуется временем $\Theta\left(n^{100}\right)$, достаточно велика вероятность того, что скоро будет разработан алгоритм с намного лучшим временем работы.

Во-вторых, для многих приемлемых вычислительных моделей задача, которая решается в течение полиномиального времени в одной модели, может быть решена в течение полиномиального времени и в другой. Например, в большей части книги рассматривается класс задач, разрешимых в течение полиномиального времени с помощью последовательных машин с произвольной выборкой. Этот класс совпадает с классом задач, разрешимых в течение полиномиального времени на абстрактных машинах Тьюринга[1]. Он также совпадает с классом задач, разрешимых в течение полиномиального времени на параллельных компьютерах, если зависимость количества процессоров от объема входных данных описывается полиномиальной функцией.

В-третьих, класс задач, разрешимых в течение полиномиального времени, обладает полезными свойствами замкнутости, поскольку множество полиномов замкнуто относительно операций сложения, умножения и композиции. Например, если выход одного алгоритма с полиномиальным временем работы соединить со входом другой такой задачи, то получим полиномиальный составной алгоритм. Если же в другом алгоритме с полиномиальным временем работы фиксированное количество раз вызываются подпрограммы с полиномиальным временем работы, то время работы такого составного алгоритма также является полиномиальным.

Абстрактные задачи

Чтобы получить понятие о классе задач, разрешимых в течение полиномиального времени, сначала необходимо формально описать само понятие “задача”. Определим ***абстрактную задачу*** (abstract problem) Q как бинарное отношение между множеством ***экземпляров*** (instances) задач I и множеством ***решений*** (solutions) задач S. Например, экземпляр задачи SHORTEST_PATH состоит из трех элементов: графа и двух вершин. Решением этой задачи является последовательность

[1]Подробное рассмотрение модели Тьюринга можно найти в книгах Хопкрофта (Hopcroft) и Ульмана (Ullman) [156], а также Льюиса (Lewis) и Пападимитриу (Papadimitriou) [204].

вершин графа; при этом пустая последовательность может означать, что искомого пути не существует. Сама задача SHORTEST_PATH представляет собой отношение, сопоставляющее каждому экземпляру графа и двум его вершинам кратчайший путь по графу, соединяющий эти две вершины. Поскольку кратчайший путь может быть не единственным, конкретный экземпляр задачи может иметь несколько решений.

Представленная выше формулировка абстрактной задачи носит более широкий характер, чем требуется для наших целей. Как мы уже убедились, теория NP-полноты ограничивается рассмотрением ***задач принятия решения*** (decision problems), требующих решения вида "да-нет". В этом случае абстрактную задачу принятия решения можно рассматривать как функцию, отображающую экземпляр множества I на множество решений $\{0, 1\}$. Например, задача принятия решения, соответствующая задаче SHORTEST_PATH, — рассмотренная выше задача PATH. Если $i = \langle G, u, v, k\rangle$ — экземпляр задачи принятия решения PATH, то равенство $\text{PATH}(i) = 1$ (да) выполняется, если количество ребер в кратчайшем пути из вершины u в вершину v не превышает k; в противном случае $\text{PATH}(i) = 0$ (нет). Многие абстрактные задачи являются не задачами принятия решения, а ***задачами оптимизации*** (optimization problems), в которых некоторое значение подлежит минимизации или максимизации. Однако ранее мы убедились, что обычно не составляет труда сформулировать задачу оптимизации как задачу принятия решения, которая не сложнее исходной.

Кодирование

Если абстрактная задача решается с помощью компьютерной программы, экземпляры задач необходимо представить в виде, понятном этой программе. ***Кодирование*** (encoding) множества S абстрактных объектов — это отображение e множества S на множество бинарных строк[2]. Например, всем известно, что натуральные числа $\mathbf{N} = \{1, 2, \ldots\}$ кодируются строками $\{1, 10, 11, \ldots\}$. В этой кодировке $e(17) = 10001$. Каждый, кто интересовался, как в компьютере представляются символы клавиатуры, знаком с кодами ASCII или кодами EBCDIC. В кодировке ASCII представление символа A выглядит как 1000001. В виде бинарной строки можно закодировать даже составной объект. Для этого конструируется комбинация, состоящая из представлений элементов, которые содержит в себе этот объект. Многоугольники, графы, функции, ориентированные пары, программы — все это можно закодировать бинарными строками.

Таким образом, компьютерный алгоритм, который "решает" некоторую абстрактную задачу принятия решения, фактически принимает в качестве входных данных закодированный экземпляр задачи. Назовем задачу, множество

[2]Область значений отображения e — не обязательно *бинарные* строки; подойдет любое множество строк, состоящих из символов конечного алфавита, содержащего не менее двух символов.

экземпляров которой является множеством бинарных строк, ***конкретной*** (concrete problem). Говорят, что алгоритм ***решает*** (solves) конкретную задачу в течение времени $O(T(n))$, если для заданного экземпляра задачи i длиной $n = |i|$ с его помощью можно получить решение в течение времени $O(T(n))$[3]. Поэтому конкретная задача ***разрешима в течение полиномиального времени*** (polynomial-time solvable), если существует алгоритм, позволяющий решить ее за время $O\left(n^k\right)$, где k — некоторая константа.

Теперь ***класс сложности*** **P** (complexity class P) можно формально определить как множество конкретных задач принятия решения, разрешимых в течение полиномиального времени.

С помощью кодирования абстрактные задачи можно отображать на конкретные. Данную абстрактную задачу принятия решения Q, отображающую множество экземпляров на множество $\{0,1\}$, с помощью кодирования $e : I \to \{0,1\}^*$ можно свести к связанной с ней конкретной задаче принятия решения, которая будет обозначаться как $e(Q)$[4]. Если $Q(i) \in \{0,1\}$ — решение экземпляра абстрактной задачи $i \in I$, то оно же является и решением экземпляра конкретной задачи $e(i) \in \{0,1\}^*$. Заметим, что некоторые бинарные строки могут не представлять осмысленного экземпляра абстрактной задачи. Для удобства мы будем предполагать, что любая такая строка произвольным образом отображается на 0. Таким образом, конкретная задача имеет те же решения, что и абстрактная, если ее экземпляры в виде бинарных строк представляют закодированные экземпляры абстрактной задачи.

Попытаемся расширить определение разрешимости в течение полиномиального времени для конкретных задач на абстрактные задачи, воспользовавшись в качестве связующего звена кодами; однако хотелось бы, чтобы определение не зависело от вида кодировки. Другими словами, эффективность решения задачи не должна зависеть от того, как она кодируется. К сожалению, на самом деле существует достаточно сильная зависимость от кодирования. Например, предположим, что в качестве входных данных алгоритма выступает только целое число k и что время работы этого алгоритма равно $\Theta(k)$. Если число k передается как ***унарное*** (unary), т.е. в виде строки, состоящей из k единиц, то время работы алгоритма для входных данных длины n равно $O(n)$ (выражается полиномиальной функцией). Если же используется более естественное бинарное представление числа k, то длина входной строки равна $n = \lfloor \lg k \rfloor + 1$. В этом случае время работы алгоритма равно $\Theta(k) = \Theta(2^n)$, т.е. зависит от объема входных данных как

[3] Предполагается, что выходные данные алгоритма отделены от его входных данных. Поскольку для получения каждого выходного бита требуется по крайней мере один элементарный временной интервал, а всего имеется $O(T(n))$ временных интервалов, объем выходных данных должен быть $O(T(n))$.

[4] Как вскоре станет понятно, $\{0,1\}^*$ обозначает множество всех строк, состоящих из символов множества $\{0,1\}$.

показательная функция. Таким образом, в зависимости от способа кодирования, алгоритм работает в течение полиномиального времени или превосходит его.

Поэтому кодирование абстрактной задачи — достаточно важный вопрос для нашего понимания полиномиального времени. Невозможно вести речь о решении абстрактной задачи, не представив подробного описания кодировки. Тем не менее, если отбросить такие "дорогостоящие" коды, как унарные, само кодирование задачи на практике будет мало влиять на то, разрешима ли задача в течение полиномиального времени. Например, если представить целые числа в троичной системе счисления, а не в двоичной, это не повлияет на то, разрешима ли задача в течение полиномиального времени, поскольку целое число в троичной системе счисления можно преобразовать в целое число в двоичной системе счисления в течение полиномиального времени.

Говорят, что функция $f : \{0,1\}^* \to \{0,1\}^*$ ***вычислима в течение полиномиального времени*** (polynomial-time computable), если существует алгоритм A с полиномиальным временем работы, который для произвольных входных данных $x \in \{0,1\}^*$ возвращает выходные данные $f(x)$. Для некоторого множества I, состоящего из экземпляров задач, две кодировки e_1 и e_2 называются ***полиномиально связанными*** (polynomially related), если существуют такие две вычислимые в течение полиномиального времени функции f_{12} и f_{21}, что для любого экземпляра $i \in I$ выполняются равенства $f_{12}(e_1(i)) = e_2(i)$ и $f_{21}(e_2(i)) = e_1(i)$[5]. Другими словами, закодированную величину $e_2(i)$ можно вычислить на основе закодированной величины $e_1(i)$ с помощью алгоритма с полиномиальным временем работы и наоборот. Если две кодировки e_1 и e_2 абстрактной задачи полиномиально связаны, то, как следует из представленной ниже леммы, разрешимость задачи в течение полиномиального времени не зависит от используемой кодировки.

Лемма 34.1. Пусть Q — абстрактная задача принятия решения, определенная на множестве экземпляров I, а e_1 и e_2 — полиномиально связанные кодировки множества I. В этом случае $e_1(Q) \in \mathrm{P}$ тогда и только тогда, когда $e_2(Q) \in \mathrm{P}$.

Доказательство. Достаточно доказать только прямое утверждение, поскольку обратное утверждение симметрично по отношению к прямому. Поэтому предположим, что задачу $e_1(Q)$ можно решить за время $O(n^k)$, где k — некоторая константа. Далее, предположим, что для любого экземпляра i задачи кодирование $e_1(i)$ можно получить из кодирования $e_2(i)$ за время $O(n^c)$, где c — некоторая константа, а $n = |e_2(i)|$. Чтобы решить задачу $e_2(Q)$ с входными данными

[5]Кроме того, накладывается техническое требование, чтобы функции f_{12} и f_{21} "отображали неэкземпляры в неэкземпляры". Неэкземпляр (noninstance) в кодировке e — это такая строка $x \in \{0,1\}^*$, что не существует экземпляра i, для которого $e(i) = x$. Потребуем, чтобы равенство $f_{12}(x) = y$ выполнялось для каждого неэкземпляра x в кодировке e_1, где y — некоторый неэкземпляр в кодировке e_2, а равенство $f_{21}(x') = y'$ — для каждого неэкземпляра x' в кодировке e_2, где y' — некоторый неэкземпляр в кодировке e_1.

$e_2(i)$, сначала вычисляется $e_1(i)$, после чего алгоритм запускается для решения задачи $e_1(Q)$ с входными данными $e_1(i)$. Сколько времени это займет? Для преобразования кодировки требуется время $O(n^c)$, поэтому выполняется равенство $|e_1(i)| = O(n^c)$, поскольку объем выходных данных алгоритма на последовательном компьютере не может превосходить по величине время его работы. Решение задачи с входными данными $e_1(i)$ занимает время $O\left(|e_1(i)|^k\right) = O\left(n^{ck}\right)$, которое является полиномиальным, поскольку и c, и k — константы. ■

Таким образом, тот факт, в каком виде закодированы экземпляры абстрактной задачи — в двоичной системе счисления или троичной, — не влияет на ее "сложность", т.е. на то, разрешима она в течение полиномиального времени или нет. Однако если эти экземпляры имеют унарную кодировку, сложность задачи может измениться. Чтобы иметь возможность осуществлять преобразование независимым от кодировки образом, в общем случае предполагается, что экземпляры задачи закодированы в произвольном рациональном и сжатом виде, если специально не оговаривается противное. Точнее говоря, предполагается, что кодирование целых чисел полиномиально связано с их бинарным представлением и что кодирование ограниченного множества полиномиально связано с его кодированием в виде списка элементов, взятых в скобки и разделенных запятыми. (Одна из таких схем кодирования — ASCII.) Располагая таким "стандартным" кодом, можно получить рациональный код других математических объектов, таких как кортежи, графы и формулы. Для обозначения стандартного кода объекта этот объект будет заключаться в угловые скобки. Таким образом, $\langle G \rangle$ обозначает стандартный код графа G.

До тех пор пока неявно используется код, полиномиально связанный с таким стандартным кодом, можно прямо говорить об абстрактных задачах, не ссылаясь при этом на какой-то отдельный код, зная, что выбор кода не повлияет на разрешимость абстрактной задачи в течение полиномиального времени. С этого момента в общем случае предполагается, что все экземпляры задачи представляют собой бинарные строки, закодированные с помощью стандартного кодирования, если явно не оговаривается противное. Кроме того, в большинстве случаев мы будем пренебрегать различием между абстрактными и конкретными задачами. Однако на практике читателю следует остерегаться тех возникающих на практике задач, в которых стандартное кодирование не очевидно и выбор кодирования имеет значение.

Структура формальных языков

Один из аргументов, свидетельствующих в пользу удобства задач принятия решения, заключается в том, что для них можно использовать алгоритмы, позаимствованные из теории формальных языков. Здесь стоит привести некоторые

определения из этой теории. ***Алфавит*** Σ (alphabet Σ) — это конечное множество символов. ***Язык*** L (language L), определенный над множеством Σ, — это произвольное множество строк, состоящих из символов из множества Σ. Например, если $\Sigma = \{0, 1\}$, то множество $L = \{10, 11, 101, 111, 1011, \ldots\}$ является языком бинарного представления простых чисел. Обозначим ***пустую строку*** (empty string) через ε, а ***пустой язык*** (empty language) — через $\emptyset$. Язык всех строк, заданных над множеством Σ, обозначается через Σ^*. Например, если $\Sigma = \{0, 1\}$, то $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, \ldots\}$ — множество всех бинарных строк. Любой язык L над множеством Σ является подмножеством множества Σ^*.

Над языками можно определить ряд операций. Наличие таких операций из теории множеств, как ***объединение*** (union) и ***пересечение*** (intersection), непосредственно следует из теоретико-множественной природы определения языков. ***Дополнение*** (complement) языка L определим с помощью соотношения $\bar{L} = \Sigma^* - L$. ***Конкатенацией*** (concatenation) двух языков L_1 и L_2 является язык

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ и } x_2 \in L_2\}.$$

Замыканием (closure), или ***замыканием Клини*** (Kleene star), языка L называется язык

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots,$$

где L^k — язык, полученный в результате k-кратной конкатенации языка L с самим собой.

С точки зрения теории языков множество экземпляров любой задачи принятия решения Q — это просто множество Σ^*, где $\Sigma = \{0, 1\}$. Поскольку множество Q полностью характеризуется теми экземплярами задачи, в которых выдается ответ 1 (да), это множество можно рассматривать как язык L над множеством $\Sigma = \{0, 1\}$, где

$$L = \{x \in \Sigma^* : Q(x) = 1\}.$$

Например, задаче принятия решения PATH соответствует язык

$$\begin{aligned} \text{PATH} = \{\langle G, u, v, k\rangle : {} & G = (V, E) \text{ — неориентированный граф,} \\ & u, v \in V, \\ & k \geqslant 0 \text{ — целое число, и} \\ & \text{в } G \text{ существует путь от } u \text{ до } v, \text{ состоящий} \\ & \text{не более чем из } k \text{ ребер}\}. \end{aligned}$$

(Для удобства иногда одно и то же имя (в данном случае это имя PATH) будет употребляться и для задачи принятия решения, и для соответствующего ей языка.)

Схема формальных языков позволяет в сжатом виде выразить взаимоотношение между задачами принятия решения и алгоритмами их решения. Говорят, что

алгоритм A ***принимает*** (accepts) строку $x \in \{0,1\}^*$, если для заданных входных данных x выход алгоритма $A(x)$ равен 1. Язык, ***принимаемый*** (accepted) алгоритмом A — это множество строк $L = \{x \in \{0,1\}^* : A(x) = 1\}$, т.е. множество строк, воспринимаемых алгоритмом. Алгоритм A ***отвергает*** (rejects) строку x, если $A(x) = 0$.

Если язык L принимается алгоритмом A, этот алгоритм не обязательно отвергает строку $x \notin L$, предоставляемую в качестве его входных данных. Например, алгоритм может быть организован в виде бесконечного цикла. Язык L ***распознается*** (decided) алгоритмом A, если каждая бинарная строка этого языка принимается алгоритмом A, а каждая бинарная строка, которая не принадлежит языку L, отвергается этим алгоритмом. Язык L ***принимается за полиномиальное время*** (accepted in polynomial time) алгоритмом A, если он принимается алгоритмом A и, кроме того, если существует такая константа k, что для любой n-символьной строки $x \in L$ алгоритм A принимает строку x за время $O\left(n^k\right)$. Язык L ***распознается за полиномиальное время*** (decided in polynomial time) алгоритмом A, если существует такая константа k, что для любой n-символьной строки $x \in \{0,1\}^*$ алгоритм за время $O\left(n^k\right)$ правильно выясняет, принадлежит ли строка x языку L. Таким образом, чтобы принять язык, алгоритму нужно заботиться только о строках языка L, а чтобы распознать язык, он должен корректно принять или отвергнуть каждую строку из множества $\{0,1\}^*$.

Например, язык PATH может быть принят в течение полиномиального времени. Один из принимающих алгоритмов с полиномиальным временем работы проверяет, что объект G закодирован как неориентированный граф, убеждается, что u и v — его вершины, с помощью поиска в ширину находит в графе G кратчайший путь из вершины u к вершине v, а затем сравнивает количество ребер в полученном кратчайшем пути с числом k. Если в объекте G закодирован неориентированный граф, и путь из вершины u к вершине v содержит не более k ребер, алгоритм выводит значение 1 и останавливается. В противном случае он работает бесконечно долго. Однако такой алгоритм не распознает язык PATH, поскольку он не выводит явно значение 0 для экземпляров, длина кратчайшего пути в которых превышает k ребер. Предназначенный для задачи PATH алгоритм распознавания должен явно отвергать бинарные строки, не принадлежащие языку PATH. Для такой задачи принятия решения, как задача PATH, подобный алгоритм распознавания разработать легко: вместо того чтобы работать бесконечно долго, если не существует пути из вершины u в вершину v, количество ребер в котором не превышает k, этот алгоритм должен выводить значение 0 и останавливаться. Для других задач, таких как задача останова Тьюринга, принимающий алгоритм существует, а алгоритма распознавания не существует.

Можно неформально определить ***класс сложности*** (complexity class) как множество языков, принадлежность к которому определяется ***мерой сложности*** (complexity measure), такой как время работы алгоритма, определяющего, принад-

лежит ли данная строка x языку L. Фактическое определение класса сложности носит более технический характер. Интересующийся читатель найдет его в статье Хартманиса (Hartmanis) и Стирнса (Stearns) [140].

Воспользовавшись описанным выше формализмом теории языков, можно дать альтернативное определение класса сложности P:

$$\mathrm{P} = \{L \subseteq \{0,1\}^* : \text{существует алгоритм } A, \text{ разрешающий язык } L \text{ за полиномиальное время}\}.$$

Фактически, P также является классом языков, которые могут быть приняты за полиномиальное время.

Теорема 34.2.

$$\mathrm{P} = \{L : L \text{ принимается алгоритмом с полиномиальным временем работы}\}.$$

Доказательство. Поскольку класс языков, которые распознаются алгоритмами с полиномиальным временем работы, — это подмножество класса языков, которые принимаются алгоритмами с полиномиальным временем работы, остается лишь показать, что если язык L принимается алгоритмом с полиномиальным временем работы, то он также распознается алгоритмом с полиномиальным временем работы. Пусть L — язык, который принимается некоторым алгоритмом с полиномиальным временем работы A. Воспользуемся классическим "модельным" аргументом, чтобы сконструировать другой алгоритм с полиномиальным временем работы A', который бы распознавал язык L. Поскольку алгоритм A принимает язык L за время $O\left(n^k\right)$, где k — некоторая константа, существует такая константа c, что алгоритм A принимает язык L не более чем за $T = cn^k$ шагов. Для любой входной строки x алгоритм A' моделирует действие алгоритма A за время T. По истечении интервала T алгоритм A' проверяет поведение алгоритма A. Если он принял строку x, то алгоритм A' также принимает эту строку, выводя значение 1. Если же алгоритм A не принял строку x, то алгоритм A' отвергает эту строку, выводя значение 0. Накладные расходы алгоритма A', моделирующего алгоритм A, приводят к увеличению времени работы не более чем на полиномиальный множитель, поэтому A' — алгоритм с полиномиальным временем работы, распознающий язык L. ■

Заметим, что доказательство теоремы 34.2 является неконструктивным. Для данного языка $L \in P$ граница для времени работы алгоритма A, который принимает язык L, на самом деле может быть неизвестна. Тем не менее, известно, что такая граница существует, поэтому существует алгоритм A', способный проверить эту границу, даже если не всегда удается легко найти такой алгоритм.

Упражнения

34.1-1. Определим задачу оптимизации LONGEST_PATH_LENGTH как отношение, связывающее каждый экземпляр задачи, состоящий из неориентированного графа и двух его вершин, с количеством ребер в самом длинном пути между этими двумя вершинами. Определим задачу принятия решения LONGEST_PATH_LENGTH $= \{\langle G, u, v, k\rangle : G = (V, E) -$ неориентированный граф, $u, v \in V$, $k \geqslant 0 -$ целое число, существует путь из вершины u в вершину v, состоящий не менее чем из k ребер$\}$. Покажите, что задачу оптимизации LONGEST_PATH_LENGTH можно решить за полиномиальное время тогда и только тогда, когда LONGEST_PATH_LENGTH $\in$ P.

34.1-2. Дайте формальное определение задачи поиска самого длинного простого цикла в неориентированном графе. Сформулируйте соответствующую задачу принятия решения. Опишите язык, соответствующий этой задаче принятия решения.

34.1-3. Разработайте формальное кодирование для ориентированного графа в виде бинарных строк для представления при помощи матриц смежности. Выполните то же самое для представления с использованием списка смежных вершин. Покажите, что эти два представления полиномиально связаны.

34.1-4. Является ли алгоритм динамического программирования для решения дискретной задачи о рюкзаке, который предлагалось разработать в задаче 16.2-2, алгоритмом с полиномиальным временем работы? Обоснуйте ваш ответ.

34.1-5. Покажите, что алгоритм, который содержит не больше некоторого постоянного количества вызовов процедуры с полиномиальным временем работы, сам работает полиномиальное время. Если же алгоритм делает полиномиальное число вызовов такой процедуры, то общее время работы может быть экспоненциальным.

34.1-6. Покажите, что класс P, который рассматривается как множество языков, замкнут относительно операций объединения, пересечения, конкатенации, дополнения и замыкания Клини. Другими словами, если $L_1, L_2 \in$ P, то $L_1 \cup L_2 \in$ P и т.д.

34.2 Проверка за полиномиальное время

Теперь рассмотрим алгоритм, "проверяющий" принадлежность языку. Например, предположим, что в данном экземпляре $\langle G, u, v, k\rangle$ задачи принятия решения PATH задан также путь p из вершины u в вершину v. Легко проверить, превышает ли длина пути p величину k. Если она не превышает эту величину, путь

p можно рассматривать как “сертификат” того, что данный экземпляр действительно принадлежит PATH. Для задачи принятия решения PATH такой сертификат, по-видимому, не дает ощутимых преимуществ. В конце концов, эта задача принадлежит классу P (фактически ее можно решить за линейное время), поэтому проверка принадлежности с помощью такого сертификата занимает столько же времени, сколько и решение задачи. А теперь исследуем задачу, для которой пока неизвестен алгоритм принятия решения с полиномиальным временем работы, но если имеется сертификат, то легко выполнить его проверку.

Гамильтоновы циклы

Задача поиска гамильтоновых циклов в неориентированном графе изучается уже более ста лет. Формально ***гамильтонов цикл*** (hamiltonian cycle) неориентированного графа $G = (V, E)$ — это простой цикл, содержащий все вершины множества V. Граф, содержащий гамильтонов цикл, называют ***гамильтоновым*** (hamiltonian); в противном случае он является ***негамильтоновым*** (nonhamiltonian). В книге Бонди (Bondy) и Мьюрти (Murty) [45] цитируется письмо Гамильтона (W.R. Hamilton) с описанием математической игры на додекаэдре (рис. 34.2*а*), в которой один игрок отмечает пятью булавками пять произвольных последовательных вершин, а другой игрок должен дополнить этот путь, чтобы в результате получился путь, содержащий все вершины. Додекаэдр является гамильтоновым графом, и один из гамильтоновых циклов показан на рис. 34.2*а*. Однако не все графы являются гамильтоновыми. Например, на рис. 34.2*б* показан двудольный граф

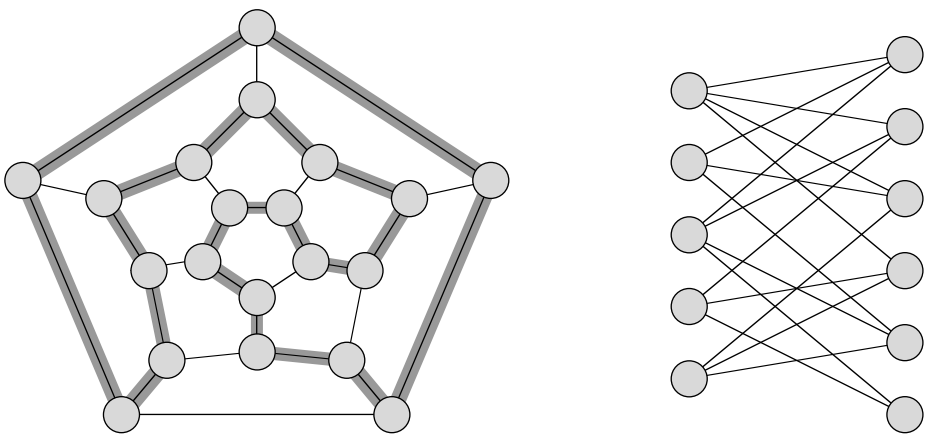


Рис. 34.2. Примеры гамильтонового (*а*) и негамильтонового (*б*) графов

с нечетным количеством вершин. В упражнении 34.2-2 предлагается доказать, что все такие графы негамильтоновы.

Задачу о гамильтоновых циклах (hamiltonian-cycle problem), в которой нужно определить, содержит ли граф G гамильтонов цикл, можно определить как формальный язык:

$$\text{HAM\_CYCLE} = \{\langle G\rangle : G \text{ — гамильтонов граф}\}.$$

Как алгоритм мог бы распознать язык HAM_CYCLE? Для заданного экземпляра задачи $\langle G\rangle$ можно предложить алгоритм принятия решения, который бы формировал список всех перестановок вершин графа G, а затем проверял бы каждую перестановку, не является ли она гамильтоновым путем. Чему равнялось бы время работы такого алгоритма? При использовании "рационального" кодирования графа с использованием матриц смежности количество вершин m графа равно $\Omega(\sqrt{n})$, где $n = |\langle G\rangle|$ — длина кода для графа G. Всего имеется $m!$ возможных перестановок вершин, поэтому время работы алгоритма равно $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega\left(2^{\sqrt{n}}\right)$, и оно не ведет себя в асимптотическом пределе как $O\left(n^k\right)$ ни для какой константы k. Таким образом, время работы подобного прямолинейного алгоритма не выражается полиномиальной функцией. Фактически, как будет доказано в разделе 34.5, задача о гамильтоновых циклах является NP-полной.

Алгоритмы верификации

Рассмотрим несколько упрощенную задачу. Предположим, что приятель сообщил вам, что данный граф G — гамильтонов, а затем предложил доказать это, предоставив последовательность вершин, образующих гамильтонов цикл. Очевидно, в такой ситуации доказательство было бы достаточно простым: следует просто проверить, что предоставленный цикл является гамильтоновым, убедившись, что данная последовательность вершин является перестановкой множества всех вершин V и что каждое встречающееся в цикле ребро действительно принадлежит графу. Легко понять, что подобный алгоритм верификации можно реализовать так, чтобы время его работы было равно $O\left(n^2\right)$, где n — длина графа G в закодированном виде. Таким образом, доказательство того, что гамильтонов цикл в графе существует, можно проверить в течение полиномиального времени.

Определим ***алгоритм верификации*** (verification algorithm) как алгоритм A с двумя аргументами, один из которых является обычной входной строкой x, а второй — бинарной строкой y под названием ***сертификат*** (certificate). Двухаргументный алгоритм A ***верифицирует*** (verifies) входную строку x, если существует сертификат y, удовлетворяющий уравнению $A(x, y) = 1$. ***Язык, проверенный*** алгоритмом верификации A, — это множество

$$L = \{x \in \{0,1\}^* : \text{существует } y \in \{0,1\}^* \text{ такой, что } A(x, y) = 1\}.$$

Интуитивно понятно, что алгоритм A верифицирует язык L, если для любой строки $x \in L$ существует сертификат y, позволяющий доказать с помощью алгоритма A, что $x \in L$. Кроме того, для любой строки $x \notin L$ не должно существовать сертификата, доказывающего, что $x \in L$. Например, в задаче о гамильтоновом цикле сертификатом является список вершин некоторого гамильтонового цикла. Если граф гамильтонов, то гамильтонов цикл сам по себе предоставляет достаточно информации для верификации этого факта. В обратном случае, т.е. если граф не является гамильтоновым, то не существует списка вершин, позволяющего обмануть алгоритм верификации и установить, что граф является гамильтоновым, так как алгоритм верификации производит тщательную проверку предложенного "цикла", чтобы убедиться в его правильности.

Класс сложности NP

Класс сложности **NP** (complexity class NP) — это класс языков, которые можно верифицировать с помощью алгоритма с полиномиальным временем работы[6]. Точнее говоря, язык принадлежит классу NP тогда и только тогда, когда существует алгоритм A с двумя входными параметрами и полиномиальным временем работы, а также константа c, такая что

$$L = \{x \in \{0,1\}^* : \text{существует сертификат } y \ (|y| = O(|x|^c)) \text{ такой, что } A(x,y) = 1\}.$$

При этом говорят, что алгоритм A ***верифицирует*** (verify) язык L ***за полиномиальное время*** (in polynomial time).

Из проводимого ранее обсуждения задачи о гамильтоновых циклах следует, что задача HAM_CYCLE $\in$ NP (всегда приятно знать, что некоторое важное множество — не пустое). Кроме того, если $L \in \mathrm{P}$, то $L \in \mathrm{NP}$, поскольку при наличии алгоритма с полиномиальным временем работы, способного распознать язык L, алгоритм легко преобразовать в алгоритм верификации с двумя аргументами, который просто игнорирует сертификат и принимает именно те входные строки, для которых он устанавливает принадлежность языку L. Таким образом, $\mathrm{P} \subseteq \mathrm{NP}$.

Не известно, выполняется ли равенство $\mathrm{P} = \mathrm{NP}$, но, по мнению большинства исследователей, классы P и NP — не одно и то же. Интуитивно понятно, что класс P состоит из задач, которые решаются быстро. Класс же NP состоит из задач, решение которых можно быстро проверить. Возможно, из опыта вы уже знаете, что

[6] Название "NP" обозначает "nondeterministic polynomial time" (недетерминистическое полиномиальное время). Класс NP изначально изучался в контексте недетерминизма, но нами используется более простое, хотя и эквивалентное понятие верификации. В книге Хопкрофта (Hopcroft) и Ульмана (Ullman) [156] хорошо изложено понятие NP-полноты в терминах недетерминистических вычислительных моделей.

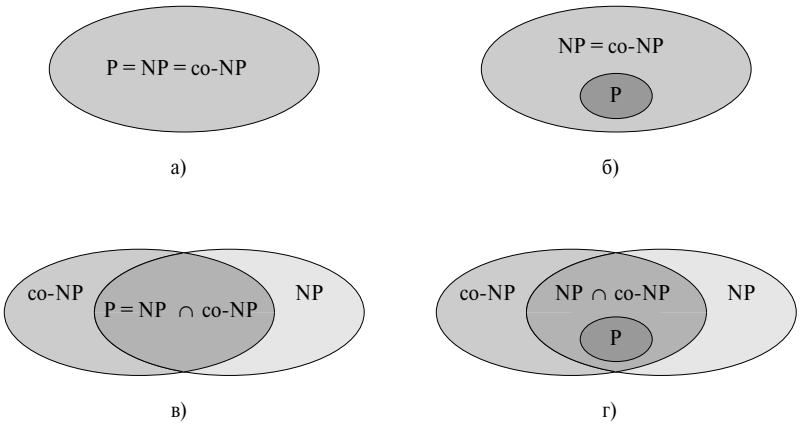


Рис. 34.3. Четыре возможных вида отношений между классами сложности

часто сложнее решить задачу с самого начала, чем проверить представленное решение, особенно если вы ограничены во времени. Среди ученых, занимающихся теорией вычислительных систем, общепринято считать, что эта аналогия распространяется на классы P и NP и, следовательно, что класс NP содержит языки, не принадлежащие классу P.

Существует более веское свидетельство того, что $\mathrm{P} \neq \mathrm{NP}$, — наличие языков, являющихся "NP-полными". Класс этих задач исследуется в разделе 34.3.

Помимо вопроса о соблюдении соотношения $\mathrm{P} \neq \mathrm{NP}$, остаются нерешенными многие другие фундаментальные вопросы. Несмотря на интенсивные исследования в этой области, никому не удалось установить, замкнут ли класс NP относительно операции дополнения. Другими словами, следует ли из соотношения $L \in \mathrm{NP}$ соотношение $\bar{L} \in \mathrm{NP}$? Можно определить ***класс сложности*** **co-NP** (complexity class co-NP) как множество языков L, для которых выполняется соотношение $\bar{L} \in \mathrm{NP}$. Вопрос о том, замкнут ли класс NP относительно дополнения, можно перефразировать как вопрос о соблюдении равенства $\mathrm{NP} = \mathrm{co\text{-}NP}$. Поскольку класс P замкнут относительно дополнения (упражнение 34.1-6), отсюда следует, что $P \subseteq \mathrm{NP} \cap \mathrm{co\text{-}NP}$. Однако, опять же не известно, выполняется ли равенство $\mathrm{P} = \mathrm{NP} \cap \mathrm{co\text{-}NP}$, т.е. существует ли хоть один язык в множестве $\mathrm{NP} \cap \mathrm{co\text{-}NP} - \mathrm{P}$. На рис. 34.3 представлены четыре возможные ситуации. На каждой диаграмме одна область, содержащая другую, указывает, что внутренняя область является собственным подмножеством внешней. В части *а* рисунка показана ситуация, соответствующая равенству $\mathrm{P} = \mathrm{NP} = \mathrm{co\text{-}NP}$. Большинство исследователей считает этот сценарий наименее вероятным. В части *б* показано, что если класс NP замкнут относительно дополнения, то $\mathrm{NP} = \mathrm{co\text{-}NP}$, но не обязательно справедливо равенство $\mathrm{P} = \mathrm{NP}$. В части *в* изображена ситуация, когда $\mathrm{P} = \mathrm{NP} \cap \mathrm{co\text{-}NP}$, но

класс NP не замкнут относительно дополнения. И наконец, в части *г* выполняются соотношения $\mathrm{NP} \neq \text{co-NP}$ и $\mathrm{P} \neq \mathrm{NP} \cap \text{co-NP}$. Большинство исследователей считают эту ситуацию наиболее вероятной.

Таким образом, к сожалению, наше понимание того, какие именно взаимоотношения существуют между классами P и NP, далеко не полные. Тем не менее, в процессе изучения теории NP-полноты станет понятно, что с практической точки зрения недостатки доказательства трудноразрешимости задачи не играют такой важной роли, как можно было бы ожидать.

Упражнения

34.2-1. Рассмотрим язык

$$\textsc{Graph\_Isomorphism} = \{\langle G_1, G_2\rangle : G_1 \text{ и } G_2 - \text{ изоморфные графы}\}.$$

Докажите, что $\textsc{Graph\_Isomorphism} \in \mathrm{NP}$, описав алгоритм с полиномиальным временем работы, позволяющий верифицировать этот язык.

34.2-2. Докажите, что если G — неориентированный двудольный граф с нечетным количеством вершин, то он не является гамильтоновым.

34.2-3. Покажите, что если $\textsc{Ham\_Cycle} \in \mathrm{P}$, то задача о выводе списка вершин гамильтонового цикла в порядке их обхода разрешима в течение полиномиального времени.

34.2-4. Докажите, что класс языков NP замкнут относительно операций объединения, пересечения, конкатенации и замыкания Клини. Обсудите замкнутость класса NP относительно дополнения.

34.2-5. Покажите, что любой язык из класса NP можно распознать с помощью алгоритма, время работы которого равно $2^{O(n^k)}$, где k — некоторая константа.

34.2-6. ***Гамильтонов путь*** (hamiltonian path) графа — это простой путь, который проходит через каждую вершину ровно по одному разу. Покажите, что язык

$$\textsc{Ham\_Path} = \{\langle G, u, v\rangle : \text{в графе } G \text{ имеется гамильтонов путь от } u \text{ к } v\}$$

принадлежит классу NP.

34.2-7. Покажите, что задачу о гамильтоновом пути в ориентированном ациклическом графе можно решить в течение полиномиального времени. Сформулируйте эффективный алгоритм решения этой задачи.

34.2-8. Пусть ϕ — булева формула, составленная из булевых входных переменных $x_1, x_2, \ldots, x_k$, операторов НЕ ($\neg$), И ($\wedge$), ИЛИ ($\vee$) и скобок. Формула

ϕ называется ***тавтологией*** (tautology), если для всех возможных наборов входных переменных в результате вычисления формулы получается значение 1. Определите язык булевых формул TAUTOLOGY, состоящий из тавтологий. Покажите, что TAUTOLOGY $\in$ co-NP.

34.2-9. Докажите, что $\mathrm{P} \subseteq \text{co-NP}$.

34.2-10. Докажите, что если $\mathrm{NP} \neq \text{co-NP}$, то $\mathrm{P} \neq \mathrm{NP}$.

34.2-11. Пусть G — связный неориентированный граф, содержащий не менее трех вершин, а G^3 — граф, полученный путем соединения всех пар вершин, которые связаны в графе G путем, длина которого не превышает 3 ребер. Докажите, что граф G^3 гамильтонов. (*Указание*: постройте остовное дерево графа G и воспользуйтесь индукцией.)

34.3 NP-полнота и приводимость

По-видимому, одна из веских причин, по которым специалисты в области теории вычислительных систем полагают, что $\mathrm{P} \neq \mathrm{NP}$, — наличие класса "NP-полных" задач. Этот класс обладает замечательным свойством, которое состоит в том, что если хоть одну (*любую*) NP-полную задачу можно решить в течение полиномиального времени, то и *все* задачи этого класса обладают полиномиально-временным решением, т.е. $\mathrm{P} = \mathrm{NP}$. Однако, несмотря на многолетние исследования, до сих пор не обнаружено ни одного алгоритма с полиномиальным временем работы ни для одной NP-полной задачи.

Язык HAM_CYCLE — одна из NP-полных задач. Если бы этот язык можно было бы распознать за полиномиальное время, то каждую задачу из класса NP можно было бы решить в течение полиномиального времени. Фактически, если бы множество $\mathrm{NP} - \mathrm{P}$ оказалось непустым, то с уверенностью можно было бы утверждать, что HAM_CYCLE $\in \mathrm{NP} - \mathrm{P}$.

В определенном смысле NP-полные языки — "самые сложные" в классе NP. В этом разделе будет показано, как относительная "сложность" языков сравнивается с помощью точного понятия под названием "приводимость к полиномиальному времени". Затем приводится формальное определение NP-полных языков, а в конце раздела дается набросок доказательства того, что один из таких языков с именем CIRCUIT_SAT, является NP-полным. В разделах 34.4 и 34.5 с помощью понятия приводимости демонстрируется, что многие другие задачи также NP-полные.

Приводимость

Интуитивно понятно, что задачу Q можно свести к другой задаче Q', если любой экземпляр задачи Q "легко перефразируется" в экземпляр задачи Q', решение

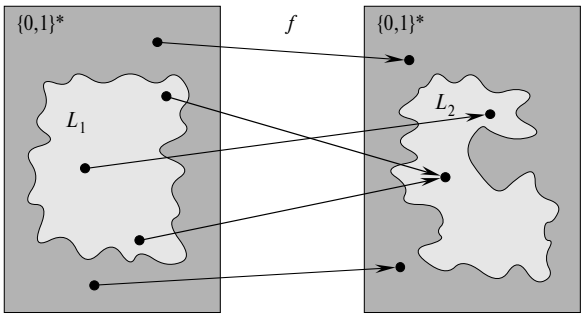


Рис. 34.4. Иллюстрация к приведению в течение полиномиального времени языка L_1 к языку L_2 с помощью функции приведения f

которого позволяет получить решение соответствующего экземпляра задачи Q. Например, задача решения линейных уравнений относительно неизвестной величины x сводится к задаче решения квадратных уравнений. Если задан экземпляр $ax + b = 0$, его можно преобразовать в уравнение $0x^2 + ax + b = 0$, решение которого совпадает с решением уравнения $ax + b = 0$. Таким образом, если задача Q сводится к другой задаче Q', то решить задачу Q в некотором смысле "не сложнее", чем задачу Q'.

Возвратимся к применению системы формальных языков для решения задач. Говорят, что язык L_1 ***приводим в течение полиномиального времени*** (polynomial-time reducible) к языку L_2 (что обозначается как $L_1 \leqslant_P L_2$), если существует вычислимая за полиномиальное время функция $f : \{0,1\}^* \to \{0,1\}^*$, такая что для всех $x \in \{0,1\}^*$

$$x \in L_1 \text{ тогда и только тогда, когда } f(x) \in L_2. \tag{34.1}$$

Функцию f называют ***функцией приведения*** (reduction function), а алгоритм F с полиномиальным временем работы, вычисляющий функцию f, — ***алгоритмом приведения*** (reduction algorithm).

Рис. 34.4 иллюстрирует понятие приведения языка L_1 к другому языку L_2 в течение полиномиального времени. Каждый язык — это подмножество множества $\{0,1\}^*$. Функция приведения f обеспечивает такое отображение в течение полиномиального времени, что если $x \in L_1$, то $f(x) \in L_2$. Более того, если $x \notin L_1$, то $f(x) \notin L_2$. Таким образом, функция приведения отображает любой экземпляр x задачи принятия решения, представленной языком L_1, на экземпляр $f(x)$ задачи, представленной языком L_2. Ответ на вопрос о том, принадлежит ли экземпляр $f(x)$ языку L_2, непосредственно позволяет ответить на вопрос о том, принадлежит ли экземпляр x языку L_1.

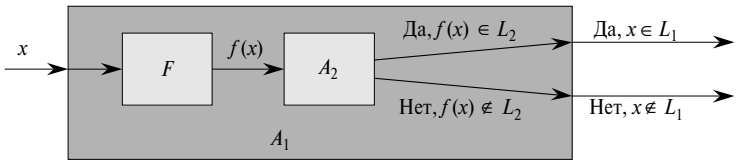


Рис. 34.5. Иллюстрация к доказательству леммы 34.3

Приведение в течение полиномиального времени служит мощным инструментом доказательства того, что различные языки принадлежат классу P.

Лемма 34.3. Если $L_1, L_2 \subseteq \{0,1\}^*$ — языки, такие что $L_1 \leqslant_{\mathrm{P}} L_2$, то из $L_2 \in \mathrm{P}$ следует $L_1 \in \mathrm{P}$.

Доказательство. Пусть A_2 — алгоритм с полиномиальным временем выполнения, который распознает язык L_2, а F — полиномиально-временной алгоритм приведения, вычисляющий приводящую функцию f. Построим алгоритм A_1 с полиномиальным временем выполнения, который распознает язык L_1.

На рис. 34.5 проиллюстрирован процесс построения алгоритма A_1. Для заданного набора входных данных $x \in \{0,1\}^*$ в алгоритме A_1 с помощью алгоритма F данные x преобразуются в $f(x)$, после чего с помощью алгоритма A_2 проверяется, является ли $f(x) \in L_2$. Результат работы алгоритма A_2 выводится алгоритмом A_1 в качестве ответа.

Корректность алгоритма A_1 следует из условия (34.1). Алгоритм выполняется в течение полиномиального времени, поскольку и алгоритм F, и алгоритм A_2 завершают свою работу за полиномиальное время (см. упражнение 34.1-5). ■

NP-полнота

Приведения в течение полиномиального времени служат формальным средством, позволяющим показать, что одна задача такая же по сложности, как и другая, по крайней мере с точностью до полиномиально-временного множителя. Другими словами, если $L_1 \leqslant_{\mathrm{P}} L_2$, то сложность языка L_1 превышает сложность языка L_2 не более чем на полиномиальный множитель. Вот почему отношение “меньше или равно” для приведения является мнемоническим. Теперь можно определить множество NP-полных языков, являющихся самыми сложными задачами класса NP.

Язык $L \subseteq \{0,1\}^*$ является ***NP-полным*** (NP-complete), если выполняются перечисленные ниже условия:

1. $L \in \mathrm{NP}$, и
2. $L' \leqslant_{\mathrm{P}} L$ для всех $L' \in \mathrm{NP}$.

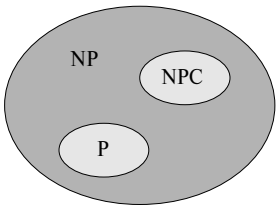


Рис. 34.6. Представление большинства специалистов в области вычислительных систем об отношении между классами P, NP и NPC; классы P и NPC полностью содержатся в классе NP, и $\mathrm{P} \cap \mathrm{NPC} = \emptyset$

Если язык L удовлетворяет свойству 2, но не обязательно удовлетворяет свойству 1, говорят, что L — ***NP-сложный*** (NP-hard). Определим также класс NPC как класс NP-полных языков.

Как показано в приведенной далее теореме, NP-полнота — ключевая проблема в разрешении вопроса о том, равны ли на самом деле классы P и NP.

Теорема 34.4. Если некоторая NP-полная задача разрешима в течение полиномиального времени, то $\mathrm{P} = \mathrm{NP}$. Это эквивалентно утверждению, что если какая-нибудь задача из класса NP не решается в течение полиномиального времени, то никакая из NP-полных задач не решается в течение полиномиального времени.

Доказательство. Предположим, что $L \in \mathrm{P}$ и что $L \in \mathrm{NPC}$. Для любого языка $L' \in \in \mathrm{NP}$, согласно свойству 2 определения NP-полноты, справедливо $L' \leqslant_{\mathrm{P}} L$. Таким образом, в соответствии с леммой 34.3, мы также имеем $L' \in \mathrm{P}$, что и доказывает первое утверждение теоремы.

Чтобы доказать второе утверждение, заметим, что оно является переформулировкой первого. ■

Вот почему при исследовании вопроса $\mathrm{P} \neq \mathrm{NP}$ внимание акцентируется на NP-полных задачах. Большинство специалистов по теории вычислительных систем полагают, что $\mathrm{P} \neq \mathrm{NP}$, так что отношения между классами P, NP и NPC являются такими, как показано на рис. 34.6. Но если кто-нибудь додумается до алгоритма с полиномиальным временем работы, способного решить NP-полную задачу, тем самым будет доказано, что $\mathrm{P} = \mathrm{NP}$. Тем не менее, поскольку до сих пор такой алгоритм не обнаружен ни для одной NP-полной задачи, доказательство NP-полноты задачи является веским свидетельством ее трудноразрешимости.

Выполнимость схем

Определение NP-полноты уже сформулировано, но до сих пор не была доказана NP-полнота ни одной задачи. Как только это будет сделано хотя бы для одной задачи, с помощью приводимости в течение полиномиального времени можно будет доказать NP-полноту других задач. Таким образом, сосредоточим внимание на том, чтобы продемонстрировать NP-полноту задачи о выполнимости схем.

К сожалению, для формального доказательства того, что задача о выполнимости схем является NP-полной, требуются технические детали, выходящие за рамки настоящей книги. Вместо этого мы дадим неформальное описание доказательства, основанного на базовом понимании булевых комбинационных схем.

Булевы комбинационные схемы составляются из булевых комбинационных элементов, соединенных между собой проводами. ***Булев комбинационный элемент*** (boolean combinational element), или ***комбинационный логический элемент***, — это любой элемент сети, обладающий фиксированным количеством булевых входов и выходов, который выполняет вполне определенную функцию. Булевы значения извлекаются из множества $\{0, 1\}$, где 0 представляет значение FALSE, а 1 — значение TRUE.

Комбинационные логические элементы, используемые в задаче о выполнимости схем, вычисляют простую булеву функцию, и они известны как ***логические вентили*** (logic gates). На рис. 34.7 показаны три основных логических элемента, использующихся в задаче о выполнимости схем: ***логический вентиль НЕ*** (NOT gate), или ***инвертор*** (inverter), ***логический вентиль И*** (AND gate) и ***логический вентиль ИЛИ*** (OR gate). На вентиль НЕ поступает одна бинарная ***входная величина*** (input) x, значение которой равно 0 или 1; вентиль выдает бинарную ***выходную величину*** (output) z, значение которой противоположно значению входной величины. На каждый из двух других вентилей поступают две бинарных входных величины x и y, а в результате получается одна бинарная выходная величина z.

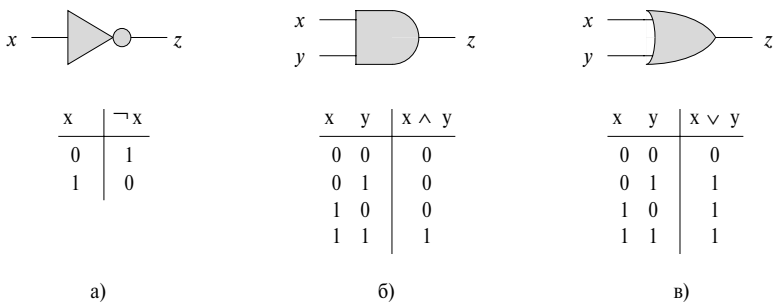


Рис. 34.7. Три основных логических вентиля с бинарными входами и выходами

Работу каждого логического вентиля и каждого комбинационного логического элемента можно описать ***таблицей истинности*** (truth table), представленной под каждым вентилем на рис. 34.7. В части *а* показан вентиль НЕ, в части *б* — вентиль И, а в части *в* — вентиль ИЛИ. В таблице истинности для всех возможных наборов входных величин перечисляются выходные значения данного комбинационного элемента. Например, из таблицы истинности вентиля ИЛИ видно, что если его входные значения $x = 0$ и $y = 1$, то выходное значение равно $z = 1$. Для обозначения функции НЕ используется символ $\neg$, для обозначения функции И — символ $\wedge$, а для обозначения функции ИЛИ — символ $\vee$. Например, $0 \vee 1 = 1$.

Вентили И и ИЛИ можно обобщить на случай, когда входных значений больше двух. Выходное значение вентиля И равно 1, если все его входные значения равны 1; в противном случае его выходное значение равно 0. Выходное значение вентиля ИЛИ равно 1, если хоть одно из его входных значений равно 1; в противном случае его выходное значение равно 0.

Булева комбинационная схема (boolean combinational circuit) состоит из одного или нескольких комбинационных логических элементов, соединенных ***проводами*** (wires). Провод может соединять выход одного элемента со входом другого, подавая таким образом выходное значение первого элемента на вход второго. На рис. 34.8 изображены две похожие друг на друга булевы комбинационные схемы, отличающиеся лишь одним вентилем. В части *а* рисунка также приводятся значения, которые передаются по каждому проводу, если на вход подаются значения $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$. Хотя к одному и тому же проводу нельзя подключить более одного вывода комбинационного элемента, с него может поступать сигнал одновременно на входы нескольких элементов. Количество элементов, для которых данный провод предоставляет входные данные, называется его ***коэффициентом ветвления*** (fan-out). Если к проводу не подсоединены выходы никаких элементов, он называется ***входом схемы*** (circuit input), получающем входные данные из внешних источников. Если к проводу не подсоединен вход ни одного элемента, он называется ***выходом схемы*** (circuit output), по которому выдаются результаты работы схемы. (Ответвление внутреннего провода также может выступать в роли выхода.) Чтобы определить задачу о выполнимости схемы, следует ограничиться рассмотрением схем с одним выходом, хотя на практике при разработке аппаратного оборудования встречаются логические комбинационные схемы с несколькими выводами.

Логическая комбинационная схема не содержит циклов. Поясним это нагляднее. Предположим, что схеме сопоставляется ориентированный граф $G = (V, E)$ с вершинами в каждом комбинационном элементе и k ориентированными ребрами для каждого провода, коэффициент разветвления которых равен k. Ориентированное ребро (u, v) в этом графе существует, если провод соединяет выход элемента u со входом элемента v. Полученный в результате такого построения граф должен быть ациклическим.

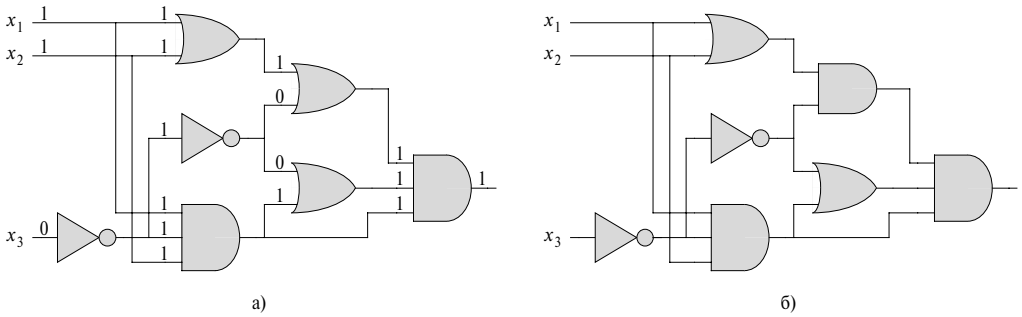


Рис. 34.8. Два экземпляра задачи о выполнимости схем

Будем рассматривать ***наборы значений*** булевых переменных, соответствующих входам схемы (truth assignment). Говорят, что логическая комбинационная схема ***выполнима*** (satisfiable), если она обладает ***выполняющим набором*** (satisfying assignment): набором значений, в результате подачи которого на вход схемы ее выходное значение равно 1. Например, для схемы, изображенной на рис. 34.8*а*, выполняющий набор имеет вид $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$; следовательно, эта схема выполнима. В упражнении 34.3-1 предлагается показать, что никакое присваивание значений величинам x_1, x_2 и x_3 не приведет к тому, что на выходе схемы, изображенной на рис. 34.8*б*, появится значение 1. Эта схема всегда выдает значение 0, поэтому она невыполнима.

Задача о выполнимости схемы (circuit-satisfiability problem) формулируется так: выполнима ли заданная логическая комбинационная схема, состоящая из вентилей И, ИЛИ и НЕ. Однако чтобы поставить этот вопрос формально, необходимо принять соглашение по поводу стандартного кода для таких схем. ***Размер*** (size) логической комбинационной схемы определяется как сумма количества логических комбинационных элементов и числа проводов в схеме. Можно предложить код, использующийся при представлении графов, который отображает каждую заданную схему C на бинарную строку $\langle C\rangle$, длина которой выражается не более чем полиномиальной функцией от размера схемы. Таким образом, можно определить формальный язык

$$\text{CIRCUIT\_SAT} = \{\langle C\rangle : C - \text{ выполнимая схема}\}.$$

Задача о выполнимости схем возникает при оптимизации компьютерного аппаратного обеспечения. Если какая-то подсхема рассматриваемой схемы всегда выдает значение 0, то ее можно заменить более простой подсхемой, в которой опускаются все логические вентили, а на выход подается постоянное значение 0. Было бы полезно иметь алгоритм решения этой задачи, время которого выражалось бы полиномиальной функцией.

Чтобы проверить, разрешима ли данная схема C, можно попытаться просто перебрать все возможные комбинации входных значений. К сожалению, если в схеме k входов, то количество таких комбинаций равно 2^k. Если размер схемы C выражается полиномиальной функцией от k, то проверка всех комбинаций занимает время $\Omega\left(2^k\right)$, а эта функция по скорости роста превосходит полиномиальную функцию[7]. Как уже упоминалось, существует веский аргумент в пользу того, что не существует алгоритмов с полиномиальным временем работы, способных решить задачу о выполнимости схем, так как эта задача является NP-полной. Разобьем доказательство NP-полноты этой задачи на две части, соответствующие двум частям определения NP-полноты.

Лемма 34.5. Задача о выполнимости схем принадлежит классу NP.

Доказательство. В этом доказательстве будет сформулирован алгоритм A с двумя входными параметрами и полиномиальным временем работы, способный проверять решение задачи CIRCUIT_SAT. В роли одного из входных параметров алгоритма A выступает логическая комбинационная схема C (точнее, ее стандартный код). Вторым входным параметром является сертификат, который представляет собой булевы значения на всех проводах схемы. (Существование сертификата меньшего объема предлагается показать в упражнении 34.3-4.)

Алгоритм A строится следующим образом. Для каждого логического вентиля схемы проверяется, что предоставляемое сертификатом значение на выходном проводе правильно вычисляется как функция значений на входных проводах. Далее, если на выход всей цепи подается значение 1, алгоритм тоже выдает значение 1, поскольку величины, поступившие на вход схемы C, являются выполняющим набором. В противном случае алгоритм выводит значение 0.

Для любой выполнимой схемы C, выступающей в роли входного параметра алгоритма A, существует сертификат, длина которого выражается полиномиальной функцией от размера схемы C, и который приводит к тому, что на выход алгоритма A подается значение 1. Для любой невыполнимой схемы, выступающей в роли входного параметра алгоритма A, никакой сертификат не может заставить этот алгоритм поверить в то, что эта схема выполнима. Алгоритм A выполняется в течение полиномиального времени: при хорошей реализации достаточно будет линейного времени. Таким образом, задачу CIRCUIT_SAT можно проверить за полиномиальное время, и CIRCUIT_SAT $\in$ NP. ■

[7] С другой стороны, если размер схемы C равен $\Theta\left(2^k\right)$, то алгоритм со временем работы $O\left(2^k\right)$ завершается по истечении времени, выражающегося полиномиальной функцией от размера схемы. Даже если P $\neq$ NP, эта ситуация не противоречит NP-полноте задачи; из наличия алгоритма с полиномиальным временем работы для особого случая не следует, что такой алгоритм существует во всех случаях.

Вторая часть доказательства NP-полноты задачи CIRCUIT_SAT заключается в том, чтобы показать, что ее язык является NP-сложным. Другими словами, необходимо показать, что каждый язык класса NP приводится в течение полиномиального времени к языку CIRCUIT_SAT. Само доказательство этого факта содержит много технических сложностей, поэтому здесь приводится набросок доказательства, основанный на некоторых принципах работы аппаратного обеспечения компьютера.

Компьютерная программа хранится в памяти компьютера в виде последовательности инструкций. Типичная инструкция содержит код операции, которую нужно выполнить, адреса операндов в памяти и адрес, куда следует поместить результат. Специальная ячейка памяти под названием ***счетчик команд*** (program counter, PC) следит за тем, какая инструкция должна выполняться следующей. При извлечении очередной инструкции показание счетчика команд автоматически увеличивается на единицу. Это приводит к последовательному выполнению инструкций компьютером. Однако в результате выполнения некоторых инструкций в счетчик команд может быть записано некоторое значение, что приводит к изменению обычного последовательного порядка выполнения. Таким образом организуются циклы и условные ветвления.

В любой момент выполнения программы состояние вычислений в целом можно представить содержимым памяти компьютера. (Имеется в виду область памяти, состоящая из самой программы, счетчика команд, рабочей области и всех других битов состояния, с помощью которых компьютер ведет учет выполнения программы.) Назовем любое отдельное состояние памяти компьютера ***конфигурацией*** (configuration). Выполнение команд можно рассматривать как отображение одной конфигурации на другую. Важно то, что аппаратное обеспечение, осуществляющее это отображение, можно реализовать в виде логической комбинационной схемы, которая при доказательстве приведенной ниже леммы будет обозначена через M.

Лемма 34.6. Задача о выполнимости схем является NP-сложной.

Доказательство. Пусть L — язык класса NP. Опишем алгоритм F с полиномиальным временем работы, вычисляющий функцию приведения f, которая отображает каждую бинарную строку x на схему $C = f(x)$, такую что $x \in L$ тогда и только тогда, когда $C \in$ CIRCUIT_SAT.

Поскольку $L \in$ NP, должен существовать алгоритм A, верифицирующий язык L в течение полиномиального времени. В алгоритме F, который будет построен ниже, для вычисления функции приведения f будет использован алгоритм A с двумя входными параметрами.

Пусть $T(n)$ — время работы алгоритма A в наихудшем случае для входной строки длиной n, а $k \geqslant 1$ — константа, такая что $T(n) = O\left(n^k\right)$, и длина

сертификата равна $O\left(n^k\right)$. (В действительности время работы алгоритма A выражается полиномиальной функцией от полного объема входных данных, в состав которых входит и входная строка, и сертификат, но так как длина сертификата полиномиальным образом зависит от длины n входной строки, время работы алгоритма полиномиально зависит от n.)

Основная идея доказательства заключается в том, чтобы представить выполнение алгоритма A в виде последовательности конфигураций. Как видно из рис. 34.9, каждую конфигурацию можно разбить на следующие части: программа алгоритма A, счетчик команд PC, регистры процессора, входные данные x, сертификат y и рабочая память. Начиная с исходной конфигурации c_0, каждая конфигурация c_i с помощью комбинационной схемы M, аппаратно реализованной в компьютере, отображается на последующую конфигурацию c_{i+1}. Выходное значение алгоритма A (0 или 1) по завершении выполнения алгоритма A записывается в некоторую специально предназначенную для этого ячейку рабочего пространства. При этом предполагается, что после останова алгоритма A это значение не изменяется. Таким образом, если выполнение алгоритма состоит не более чем из $T(n)$ шагов, результат его работы хранится в определенном бите в $c_{T(n)}$.

Алгоритм приведения F конструирует единую комбинационную схему, вычисляющую все конфигурации, которые получаются из заданной входной конфигурации. Идея заключается в том, чтобы "склеить" вместе все $T(n)$ копий схемы M. Выходные данные i-й схемы, выдающей конфигурацию c_i, подаются непосредственно на вход $(i+1)$-й схемы. Таким образом, эти конфигурации вместо того, чтобы в конце своей работы заносить выходное значение в один из регистров состояния, просто передают его по проводам, соединяющим копии схемы M.

Теперь вспомним, что должен делать алгоритм приведения F, время работы которого выражается полиномиальной функцией. Для заданных входных данных x он должен вычислить схему $C = f(x)$, которая была бы выполнима тогда и только тогда, когда существует сертификат y, удовлетворяющий равенству $A(x, y) = 1$. Когда алгоритм F получает входное значение x, он сначала вычисляет $n = |x|$ и конструирует комбинационную схему C', состоящую из $T(n)$ копий схемы M. На вход схемы C' подается начальная конфигурация, соответствующая вычислению $A(x, y)$, а выходом этой схемы является конфигурация $c_{T(n)}$.

Схема $C = f(x)$, которую создает алгоритм F, получается путем небольшой модификации схемы C'. Во-первых, входы схемы C', соответствующие программе алгоритма A, начальному значению счетчика команд, входной величине x и начальному состоянию памяти, соединяются проводами непосредственно с этими известными величинами. Таким образом, оставшиеся входы схемы соответствуют сертификату y. Во-вторых, игнорируются все выходы схемы за исключением одного бита конфигурации $c_{T(n)}$, соответствующего выходному значению алгоритма A. Сконструированная таким образом схема C для любого входного параметра y

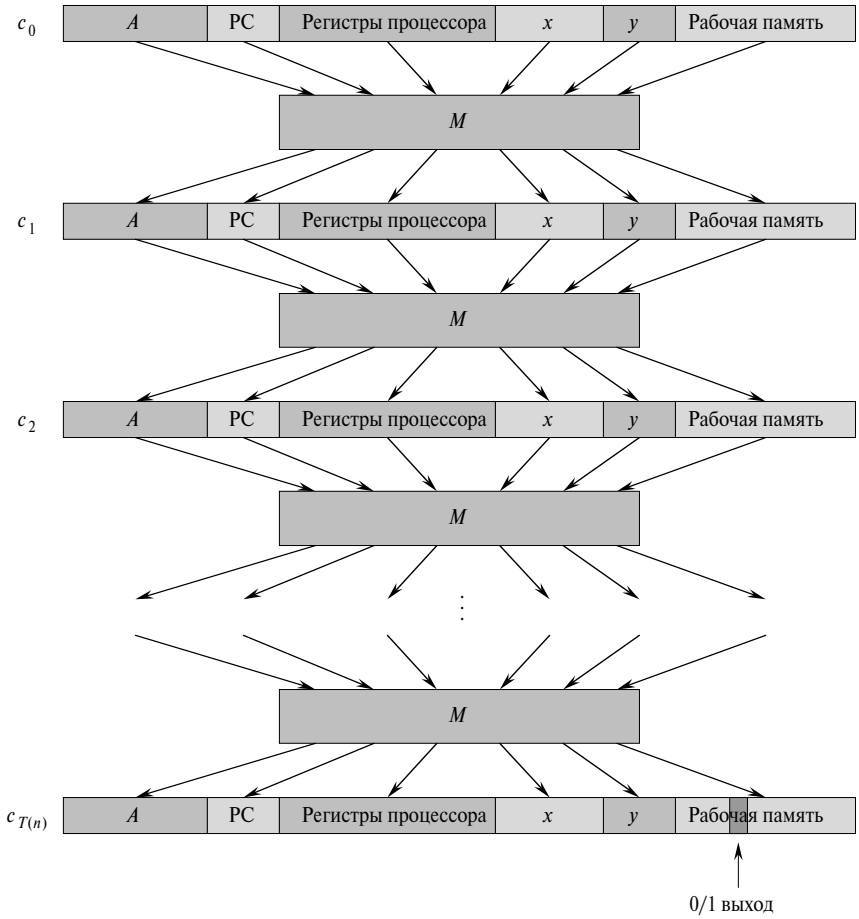


Рис. 34.9. Последовательность конфигураций, полученных в ходе работы алгоритма A, на вход которого поступила строка x и сертификат y

длиной $O\left(n^k\right)$ вычисляет величину $C\left(y\right) = A\left(x, y\right)$. Алгоритм приведения F, на вход которого подается строка x, вычисляет описанную выше схему C и выдает ее.

Осталось доказать два свойства. Во-первых, необходимо показать, что алгоритм F правильно вычисляет функцию приведения f. Другими словами, необходимо показать, что схема C выполнима тогда и только тогда, когда существует сертификат y, такой что $A\left(x, y\right) = 1$. Во-вторых, необходимо показать, что работа алгоритма F завершается по истечении полиномиального времени.

Чтобы показать, что алгоритм F корректно вычисляет функцию приведения, предположим, что существует сертификат y длиной $O\left(n^k\right)$, такой что $A\left(x, y\right) = 1$. Тогда при подаче битов сертификата y на вход схемы C на выходе этой схемы получим значение $C\left(y\right) = A\left(x, y\right) = 1$. Таким образом, если сертификат

существует, то схема C выполнима. С другой стороны, предположим, что схема C выполнима. Тогда существует такое входное значение y, что $C(y) = 1$, откуда можно заключить, что $A(x, y) = 1$. Итак, алгоритм F корректно вычисляет функцию приведения.

Чтобы завершить набросок доказательства, осталось лишь показать, что время работы алгоритма F выражается полиномиальной функцией от $n = |x|$. Первое наблюдение, которое можно сделать, заключается в том, что количество битов, необходимое для представления конфигурации, полиномиально зависит от n. Объем программы самого алгоритма A фиксирован и не зависит от длины его входного параметра x. Длина входного параметра x равна n, а длина сертификата y — $O\left(n^k\right)$. Поскольку работа алгоритма состоит не более чем из $O\left(n^k\right)$ шагов, объем необходимой ему рабочей памяти также выражается полиномиальной функцией от n. (Предполагается, что эта область памяти непрерывна; в упражнении 34.3-5 предлагается обобщить доказательство для случая, когда ячейки, к которым обращается алгоритм A, разбросаны по памяти большего объема, причем разброс ячеек имеет свой вид для каждого входного параметра x.)

Размер комбинационной схемы M, реализованной аппаратной частью компьютера, выражается полиномиальной функцией от длины конфигурации, которая в свою очередь является полиномиальной от величины $O\left(n^k\right)$ и, следовательно, полиномиально зависит от n. (В большей части такой схемы реализуется логика системы памяти.) Схема C состоит не более чем из $t = O\left(n^k\right)$ копий M, поэтому ее размер полиномиально зависит от n. Схему C для входного параметра x можно составить в течение полиномиального времени при помощи алгоритма приведения F, поскольку каждый этап построения длится в течение полиномиального времени. ■

Таким образом, язык CIRCUIT_SAT не проще любого языка класса NP, а поскольку он относится к классу NP, он является NP-полным.

Теорема 34.7. Задача о выполнимости схем является NP-полной.

Доказательство. Справедливость теоремы непосредственно следует из лемм 34.5 и 34.6, а также из определения NP-полноты. ■

Упражнения

34.3-1. Убедитесь, что схема, изображенная на рис. 34.8*б*, невыполнима.

34.3-2. Покажите, что отношение $\leqslant_{\mathrm{P}}$ является транзитивным в отношении языков; другими словами, покажите, что из $L_1 \leqslant_{\mathrm{P}} L_2$ и $L_2 \leqslant_{\mathrm{P}} L_3$ следует $L_1 \leqslant_{\mathrm{P}} L_3$.

34.3-3. Докажите, что $L \leqslant_{\mathrm{P}} \bar{L}$ тогда и только тогда, когда $\bar{L} \leqslant_{\mathrm{P}} L$.

34.3-4. Покажите, что в качестве сертификата в альтернативном доказательстве леммы 34.5 можно использовать выполняющий набор. С каким сертификатом легче провести доказательство?

34.3-5. В доказательстве леммы 34.6 предполагается, что рабочая память алгоритма A занимает непрерывную область полиномиального объема. В каком месте доказательства используется это предположение? Покажите, что оно не приводит к потере общности.

34.3-6. Язык L называется ***полным*** (complete) в классе языков C относительно приведения в течение полиномиального времени, если $L \in C$ и $L' \leqslant_{\mathrm{P}} L$ для всех $L' \in C$. Покажите, что множества $\emptyset$ и $\{0,1\}^*$ — единственные языки класса P, которые не являются полными в этом классе относительно приведения за полиномиальное время.

34.3-7. Покажите, что язык L полный для класса NP тогда и только тогда, когда язык $\bar{L}$ полный для класса co-NP.

34.3-8. Алгоритм приведения F в доказательстве леммы 34.6 строит схему $C = f(x)$ на основе знаний о x, A и k. Профессор заметил, что алгоритм F получает в качестве аргумента только x. В то же время об алгоритме A и константе k, с помощью которой выражается его время работы $O\left(n^k\right)$, известно лишь то, что они существуют (поскольку язык L принадлежит к классу NP), но не сами их значения. Из этого профессор делает вывод, что алгоритм F может не построить схему C и что язык CIRCUIT_SAT не обязательно NP-сложный. Объясните, какая ошибка содержится в рассуждениях профессора.

34.4 Доказательство NP-полноты

NP-полнота для задачи о выполнимости схем основана на непосредственном доказательстве соотношения $L \leqslant_{\mathrm{P}}$ CIRCUIT_SAT для каждого языка $L \in \mathrm{NP}$. В этом разделе будет показано, как доказать NP-полноту языка без непосредственного приведения *каждого* языка из класса NP к заданному языку. Проиллюстрируем эту методику, доказав NP-полноту различных задач на выполнимость формул. Намного большее количество примеров применения этой методики содержится в разделе 34.5.

Основой рассматриваемого в этом разделе метода, позволяющего доказать NP-полноту задачи, служит сформулированная ниже лемма.

Лемма 34.8. Если язык L такой, что $L' \leqslant_{\mathrm{P}} L$ для некоторого языка $L' \in \mathrm{NPC}$, то язык L — NP-сложный. Более того, если $L \in \mathrm{NP}$, то $L \in \mathrm{NPC}$.

Доказательство. Поскольку язык L' NP-полный, для всех $L'' \in \mathrm{NP}$ выполняется соотношение $L'' \leqslant_{\mathrm{P}} L'$. Согласно условию леммы $L' \leqslant_{\mathrm{P}} L$, поэтому в силу транзитивности (упражнение 34.3-2) мы имеем $L'' \leqslant_{\mathrm{P}} L$, что и показывает, что язык L — NP-сложный. Если $L \in \mathrm{NP}$, то мы также имеем $L \in \mathrm{NPC}$. ■

Другими словами, если язык L', о котором известно, что он — NP-полный, удается свести к языку L, тем самым к этому языку неявно сводится любой язык класса NP. Таким образом, лемма 34.8 позволяет сформулировать метод доказательства NP-полноты языка L, состоящий из перечисленных ниже этапов.

1. Доказывается, что $L \in \mathrm{NP}$.
2. Выбирается язык L', для которого известно, что он — NP-полный.
3. Описывается алгоритм, который вычисляет функцию f, отображающую каждый экземпляр $x \in \{0,1\}^*$ языка L' на экземпляр $f(x)$ языка L.
4. Доказывается, что для функции f соотношение $x \in L'$ выполняется тогда и только тогда, когда $f(x) \in L$ для всех $x \in \{0,1\}^*$.
5. Доказывается, что время работы алгоритма, вычисляющего функцию f, выражается полиномиальной функцией.

(Выполнение этапов 2-5 доказывает, что язык L NP-сложный.) Такая методология приведения с помощью одного языка, для которого известно, что он NP-полный, намного проще, чем процесс, когда непосредственно демонстрируется, как выполнить приведение каждого языка из класса NP. Доказательство соотношения CIRCUIT_SAT $\in \mathrm{NPC}$ — первый важный шаг в этом направлении. Знание того факта, что задача о выполнимости схемы является NP-полной, позволяет доказывать NP-полноту других задач намного проще. Более того, по мере расширения списка известных NP-полных задач у нас появляется все больше возможностей для выбора языков, которые будут использоваться для приведения.

Выполнимость формулы

Чтобы проиллюстрировать методику приведения, докажем NP-полноту задачи, состоящей в определении того, выполнима ли не схема, а формула. Именно для этой задачи впервые была доказана NP-полнота.

Сформулируем задачу о ***выполнимости формулы*** (formula satisfiability) в терминах языка SAT. Экземпляр языка SAT — это булева формула ϕ, состоящая из перечисленных ниже элементов.

1. n булевых переменных: $x_1, x_2, \ldots, x_n$.
2. m булевых соединяющих элементов, в роли которых выступает произвольная логическая функция с одним или двумя входными значениями и одним выходным значением, например, $\wedge$ (И), $\vee$ (ИЛИ), $\neg$ (НЕ), $\rightarrow$ (импликация), $\leftrightarrow$ (эквивалентность).

3. Скобки. (Без потери общности предполагается, что лишние скобки не употребляются, т.е. что на каждый логический соединяющий элемент приходится не более одной пары скобок.)

Логическую формулу ϕ легко закодировать строкой, длина которой выражается полиномиальной функцией от $n+m$. Как и для логических комбинационных схем, ***набором значений*** (truth assignment) логической формулы ϕ называется множество значений переменных этой формулы, а ***выполняющим набором*** (satisfying assignment) — такой набор значений, при котором результат вычисления формулы равен 1. Формула, для которой существует выполняющий набор, является ***выполнимой*** (satisfiable). В задаче о выполнимости спрашивается, выполнима ли данная формула. В терминах формальных языков эта задача имеет вид

$$\textsc{Sat} = \{\langle\phi\rangle : \phi - \text{ выполнимая булева формула}\}.$$

В качестве примера приведем формулу

$$\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2,$$

для которой существует выполняющий набор $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$, поскольку

$$\begin{aligned}\phi &= ((0 \to 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 = \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 = \\ &= (1 \vee 0) \wedge 1 = \\ &= 1.\end{aligned} \tag{34.2}$$

Таким образом, формула ϕ принадлежит языку SAT.

Прямолинейный алгоритм, позволяющий определить, выполнима ли произвольная булева формула, не укладывается в полиномиально-временные рамки. В формуле ϕ с n переменными всего имеется 2^n возможных вариантов присваивания. Если длина $\langle\phi\rangle$ выражается полиномиальной функцией от n, то для проверки каждого варианта присваивания потребуется время $\Omega(2^n)$, т.е. оно выражается функцией, показатель роста которой превосходит полиномиальную функцию от длины $\langle\phi\rangle$. Как видно из приведенной ниже теоремы, существование алгоритма с полиномиальным временем маловероятно.

Теорема 34.9. Задача о выполнимости булевых формул является NP-полной.

Доказательство. Сначала докажем, что $\textsc{Sat} \in \mathrm{NP}$. Затем покажем, что язык SAT NP-сложный, для чего продемонстрируем справедливость соотношения $\textsc{Circuit\_Sat} \leqslant_{\mathrm{P}} \textsc{Sat}$. Согласно лемме 34.8, этого достаточно для доказательства теоремы.

Чтобы показать, что язык SAT относится к классу NP, покажем, что сертификат, состоящий из выполняющего набора входной формулы ϕ, можно верифицировать

в течение полиномиального времени. В алгоритме верификации каждая содержащаяся в формуле переменная просто заменяется соответствующим значением, после чего вычисляется выражение, как это было проделано с уравнением (34.2). Эту задачу легко выполнить в течение полиномиального времени. Если в результате получится значение 1, то формула выполнима. Таким образом, первое условие леммы 34.8 выполняется.

Чтобы доказать, что язык SAT NP-сложный, покажем, что справедливо соотношение CIRCUIT_SAT $\leqslant_{\mathrm{P}}$ SAT. Другими словами, любой экземпляр задачи о выполнимости схемы можно в течение полиномиального времени свести к экземпляру задачи о выполнимости формулы. С помощью индукции любую логическую комбинационную схему можно выразить в виде булевой формулы. Для этого достаточно рассмотреть вентиль, который выдает выходное значение схемы, и по индукции выразить каждое входное значение этого вентиля в виде формул. Формула схемы получается путем выписывания выражения, в котором функции вентиля применяются к формулам входов.

К сожалению, такой незамысловатый метод не может стать основой приведения в течение полиномиального времени. Как предлагается показать в упражнении 34.4-1, общие вспомогательные формулы, соответствующие вентилям, коэффициент ветвления которых равен 2 или превышает это значение, могут привести к экспоненциальному росту размера формулы. Таким образом, алгоритм приведения должен быть несколько остроумнее.

Основная идея приведения задачи CIRCUIT_SAT к задаче SAT для схемы из рис. 34.8*а* проиллюстрирована на рис. 34.10. Каждому проводу x_i схемы C сопоставляется одноименная переменная формулы ϕ. Тогда надлежащее действие вентиля можно выразить в виде формулы, включающей в себя переменные, которые соответствуют проводам, подсоединенным к этому вентилю. Например, действие вентиля И выражается формулой $x_{10} \leftrightarrow (x_7 \land x_8 \land x_9)$.

Формула ϕ, которая является результатом выполнения алгоритма приведения, получается путем конъюнкции (вентиль И) выходной переменной схемы с выражением, представляющим собой конъюнкцию взятых в скобки выражений, описывающих действие каждого вентиля. Для схемы, изображенной на рисунке, формула имеет такой вид:

$$\begin{aligned}\phi = x_{10} \land (x_4 &\leftrightarrow \neg x_3) \land \\ &\land (x_5 \leftrightarrow (x_1 \lor x_2)) \land \\ &\land (x_6 \leftrightarrow \neg x_4) \land \\ &\land (x_7 \leftrightarrow (x_1 \land x_2 \land x_4)) \land \\ &\land (x_8 \leftrightarrow (x_5 \lor x_6)) \land \\ &\land (x_9 \leftrightarrow (x_6 \lor x_7)) \land \\ &\land (x_{10} \leftrightarrow (x_7 \land x_8 \land x_9))\,.\end{aligned}$$

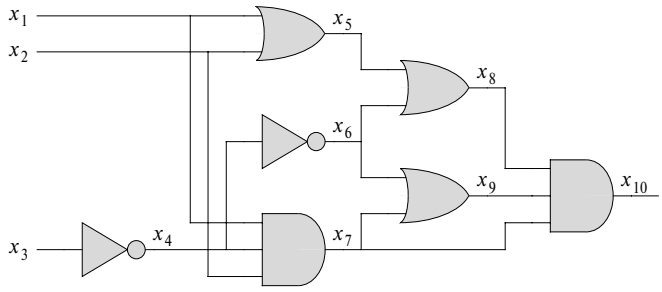


Рис. 34.10. Приведение задачи о выполнимости схем к задаче о выполнимости формул; формула, полученная в результате выполнения алгоритма приведения, содержит переменные, соответствующие каждому проводу схемы

Для заданной схемы C легко получить такую формулу ϕ в течение полиномиального времени.

Почему схема C выполнима именно тогда, когда выполнима формула ϕ? Если у схемы C имеется выполняющий набор, то по каждому проводу схемы передается вполне определенное значение, и на выходе схемы получается значение 1. Поэтому набор значений, передающихся по проводам схемы, в формуле ϕ приведет к тому, что значение каждого выражения в скобках в формуле ϕ будет равно 1, вследствие чего конъюнкция всех этих выражений равна 1. Верно и обратное — если существует набор данных, для которого значение формулы ϕ равно 1, то схема C выполнима (это можно показать аналогично). Таким образом, показана справедливость соотношения CIRCUIT_SAT $\leqslant_P$ SAT, что завершает доказательство теоремы. ■

3-CNF выполнимость

NP-полноту многих задач можно доказать путем приведения к ним задачи о выполнимости формул. Однако алгоритм приведения должен работать с любыми входными формулами, что может привести к огромному количеству случаев, подлежащих рассмотрению. Поэтому часто желательно выполнять приведение с помощью упрощенной версии языка булевых формул, чтобы уменьшить количество рассматриваемых случаев (конечно же, ограничивать язык настолько, чтобы он стал распознаваем в течение полиномиального времени, при этом нельзя). Одним из таких удобных языков оказывается язык формул в 3-конъюктивной нормальной форме 3-CNF_SAT.

Определим задачу о 3-CNF выполнимости в описанных ниже терминах. ***Литерал*** (literal) в булевой формуле — это входящая в нее переменная или отрицание этой переменной. Булева формула приведена к ***конъюнктивной нормальной фор-***

ме (Conjunctive Normal Form, CNF), если она имеет вид конъюнкции (вентиль И) ***выражений в скобках*** (clauses), каждое из которых представляет собой дизъюнкцию (вентиль ИЛИ) одного или нескольких литералов. Булева формула выражена в ***3-конъюнктивной нормальной форме*** (3-conjunctive normal form), или ***3-CNF***, если в каждых скобках содержится ровно три различных литерала.

Например, булева формула

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

принадлежит классу 3-CNF. Первое из трех выражений в скобках имеет вид $(x_1 \vee \neg x_1 \vee \neg x_2)$ и содержит три литерала: $x_1, \neg x_1, \neg x_2$.

В задаче 3-CNF_SAT спрашивается, выполнима ли заданная формула ϕ, принадлежащая классу 3-CNF. В приведенной ниже теореме показано, что алгоритм с полиномиальным временем работы, способный определять выполнимость даже таких простых булевых формул, скорее всего, не существует.

Теорема 34.10. Задача о выполнимости булевых формул в 3-конъюнктивной нормальной форме является NP-полной.

Доказательство. Рассуждения, которые использовались в теореме 34.9 для доказательства того, что SAT $\in$ NP, применимы и для доказательства того, что 3-CNF_SAT $\in$ NP. Таким образом, согласно лемме 34.8, нам требуется лишь показать, что SAT $\leqslant_P$ 3-CNF_SAT.

Алгоритм приведения можно разбить на три основных этапа. На каждом этапе входная формула ϕ последовательно приводится к 3-конъюнктивной нормальной форме.

Первый этап аналогичен тому, который был использован при доказательстве соотношения SAT $\leqslant_P$ 3-CNF_SAT в теореме 34.9. Сначала для входной формулы ϕ конструируется бинарное "синтаксическое" дерево, листья которого соответствуют литералам, а узлы — соединительным элементам. На рис. 34.11 показано такое синтаксическое дерево для формулы

$$\phi = ((x_1 \rightarrow x_2) \vee \neg ((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2. \tag{34.3}$$

Если входная формула содержит выражение в скобках, представляющее собой дизъюнкцию или конъюнкцию нескольких литералов, то, воспользовавшись свойством ассоциативности, можно провести расстановку скобок в выражении таким образом, чтобы каждый внутренний узел полученного в результате дерева содержал 1 или 2 дочерних узла. Теперь такое бинарное синтаксическое дерево можно рассматривать как схему, предназначенную для вычисления функции.

По аналогии с процедурой приведения в теореме 34.9, введем переменную y_i, соответствующую выходному значению каждого внутреннего узла. Затем перепишем исходную формулу ϕ как конъюнкцию переменной, соответствующей корню

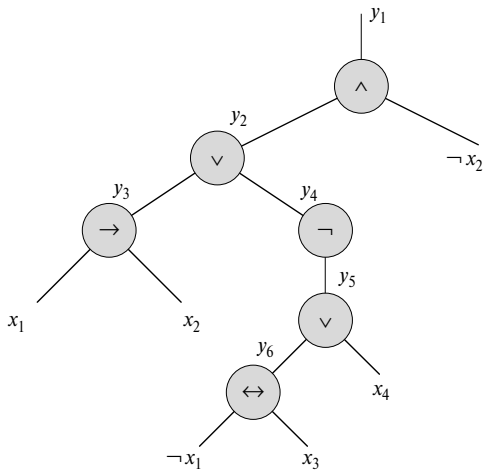


Рис. 34.11. Дерево, соответствующее формуле $\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

дерева, и выражений, описывающих операции в каждом узле. Для формулы (34.3) получившееся в результате выражение имеет следующий вид:

$$\begin{aligned}\phi' = y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \wedge \\ \wedge (y_2 \leftarrow (y_3 \vee y_4)) \wedge \\ \wedge (y_3 \leftrightarrow (x_1 \to x_2)) \wedge \\ \wedge (y_4 \leftrightarrow \neg y_5) \wedge \\ \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \wedge \\ \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\,.\end{aligned}$$

Заметим, что полученная таким образом формула ϕ' представляет собой конъюнкцию выражений в скобках ϕ'_i, каждое из которых содержит не более трех литералов. Единственное дополнительное требование состоит в том, что каждое выражение в скобках должно быть логической суммой литералов.

На втором этапе приведения каждое выражение в скобках ϕ'_i преобразуется к конъюнктивной нормальной форме. Составим таблицу истинности формулы ϕ'_i путем ее вычисления при всех возможных наборах значений ее переменных. Каждая строка такой таблицы соответствует одному из вариантов набора, и содержит сам вариант и результат вычисления исследуемого выражения. С помощью элементов таблицы истинности, которые приводят к нулевому результату в формуле, можно составить формулу, эквивалентную выражению $\neg\phi'_i$, в ***дизъюнктивной нормальной форме*** (Disjunctive Normal Form, DNF), которая является дизъюнкцией конъюнкций. Затем эта формула с помощью законов де Моргана (DeMorgan)

(уравнение (Б.2)) преобразуется в CNF-формулу ϕ_i'' путем замены всех литералов их дополнениями и замены операций дизъюнкций и конъюнкций друг на друга.

В качестве примера преобразуем выражение $\phi_1' = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ в CNF. Таблица истинности для этой формулы представлена в табл. 34.1. DNF-формула, эквивалентная выражению $\neg\phi_1'$, имеет вид

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)\,.$$

Применяя законы де Моргана, получим CNF-формулу

$$\phi_1'' = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)\,,$$

которая эквивалентна исходной формуле ϕ_1'.

Таблица 34.1. Таблица истинности для выражения $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

После того как каждое подвыражение ϕ_i', содержащееся в формуле ϕ', оказывается преобразовано в CNF-формулу ϕ_i'', так что ϕ' эквивалентно CNF-формуле ϕ'', состоящей из конъюнкции выражений ϕ_i''. Кроме того, каждое подвыражение формулы ϕ'' содержит не более трех литералов.

На третьем и последнем этапе приведения формула преобразуется таким образом, чтобы в каждых скобках содержалось *точно* 3 различных литерала. Полученная в результате 3-CNF формула ϕ''' составлена из подвыражений, входящих в CNF-формулу ϕ''. В ней также используются две вспомогательные переменные, которые будут обозначаться как p и q. Для каждого подвыражения C_i из формулы ϕ'' в формулу ϕ''' включаются следующие подвыражения.

- Если C_i содержит 3 различных литерала, то это подвыражение включается в формулу ϕ''' в неизменном виде.
- Если выражение C_i содержит 2 различных литерала, т.е. если $C_i = (l_1 \vee l_2)$, где l_1 и l_2 — литералы, то в качестве подвыражения в формулу ϕ''' включается выражение $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$. Литералы p и $\neg p$ служат лишь

для того, чтобы удовлетворялось требование о наличии в каждом подвыражении в скобках ровно трех разных литералов: выражение $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ эквивалентно выражению $(l_1 \vee l_2)$ и при $p = 0$, и при $p = 1$.

- Если выражение C_i содержит ровно 1 литерал l, то вместо него в качестве подвыражения в формулу ϕ''' включается выражение $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$. При любых значениях переменных p и q приведенная конъюнкция четырех подвыражений в скобках эквивалентна l.

Проверив каждый из описанных выше трех этапов, можно убедиться, что 3-CNF формула ϕ''' выполнима тогда и только тогда, когда выполнима формула ϕ. Как и в случае приведения задачи CIRCUIT_SAT к задаче SAT, составление на первом этапе формулы ϕ' по формуле ϕ не повлияет на выполнимость. На втором этапе конструируется CNF-формула ϕ'', алгебраически эквивалентная формуле ϕ'. На третьем этапе конструируется 3-CNF формула ϕ''', результат которой эквивалентен формуле ϕ'', поскольку присваивание любых значений переменным p и q приводит к формуле, алгебраически эквивалентной формуле ϕ''.

Необходимо также показать, что приведение можно выполнить в течение полиномиального времени. В ходе конструирования формулы ϕ' по формуле ϕ приходится добавлять не более одной переменной и одной пары скобок на каждый соединительный элемент формулы ϕ. В процессе построения формулы ϕ'' по формуле ϕ' в формулу ϕ'' добавится не более восьми подвыражений в скобках на каждое подвыражение в скобках из формулы ϕ', поскольку каждое подвыражение в этой формуле содержит не более трех переменных и таблица его истинности состоит не более чем из $2^3 = 8$ строк. В ходе конструирования формулы ϕ''' по формуле ϕ'' в формулу ϕ''' добавляется не более четырех подвыражений на каждое подвыражение формулы ϕ''. Таким образом, размер полученной в результате формулы ϕ''' выражается полиномиальной функцией от длины исходной формулы. Следовательно, каждый этап можно легко выполнить в течение полиномиального времени. ■

Упражнения

34.4-1. Рассмотрите приведение “в лоб” (время работы которого не выражается полиномиальной функцией) из теоремы 34.9. Опишите схему размера n, размер которой после преобразования этим методом превратится в формулу, размер которой выражается показательной функцией от n.

34.4-2. Приведите 3-CNF формулу, которая получится в результате применения метода, описанного в теореме 34.10, к формуле (34.3).

34.4-3. Профессор предложил показать, что SAT $\leqslant_P$ 3-CNF_SAT, лишь с помощью метода с использованием таблицы истинности, описанного в до-

казательстве теоремы 34.10, исключая при этом другие этапы. Другими словами, профессор предложил взять булеву формулу ϕ, составить таблицу истинности для ее переменных, получить из нее формулу в 3-DNF, эквивалентную $\neg\phi$, после чего составить логическое отрицание полученной формулы и с помощью законов де Моргана получить 3-CNF формулу, эквивалентную формуле ϕ. Покажите, что в результате применения такой стратегии не удается выполнить приведение формулы в течение полиномиального времени.

34.4-4. Покажите, что задача определения того, является ли тавтологией данная формула, является полной в классе co-NP. (*Указание*: см. упражнение 34.3-7).

34.4-5. Покажите, что задача по определению выполнимости булевых формул в дизъюнктивной нормальной форме разрешима в течение полиномиального времени.

34.4-6. Предположим, кто-то разработал алгоритм с полиномиальным временем выполнения, позволяющий решить задачу о выполнимости формул. Опишите, как с помощью этого алгоритма в течение полиномиального времени находить выполняющие наборы.

34.4-7. Пусть 2-CNF_SAT — множество выполнимых булевых формул в CNF, у которых каждое подвыражение в скобках содержит ровно по два литерала. Покажите, что 2-CNF_SAT $\in$ P. Постарайтесь, чтобы ваш алгоритм был как можно более эффективен. (*Указание*: воспользуйтесь тем, что выражение $x \vee y$ эквивалентно выражению $\neg x \rightarrow y$. Приведите задачу 2-CNF_SAT к задаче на ориентированном графе, имеющей эффективное решение.)

34.5 NP-полные задачи

NP-полные задачи возникают в различных областях: в булевой логике, в теории графов, в арифметике, при разработке сетей, в теории множеств и разбиений, при хранении и поиске информации, при планировании вычислительных процессов, в математическом программировании, в алгебре и теории чисел, при создании игр и головоломок, в теории автоматов и языков, при оптимизации программ, в биологии, в химии, физике и т.п. В настоящем разделе с помощью методики приведения будет доказана NP-полнота различных задач, возникающих в теории графов и при разбиении множеств.

На рис. 34.12 приводится схема доказательства NP-полноты, которая используется в этом разделе и в разделе 34.4. Для каждого из представленных на рисунке языков NP-полнота доказывается путем приведения его к языку, на который

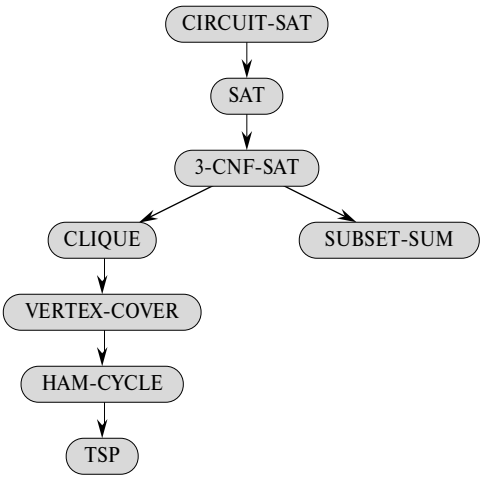


Рис. 34.12. Схема доказательств NP-полноты для задач, рассматриваемых в разделах 34.4 и 34.5; все доказательства в конечном итоге сводятся к приведению NP-полной задачи CIRCUIT_SAT к рассматриваемой

указывает идущая от этого языка стрелка. В качестве корневого выступает язык CIRCUIT_SAT, NP-полнота которого доказывается в теореме 34.7.

34.5.1 Задача о клике

Клика (clique) неориентированного графа $G = (V, E)$ — это подмножество $V' \subseteq V$ вершин, каждая пара в котором связана ребром из множества E. Другими словами, клика — это полный подграф графа G. ***Размер*** (size) клики — это количество содержащихся в нем вершин. ***Задача о клике*** (clique problem) — это задача оптимизации, в которой требуется найти клику максимального размера, содержащуюся в заданном графе. В соответствующей задаче принятия решения спрашивается, содержится ли в графе клика заданного размера k. Формальное определение клики имеет вид

$$\text{CLIQUE} = \{\langle G, k\rangle : G \text{ — граф с кликой размера } k\}\,.$$

Простейший алгоритм определения того, содержит ли граф $G = (V, E)$ с $|V|$ вершинами клику размера k, заключается в том, чтобы перечислить все k-элементные подмножества множества V, и проверить, образует ли каждое из них клику. Время работы этого алгоритма равно $\Omega\left(k^2\binom{|V|}{k}\right)$ и является полиномиальным, если k — константа. Однако в общем случае величина k может достигать значения, близкого к $|V|/2$, и в этом случае время работы алгоритма превышает

полиномиальное. Есть основания предполагать, что эффективного алгоритма для задачи о клике не существует.

Теорема 34.11. Задача о клике является NP-полной.

Доказательство. Чтобы показать, что CLIQUE$\in$NP для заданного графа $G = (V, E)$, воспользуемся множеством $V' \subseteq V$ вершин клики в качестве сертификата для данного графа. Проверить, является ли множество вершин V' кликой, можно в течение полиномиального времени, проверяя для каждой пары вершин $u, v \in V'$, принадлежит ли соединяющее их ребро (u, v) множеству E.

Теперь докажем, что 3-CNF_SAT $\leqslant_{\text{P}}$ CLIQUE, откуда следует, что задача о клике является NP-сложной. То, что этот результат удается доказать, может показаться удивительным, потому что на первый взгляд логические формулы мало напоминают графы.

Построение алгоритма приведения начинается с экземпляра задачи 3-CNF_SAT. Пусть $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ — булева формула из класса 3-CNF с k подвыражениями в скобках. Каждое подвыражение C_r при $r = 1, 2, \ldots, k$ содержит ровно три различных литерала l_1^r, l_2^r, и l_3^r. Сконструируем такой граф G, чтобы формула ϕ была выполнима тогда и только тогда, когда граф G содержит клику размера k.

Это делается так. Для каждого подвыражения $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ в формуле ϕ в множество V добавляется тройка вершин v_1^r, v_2^r и v_3^r. Вершины v_i^r и v_j^s соединяются ребром, если справедливы оба сформулированных ниже утверждения:

- вершины v_i^r и v_j^s принадлежат разным тройкам, т.е. $r \neq s$;
- литералы, соответствующие этим вершинам, ***совместимы*** (consistent), т.е. l_i^r не является логическим отрицанием l_j^s.

Такой граф легко построить на основе формулы ϕ в течение полиномиального времени. В качестве примера на рис. 34.13 приведен граф G, соответствующий формуле

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)\,.$$

Необходимо показать, что это преобразование формулы ϕ в граф G является приведением. Во-первых, предположим, что для формулы ϕ имеется выполняющий набор. Тогда каждое выражение C_r содержит не менее одного литерала l_i^r, значение которого равно 1, и каждый такой литерал соответствует вершине v_i^r. В результате извлечения такого “истинного” литерала из каждого подвыражения получается множество V', состоящее из k вершин. Мы утверждаем, что V' — клика. Для любых двух вершин $v_i^r, v_j^s \in V'$, где $r \neq s$, оба соответствующих литерала l_i^r и l_j^s при данном выполняющем наборе равны 1, поэтому они не могут быть отрицаниями друг друга. Таким образом, в соответствии со способом построения графа G, ребро $\left(v_i^r, v_j^s\right)$ принадлежит множеству E.

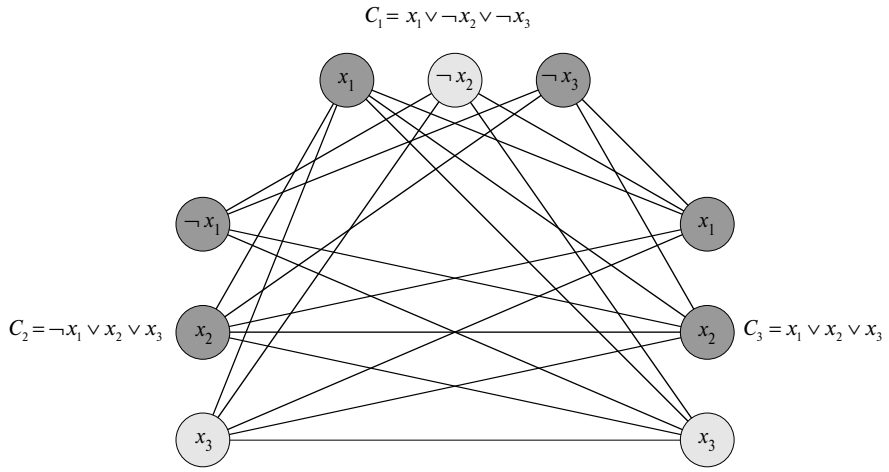


Рис. 34.13. Граф G, полученный в процессе приведения задачи 3-CNF_SAT к задаче CLIQUE

Проведем обратные рассуждения. Предположим, что граф G содержит клику V' размера k. Ни одно из ее ребер не соединяет вершины одной и той же тройки, поэтому клика V' содержит ровно по одной вершине каждой тройки. Каждому литералу l_i^r, такому что $v_i^r \in V'$, можно присвоить значение 1, не опасаясь того, что это значение будет присвоено как литералу, так и его дополнению, поскольку граф G не содержит ребер, соединяющих противоречивые литералы. Каждое подвыражение в скобках является выполнимым, поэтому формула ϕ тоже выполнима. (Переменным, которым не соответствует ни одна вершина клики, можно присваивать произвольные значения.) ■

В примере, представленном на рис. 34.13, выполняющий набор для формулы ϕ имеет вид $x_2 = 0$ и $x_3 = 1$. Соответствующая клика размера $k = 3$ состоит из вершин, отвечающих литералу $\neg x_2$ из первого подвыражения в скобках, литералу x_3 из второго выражения в скобках и литералу x_3 из третьего выражения в скобках. Поскольку клика не содержит вершин, соответствующих литералу x_1 или литералу $\neg x_1$, переменная x_1 в выполняющем наборе может принимать как значение 0, так и значение 1.

Обратите внимание, что при доказательстве теоремы 34.11 произвольный экземпляр задачи 3-CNF_SAT был сведен к экземпляру задачи CLIQUE, обладающему определенной структурой. Может показаться, что принадлежность задачи CLIQUE категории NP-сложных была доказана лишь для графов, все вершины которых можно разбить на тройки, причем таких, в которых отсутствуют ребра, соединяющие вершины одной и той же тройки. В самом деле, NP-сложность задачи CLIQUE была показана только для этого ограниченного случая, однако этого до-

казательства достаточно, чтобы сделать вывод о NP-сложности этой задачи для графа общего вида. Почему? Дело в том, что из наличия алгоритма с полиномиальным временем работы, решающего задачу CLIQUE с графами общего вида, следует существование такого алгоритма решения этой задачи с графами, имеющими ограниченную структуру.

Однако было бы недостаточно привести экземпляры задачи 3-CNF_SAT, обладающие какой-то особой структурой, к экземплярам задачи CLIQUE общего вида. Почему? Может случиться так, что экземпляры задачи 3-CNF_SAT, с помощью которых выполняется приведение, окажутся слишком "легкими", и к задаче CLIQUE приводится задача, не являющаяся NP-сложной.

Заметим также, что для приведения используется экземпляр задачи 3-CNF_SAT, но не ее решение. Было бы ошибкой основывать приведение в течение полиномиального времени на знании того, выполнима ли формула ϕ, поскольку неизвестно, как получить эту информацию в течение полиномиального времени.

34.5.2 Задача о вершинном покрытии

Вершинное покрытие (vertex cover) неориентированного графа $G = (V, E)$ — это такое подмножество $V' \subseteq V$, что если $(u, v) \in E$, то $u \in V'$ либо $v \in V'$ (либо справедливы оба эти соотношения). Другими словами, каждая вершина "покрывает" инцидентные ребра, а вершинное покрытие графа G — это множество вершин, покрывающих все ребра из множества E. ***Размером*** (size) вершинного покрытия называется количество содержащихся в нем вершин. Например, граф, изображенный на рис. 34.14*б*, имеет вершинное покрытие $\{w, z\}$ размера 2.

Задача о вершинном покрытии (vertex-cover problem) заключается в том, чтобы найти в заданном графе вершинное покрытие минимального размера. Переформулируем эту задачу оптимизации в виде задачи принятия решения, в которой требуется определить, содержит ли граф вершинное покрытие заданного размера k. Определим язык

$$\text{VERTEX\_COVER} = \{\langle G, k\rangle : \text{ граф } G \text{ имеет вершинное покрытие размера } k\}.$$

В сформулированной ниже теореме доказывается, что эта задача является NP-полной.

Теорема 34.12. Задача о вершинном покрытии является NP-полной.

Доказательство. Сначала покажем, что $\text{VERTEX\_COVER} \in \text{NP}$. Предположим, что задан граф $G = (V, E)$ и целое число k. В качестве сертификата выберем само вершинное покрытие $V' \subseteq V$. В алгоритме верификации проверяется, что $|V'| = k$, а затем для каждого ребра $(u, v) \in E$ проверяется, что $u \in V'$ или $v \in V'$. Такую верификацию можно выполнить непосредственно, как описано выше, в течение полиномиального времени.

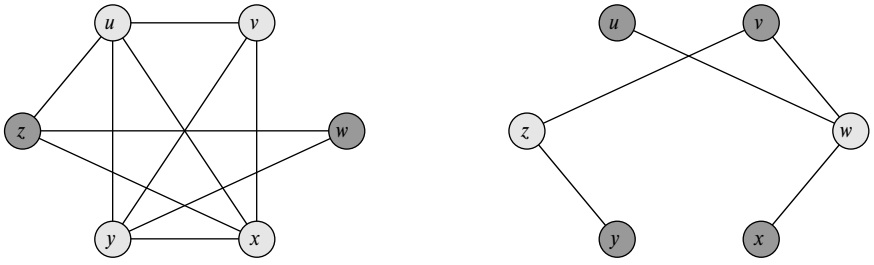


Рис. 34.14. Приведение задачи CLIQUE к задаче VERTEX_COVER

Докажем, что задача о вершинном покрытии NP-сложная, для чего покажем справедливость соотношения CLIQUE $\leqslant_{\text{P}}$ VERTEX_COVER. Это приведение основывается на понятии "дополнения" графа. ***Дополнение*** (complement) данного неориентированного графа $G = (V, E)$ определяется как $\bar{G} = \left(V, \bar{E}\right)$, где $\bar{E} = \{(u, v) : u, v \in V, u \neq v \text{ и } (u, v) \notin E\}$. Другими словами, $\bar{G}$ — это граф, содержащий те ребра, которые не содержатся в графе G. На рис. 34.14 показан граф и его дополнение, а также проиллюстрировано приведение задачи CLIQUE к задаче VERTEX_COVER.

Алгоритм приведения в качестве входных данных получает экземпляр задачи о клике $\langle G, k\rangle$. В этом алгоритме вычисляется дополнение $\bar{G}$, что легко осуществить в течение полиномиального времени. Выходом алгоритма приведения является экземпляр $\left\langle \bar{G}, |V| - k\right\rangle$ задачи о вершинном покрытии. Чтобы завершить доказательство, покажем, что это преобразование действительно является приведением: граф G содержит клику размера k тогда и только тогда, когда граф $\bar{G}$ имеет вершинное покрытие размером $|V| - k$.

Предположим, что граф G содержит клику $V' \subseteq V$ размером $|V'| = k$. Мы утверждаем, что $V - V'$ — вершинное покрытие графа $\bar{G}$. Пусть (u, v) — произвольное ребро из множества $\bar{E}$. Тогда $(u, v) \notin E$, из чего следует, что хотя бы одна из вершин u и v не принадлежит множеству V', поскольку каждая пара вершин из V' соединена ребром, входящим в множество E. Это эквивалентно тому, что хотя бы одна из вершин u и v принадлежит множеству $V - V'$, а, следовательно, ребро (u, v) покрывается этим множеством. Поскольку ребро (u, v) выбрано из множества $\bar{E}$ произвольным образом, каждое ребро из этого множества покрывается вершиной из множества $V - V'$. Таким образом, множество $V - V'$ размером $|V| - k$ образует вершинное покрытие графа $\bar{G}$.

Справедливо и обратное. Предположим, что граф $\bar{G}$ имеет вершинное покрытие $V' \subseteq V$, где $|V'| = |V| - k$. Тогда для всех пар вершин $u, v \in V$ из $(u, v) \in \bar{E}$ следует, что или $u \in V'$, или $v \in V'$, или справедливы оба эти утверждения. Обращение

следствия дает, что для всех пар вершин $u, v \in V$, если $u \notin V'$ и $v \notin V'$, то $(u, v) \in$ $\in E$. Другими словами, $V - V'$ — это клика, а ее размер равен $|V| - |V'| = k$. ■

Поскольку задача VERTEX_COVER является NP-полной, маловероятным представляется то, что удастся разработать алгоритм поиска вершинного покрытия минимального размера за полиномиальное время. Однако в разделе 35.1 представлен "приближенный алгоритм" с полиномиальным временем работы, позволяющий находить "приближенные" решения этой задачи. Размер вершинного покрытия, полученного в результате работы этого алгоритма, не более чем в 2 раза превышает размер минимального вершинного покрытия.

Таким образом, не стоит лишать себя надежды только из-за того, что задача NP-полная. Может оказаться, что для нее существует приближенный алгоритм с полиномиальным временем работы, позволяющий получать решения, близкие к оптимальным. В главе 35 описано несколько приближенных алгоритмов, предназначенных для решения NP-полных задач.

34.5.3 Задача о гамильтоновых циклах

Теперь вернемся к задаче о гамильтоновых циклах, определенной в разделе 34.2.

Теорема 34.13. Задача о гамильтоновых циклах является NP-полной.

Доказательство. Сначала покажем, что задача HAM_CYCLE принадлежит классу NP. Для заданного графа $G = (V, E)$ сертификат задачи имеет вид последовательности, состоящей из $|V|$ вершин, из которых состоит гамильтонов цикл. В алгоритме верификации проверяется, что каждая вершина этой последовательности входит в множество V ровно по одному разу и что первая вершина повторяется в конце, т.е. последовательность образует цикл в графе G. Другими словами, проверяется, что каждая пара последовательных вершин соединена ребром, а также что ребро соединяет первую и последнюю вершины последовательности. Подобную проверку можно выполнить в течение полиномиального времени.

Докажем теперь, что VERTEX_COVER $\leqslant_P$ HAM_CYCLE, откуда следует, что задача HAM_CYCLE NP-полная. Построим для заданного неориентированного графа $G = (V, E)$ и целого числа k неориентированный граф $G' = (V', E')$, в котором гамильтонов цикл содержится тогда и только тогда, когда размер вершинного покрытия графа G равен k.

Наше построение основывается на ***структурном элементе*** (widget), представляющем собой фрагмент графа, обеспечивающий его определенные свойства. На рис. 34.15*а* показан используемый нами структурный элемент. Для каждого ребра $(u, v) \in E$ строящийся граф G' будет содержать одну копию этого структурного элемента, обозначаемую как W_{uv}. Обозначим каждую вершину структурного

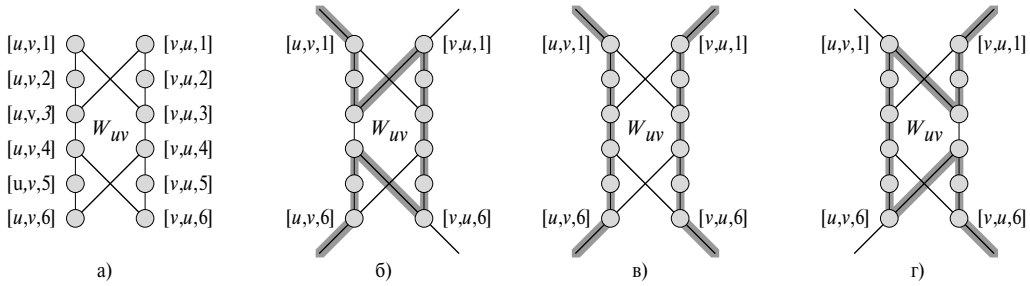


Рис. 34.15. Структурный элемент графа, использующийся в ходе приведения задачи о вершинном покрытии к задаче о гамильтоновых циклах

элемента W_{uv} как $[u, v, i]$ или $[v, u, i]$, где $1 \leqslant i \leqslant 6$, так что каждый структурный элемент W_{uv} содержит 12 вершин. Кроме того, он содержит 14 ребер, как показано на рис.34.15*а*.

Нужные свойства графа обеспечиваются не только внутренней структурой описанного выше элемента, но и наложением ограничений на связи между структурным элементом и остальной частью строящегося графа G'. В частности, наружу структурного элемента W_{uv} будут выходить ребра только из вершин $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$ и $[v, u, 6]$. Любой гамильтонов цикл G' должен проходить по ребрам структурного элемента W_{uv} одним из трех способов, показанных на рис. 34.15*б–г*. Если цикл проходит через вершину $[u, v, 1]$, выйти он должен через вершину $[u, v, 6]$, и при этом он либо проходит через все 12 вершин структурного элемента (рис. 34.15*б*), либо только через 6 его вершин, с $[u, v, 1]$ по $[u, v, 6]$ (рис. 34.15*в*). В последнем случае цикл должен будет должен повторно войти в структурный элемент, посещая вершины с $[v, u, 1]$ по $[v, u, 6]$. Аналогично, если цикл входит в структурный элемент через вершину $[v, u, 1]$, то он должен выйти из вершины $[v, u, 6]$, посетив на своем пути либо все 12 вершин (рис. 34.15*г*), либо 6 вершин, начиная с вершины $[v, u, 1]$ и заканчивая вершиной $[v, u, 6]$ (рис. 34.15*в*). Никакие другие варианты прохождения всех 12 вершин структурного элемента невозможны. В частности, невозможно таким образом построить 2 отдельных пути, один из которых соединяет вершины $[u, v, 1]$ и $[v, u, 6]$, а другой — вершины $[v, u, 1]$ и $[u, v, 6]$, чтобы объединение этих путей содержало все вершины структурного элемента.

Единственные вершины, содержащиеся в множестве V', кроме вершин структурного элемента, — ***переключающие вершины*** (selector vertex) $s_1, s_2, \ldots, s_k$. Ребра графа G', инцидентные переключающей вершине, используются для выбора k вершин из покрытия графа G.

Кроме ребер, входящих в состав структурных элементов, множество E' содержит ребра двух других типов, показанных на рис. 34.16. Во-первых, для каждой вершины $u \in V$ добавляются ребра, соединяющие пары структурных элементов

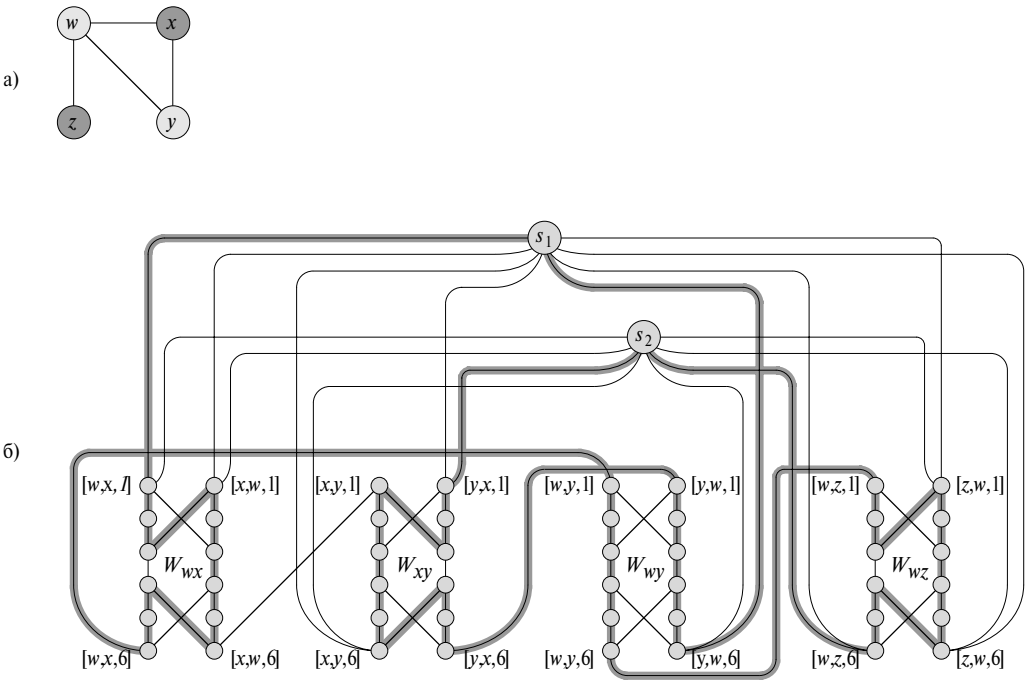


Рис. 34.16. Приведение экземпляра задачи о вершинном покрытии к экземпляру задачи о гамильтоновом цикле

в таком порядке, чтобы получился путь, содержащий все структурные элементы, соответствующие ребрам, инцидентным вершине u графа G. Вершины, смежные с каждой вершиной $u \in V$, упорядочиваются произвольным образом как $u^{(1)}, u^{(2)}, \ldots, u^{(\mathrm{degree}(u))}$, где $\mathrm{degree}\,(u)$ — количество вершин, смежных с вершиной u. В графе G' создается путь, проходящий через все структурные элементы, соответствующие инцидентным ребрам вершины u. Для этого к множеству E' добавляются ребра $\left\{\left(\left[u, u^{(i)}, 6\right], \left[u, u^{(i+1)}, 1\right]\right) : 1 \leqslant i \leqslant \mathrm{degree}\,(u) - 1\right\}$. Например, на рис. 34.16 вершины, смежные с вершиной w, упорядочиваются как x, y, z, так что граф G', изображенный в части *б* рисунка, содержит ребра $([w, x, 6], [w, y, 1])$ и $([w, y, 6], [w, z, 1])$. Для каждой вершины $u \in V$ эти ребра графа G' заполняют путь, содержащий все структурные элементы, соответствующие ребрам, инцидентным вершине u графа G.

Интуитивные соображения, обосновывающие наличие этих ребер, заключаются в том, что если из вершинного покрытия графа G выбирается вершина $u \in V$, то в графе G' можно построить путь из вершины $\left[u, u^{(1)}, 1\right]$ к вершине $\left[u, u^{(\mathrm{degree}(u))}, 6\right]$, "покрывающий" все структурные элементы, соответствующие ребрам, инцидентным вершине u. Другими словами, для каждого из таких структурных элементов, скажем, структурного элемента $W_{u,u^{(i)}}$, этот путь проходит

либо по всем 12 вершинам (если вершина u входит в вершинное покрытие, а вершина $u^{(i)}$ — нет), либо только по шести вершинам $\left[u, u^{(i)}, 1\right]$, $\left[u, u^{(i)}, 2\right]$, ..., $\left[u, u^{(i)}, 6\right]$ (если вершинному покрытию принадлежит и вершина u, и вершина $u^{(i)}$).

В части *а* рисунка рис. 34.16 показан неориентированный граф G; его вершинное покрытие размера 2, состоящее из вершин w и y, выделено светло-серым цветом. В части *б* изображен неориентированный граф G', полученный в процессе приведения; гамильтонов путь, соответствующий вершинному покрытию, выделен в графе серым цветом. Вершинное покрытие $\{w, y\}$ соответствует содержащемся в гамильтоновом цикле ребрам $(s_1, [w, x, 1])$ и $(s_2, [y, x, 1])$.

Наконец, последний тип ребер множества E' соединяет первую $\left[u, u^{(1)}, 1\right]$ и последнюю $\left[u, u^{(\text{degree}(u))}, 6\right]$ вершины каждого из этих путей с каждой из переключающих вершин. Другими словами, в множество включаются ребра

$$\left\{\left(s_j, \left[u, u^{(1)}, 1\right]\right) : u \in V \text{ и } 1 \leqslant j \leqslant k\right\} \cup \\ \cup \left\{\left(s_j, \left[u, u^{(\text{degree}(u))}, 6\right]\right) : u \in V \text{ и } 1 \leqslant j \leqslant k\right\}.$$

Теперь покажем, что размер графа G' выражается полиномиальной функцией от размера графа G, поэтому граф G' можно сконструировать в течение полиномиального времени от размера графа G. Множество вершин графа G' состоит из вершин, входящих в состав структурных элементов, и переключающих вершин. Каждый структурный элемент содержит 12 вершин, и еще имеется $k \leqslant |V|$ переключающих вершин, поэтому в итоге получается

$$\left|V'\right| = 12\,|E| + k \leqslant 12\,|E| + |V|$$

вершин. Множество ребер графа G' состоит из тех, которые принадлежат структурным элементам, тех, которые соединяют структурные элементы, и тех, которые соединяют переключающие вершины со структурными элементами. Каждый структурный элемент содержит по 14 ребер, поэтому все структурные элементы в совокупности содержат $14\,|E|$ ребер. Для каждой вершины $u \in V$ имеется $\text{degree}\,(u) - 1$ ребер между структурными элементами, так что в результате суммирования по всем вершинам множества V получается

$$\sum_{u \in V} (\text{degree}\,(u) - 1) = 2\,|E| - |V|$$

ребер, соединяющих структурные элементы. Наконец, по два ребра приходится на каждую пару, состоящую из одной переключающей вершины и одной вершины множества V. Всего таких ребер получается $2k\,|V|$, а общее количество всех ребер

графа G' равно

$$\begin{aligned}\left|E'\right| &= (14\,|E|) + (2\,|E| - |V|) + (2k\,|V|) = \\ &= 16\,|E| + (2k-1)\,|V| \leqslant \\ &\leqslant 16\,|E| + (2\,|V| - 1)\,|V|\,.\end{aligned}$$

Теперь покажем, что преобразование графа G в граф G' — это приведение. Другими словами, необходимо показать, что граф G имеет вершинное покрытие размера k тогда и только тогда, когда граф G' имеет гамильтонов цикл.

Предположим, что граф $G = (V, E)$ содержит вершинное покрытие $V^* \subseteq V$ размера k. Пусть $V^* = \{u_1, u_2, \ldots, u_k\}$. Как видно из рис. 34.16, гамильтонов цикл графа G' образуется путем включения в него следующих ребер[8] для каждой вершины $u_j \in V^*$. В цикл включаются ребра $\left\{ \left(\left[u_j, u_j^{(i)}, 6\right], \left[u_j, u_j^{(i+1)}, 1\right] \right) : 1 \leqslant j \leqslant \right.$ $\left. \leqslant \text{degree}\,(u_j) \right\}$, которые соединяют все структурные элементы, соответствующие ребрам, инцидентным вершинам u_j. Включаются также ребра, содержащиеся в этих структурных элементах, как показано на рис. 34.15*б–г*, в зависимости от того, покрывается ли ребро одним или двумя вершинами множества V^*. Гамильтонов цикл также включает в себя ребра

$$\begin{aligned}&\left\{ \left(s_j, \left[u_j, u_j^{(1)}, 1\right] \right) : 1 \leqslant j \leqslant k \right\} \cup \\ &\qquad \cup \left\{ \left(s_{j+1}, \left[u_j, u_j^{(\text{degree}(u_j))}, 6\right] \right) : 1 \leqslant j \leqslant k-1 \right\} \cup \\ &\qquad \cup \left\{ \left(s_1, \left[u_k, u_k^{(\text{degree}(u_k))}, 6\right] \right) \right\}\end{aligned}$$

Ознакомившись с рис. 34.16, читатель может убедиться, что эти ребра действительно образуют цикл. Этот цикл начинается в вершине s_1, проходит через все структурные элементы, соответствующие ребрам, инцидентным вершине u_1, затем направляется к вершине u_2, проходит через все структурные элементы, соответствующие ребрам, инцидентным вершине u_2, и т.д., пока снова не вернется к вершине s_1. Каждый структурный элемент проходится однократно или дважды, в зависимости от того, одна или две вершины множества V^* покрывают соответствующее ему ребро. Поскольку V^* — вершинное покрытие графа G, каждое ребро из множества E инцидентно некоторой вершине множества V^*, поэтому цикл проходит через все вершины каждого структурного элемента графа G'. Поскольку он также проходит через все переключающие вершины, то этот цикл — гамильтонов.

[8] Технически определение цикла формулируется в терминах вершин, а не ребер (см. раздел Б.4). Для ясности здесь эти обозначения видоизменяются, а гамильтонов цикл определяется в терминах ребер.

Проведем рассуждения в обратном направлении. Предположим, что граф $G' = (V', E')$ содержит гамильтонов цикл $C \subseteq E'$. Утверждается, что множество

$$V^* = \left\{ u \in V : \left(s_j, \left[u, u^{(1)}, 1 \right] \right) \in C \text{ для некоторого } 1 \leqslant j \leqslant k \right\} \qquad (34.4)$$

является вершинным покрытием графа G. Чтобы убедиться, что это действительно так, разобьем цикл C на максимальные пути, которые начинаются в некоторой переключающей вершине s_i, проходят через ребро $\left(s_i, \left[u, u^{(1)}, 1\right]\right)$ для некоторой вершины $u \in V$ и оканчиваются в переключающей вершине s_j, не пересекая при этом никакие другие переключающие вершины. Назовем каждый такой путь "покрывающим". По способу построения графа G', каждый покрывающий путь должен начинаться в некоторой вершине s_i, включать в себя ребро $\left(s_i, \left[u, u^{(1)}, 1\right]\right)$ для некоторой вершины $u \in V$, проходить через все структурные элементы, соответствующие ребрам из множества E, инцидентным вершине u, и оканчиваться в некоторой переключающей вершине s_j. Обозначим такой покрывающий путь как p_u и, в соответствии с уравнением (34.4), включим вершину u в множество V^*. Любой структурный элемент, через который проходит путь p_u, должен быть структурным элементом W_{uv} или структурным элементом W_{vu} для некоторой вершины $v \in V$. О каждом структурном элементе, через который проходит путь p_u, можно сказать, что по его вершинам проходит один или два покрывающих пути. Если такой покрывающий путь один, то ребро $(u, v) \in E$ покрывается в графе G вершиной u. Если же через структурный элемент проходят два пути, то один из них, очевидно, путь p_u, а другой должен быть путем p_v. Из этого следует, что $v \in V^*$, и ребро $(u, v) \in E$ покрывается и вершиной u, и вершиной v. Поскольку все вершины из каждого структурного элемента посещаются некоторым покрывающим путем, видно, что каждое ребро из множества E покрывается некоторой вершиной из множества V^*. ■

34.5.4 Задача о коммивояжере

В ***задаче о коммивояжере*** (traveling-salesman problem), которая тесно связана с задачей о гамильтоновом цикле, коммивояжер должен посетить n городов. Моделируя задачу в виде полного графа с n вершинами, можно сказать, что коммивояжеру нужно совершить ***тур*** (tour), или гамильтонов цикл, посетив каждый город ровно по одному разу и завершив путешествие в том же городе, из которого он выехал. С каждым переездом из города i в город j связана некоторая стоимость $c(i, j)$, выражающаяся целым числом, и коммивояжеру нужно совершить тур таким образом, чтобы общая стоимость (т.е. сумма стоимостей всех переездов) была минимальной. Например, на рис. 34.17 изображен самый дешевый тур $\langle u, w, v, x, u \rangle$, стоимость которого равна 7. Вот как выглядит формальный язык

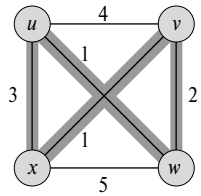


Рис. 34.17. Экземпляр задачи о коммивояжере; выделенные серым цветом ребра представляют самый дешевый тур, стоимость которого равна 7

для соответствующей задачи принятия решения:

$$\begin{aligned}\mathrm{TSP} = \{\langle G, c, k\rangle : {} & G = (V, E) \text{ — полный граф,} \\ & c \text{ — функция } V \times V \to \mathbf{Z}, \\ & k \in \mathbf{Z} \text{ и} \\ & G \text{ содержит тур коммивояжера со стоимостью,} \\ & \quad \text{не превышающей } k\}.\end{aligned}$$

Согласно сформулированной ниже теореме, существование быстрого алгоритма для решения задачи о коммивояжере маловероятно.

Теорема 34.14. Задача о коммивояжере (TSP) является NP-полной.

Доказательство. Сначала докажем, что TSP принадлежит классу NP. В качестве сертификата для заданного экземпляра задачи будет использоваться последовательность n вершин, из которых состоит тур. В алгоритме верификации проверяется, что в этой последовательности все вершины содержатся ровно по одному разу, а также суммируются стоимости всех ребер тура и проверяется, что эта сумма не превышает k. Очевидно, что этот процесс можно выполнить в течение полиномиального времени.

Чтобы убедиться в том, что задача TSP NP-сложная, покажем, что HAM_CYCLE $\leqslant_{\mathrm{P}}$ TSP. Пусть $G = (V, E)$ — экземпляр задачи HAM_CYCLE. Экземпляр TSP конструируется следующим образом. Сформируем полный граф $G' = (V, E')$, где $E' = \{(i, j) : i, j \in V \text{ и } i \neq j\}$. Определим также функцию стоимости c:

$$c\,(i, j) = \begin{cases} 0 & \text{если } (i, j) \in E, \\ 1 & \text{если } (i, j) \notin E. \end{cases}$$

(Заметим, что поскольку граф G неориентированный, в нем отсутствуют петли, поэтому для всех вершин $v \in V$ выполняется равенство $c\,(v, v) = 1$.) Тогда в качестве экземпляра TSP выступает $\langle G', c, 0\rangle$, который легко составить в течение полиномиального времени.

Теперь покажем, что граф G содержит гамильтонов цикл тогда и только тогда, когда граф G' включает в себя тур, стоимость которого не превышает 0. Предположим, что граф G содержит гамильтонов цикл h. Каждое ребро цикла h принадлежит множеству E, поэтому его стоимость в графе G' равна 0. Таким образом, стоимость цикла h в графе G' равна 0. И обратно — предположим, что граф G' содержит тур h', стоимость которого не превышает 0. Поскольку стоимости ребер графа G' равны 0 или 1, стоимость тура h' равна 0, и стоимость каждого ребра из тура должна равняться 0. Таким образом, тур h' содержит только ребра из множества E. Можно сделать вывод, что тур h' — это гамильтонов цикл в графе G. ■

34.5.5 Задача о сумме подмножества

Следующая NP-полная задача, которая будет рассмотрена, принадлежит к разряду арифметических. В ***задаче о сумме подмножества*** (subset-sum problem) задается конечное множество $S \subset \mathbf{N}$ и ***целевое значение*** (target) $t \in \mathbf{N}$. Спрашивается, существует ли подмножество $S' \subseteq S$, сумма элементов которого равна t. Например, если

$$S = \{1, 2, 7, 14, 49, 98, 343, 686, 2\,409, 2\,793, 16\,808, 17\,206, 117\,705, 117\,993\},$$

а $t = 138\,457$, то решением является подмножество

$$S' = \{1, 2, 7, 98, 343, 686, 2\,409, 17\,206, 117\,705\}.$$

Как обычно, определим язык, соответствующий задаче:

$$\text{SUBSET\_SUM} = \Big\{ \langle S, t\rangle : \text{существует подмножество } S' \subseteq S \text{ такое, что } t = \sum\nolimits_{s \in S'} s \Big\}.$$

Как в любой арифметической задаче, важно помнить, что в нашем стандартном коде предполагается, что входные целые числа кодируются бинарными значениями. При этом предположении можно показать, что вероятность наличия быстрого алгоритма, позволяющего решить задачу о сумме подмножества, очень мала.

Теорема 34.15. Задача о сумме подмножества является NP-полной.

Доказательство. Чтобы показать NP-полноту задачи SUBSET_SUM для экземпляра $\langle S, t\rangle$, выберем в качестве сертификата подмножество S'. Проверку равенства $t = \sum_{s \in S'} s$ в алгоритме верификации можно выполнить в течение полиномиального времени.

Теперь покажем, что 3-CNF_SAT $\leqslant_P$ SUBSET_SUM. Если задана 3-CNF формула ϕ, зависящая от переменных $x_1, x_2, \ldots, x_n$ и содержащая подвыражения в скобках $C_1, C_2, \ldots, C_k$, в каждое из которых входит ровно по три различных литерала, то в алгоритме приведения строится такой экземпляр $\langle S, t\rangle$ задачи о сумме подмножества, что формула ϕ выполнима тогда и только тогда, когда существует подмножество S, сумма элементов которого равна t. Не теряя общности, можно сделать два упрощающих предположения о формуле ϕ. Во-первых, ни в одном подвыражении в скобках не содержится одновременно и переменная, и ее отрицание, потому что такое выражение автоматически удовлетворяется при любых значениях этой переменной. Во-вторых, каждая переменная входит хотя бы в одно подвыражение в скобках, так как в противном случае безразлично, какое значение ей присваивается.

В процессе приведения в множестве S создается по два числа для каждой переменной x_i и для каждого выражения в скобках C_j. Эти числа будут создаваться в десятичной системе счисления, и все они будут содержать по $n + k$ цифр, каждая из которых соответствует переменной или выражению в скобках. Основание системы счисления 10 (и, как мы увидим, и другие основания) обладают свойством, необходимым для предотвращения переноса значений из младших разрядов в старшие.

Как видно из табл. 34.2, множество S и целевое значение t конструируются следующим образом. Присвоим каждому разряду числа метку, соответствующую либо переменной, либо выражению в скобках. Цифры, которые находятся в k младших разрядах, отвечают выражениям в скобках, а цифры в n старших разрядах отвечают переменным.

- Все цифры целевого значения t, соответствующие переменным, равны 1, а соответствующие подвыражениям — равны 4.
- Каждой переменной x_i в множестве S сопоставляются два целых числа, — v_i и v'_i. В каждом из этих чисел в разряде с меткой x_i находится значение 1, а в разрядах, соответствующих всем другим переменным, — значения 0. Если литерал x_i входит в подвыражение C_j, то цифра с меткой C_j в числе v_i равна 1. Если же выражение в скобках C_j содержит литерал $\neg x_i$, то 1 равна цифра с меткой C_j в числе v'_i. Все остальные цифры, метки которых соответствуют другим выражениям в скобках, в числах v_i и v'_i равны 0.

  Значения v_i и v'_i в множестве S не повторяются. Почему? Если $l \neq i$, то число, содержащееся в n старших разрядах значений v_l или v'_l, не могут быть равны соответствующим числам из значений v_i или v'_i. Кроме того, согласно сделанным выше упрощающим предположениям, такое равенство невозможно для чисел, содержащихся в k младших разрядах значений v_i и v'_i. Если бы числа v_i и v'_i были равны, то литералы x_i и $\neg x_i$ должны были бы входить в один и тот же набор подвыражений в скобках. Однако,

согласно предположению, ни одно подвыражение не содержит одновременно и x_i, и $\neg x_i$, и хотя бы один из этих литералов входит в одно из выражений в скобках. Поэтому должно существовать какое-то подвыражение C_j, благодаря которому числа v_i и v'_i различаются.

- Каждому подвыражению C_j в множестве S сопоставляются два целых числа s_j и s'_j. Каждое из них содержит нули в разрядах, метки которых отличаются от C_j. В числе s_j цифра с меткой C_j равна 1, а в числе s'_j эта цифра равна 2. Такие целые числа выступают в роли "фиктивных переменных", благодаря которым в суммарном значении каждая цифра, соответствующая выражению в скобках, равна 4.

  Простая проверка для примера, проиллюстрированного в табл. 34.2, показывает, что никакие значения s_j и s'_j в множестве S не повторяются.

В приведенном примере рассматривается 3-CNF формула $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, где $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee \neg x_2 \vee \times_3)$, $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$ и $C_4 = (x_1 \vee x_2 \vee x_3)$. Выполняющий набор для формулы ϕ имеет вид $\langle x_1 = 0, x_2 = 0, x_3 = 1\rangle$. Множество S, полученное в процессе приведения, состоит из представленных в таблице чисел в десятичной системе счисления; считывая их сверху вниз, находим, что

$$S = \{1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\};$$

целевое значение $t = 1114444$. Подмножество $S' \subseteq S$ выделено светлой штриховкой и содержит v'_1, v'_2 и v_3, соответствующие выполняющему набору. В него также входят помеченные той же штриховкой фиктивные переменные, обеспечивающие получение значений 4 в цифрах целевого значения, помеченных подвыражениями.

Заметим, что максимальная сумма цифр в каждом из разрядов равна 6. Это возможно в цифрах, метки которых соответствуют выражениям в скобках (три единицы от значений v_i и v'_i плюс 1 и 2 от значений s_j и s'_j). Поэтому эти значения интерпретируются как числа в десятичной системе счисления, чтобы не было переноса из младших разрядов в старшие[9].

Такое приведение можно выполнить в течение полиномиального времени. Множество S содержит $2n + 2k$ значений, в каждом из которых по $n + k$ цифр, поэтому время, необходимое для получения всех цифр, выражается полиномиальной функцией от $n + k$. Целевое значение t содержит $n + k$ цифр, и в процессе приведения каждую из них можно получить в течение фиксированного времени.

Теперь покажем, что 3-CNF формула выполнима тогда и только тогда, когда существует подмножество $S' \subseteq S$, сумма элементов которого равна t. Сначала

[9] Фактически, подошло бы любое основание $b \geqslant 7$. Так, экземпляр задачи, который рассматривается в начале этого подраздела, представляет собой множество S и целевое значение t из табл. 34.2, где все значения рассматриваются как числа в семеричной системе счисления, отсортированные в порядке убывания.

Таблица 34.2. Процесс приведения задачи 3-CNF_SAT к задачеSUBSET_SUM

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v'_3	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s'_1	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s'_2	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s'_3	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s'_4	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

предположим, что у формулы ϕ имеется выполняющий набор. Если в нем $x_i = 1$ (где $i = 1, 2, \ldots, n$), то число v_i включается в множество S'. В противном случае в это множество включается число v'_i. Другими словами, в множество S' включаются именно те значения v_i и v'_i, которым в выполняющем наборе соответствуют литералы, равные 1. Включая в множество S' либо число v_i, либо число v'_i, но не оба этих числа для всех i, и помещая в значениях s_j и s'_j нули во все разряды с метками, соответствующими переменным, мы видим, что сумма цифр в этих разрядах по всем значениям множества S' должна быть равной 1, что совпадает с цифрами в этих разрядах целевого значения t. Поскольку все подвыражения в скобках выполняется, в каждом таком подвыражении имеется литерал, значение которого равно 1. Поэтому каждая цифра, соответствующая подвыражению, включает единицу, вносимую в сумму благодаря вкладу значения v_i или значения v'_i из множества S'. В действительности в каждом выражении в скобках единице могут быть равны 1, 2 или 3 литерала, поэтому в каждом разряде, соответствующем выражению в скобках, сумма цифр по значениям v_i и v'_i множества S' равна 1, 2 или 3. (Например, в примере, проиллюстрированном в табл. 34.2, в выполняющем наборе единице равны литералы $\neg x_1$, $\neg x_2$ и x_3. Каждое из выражений в скобках

C_1 и C_4 содержит ровно по одному из этих литералов, поэтому числа v'_1, v'_2 и v_3 в совокупности дают единичный вклад в цифры, соответствующие выражениям C_1 и C_4. Подвыражение C_2 содержит по два этих литерала, поэтому числа v'_1, v'_2 и v_3 в совокупности дают вклад 2 в цифру, соответствующую C_2. Подвыражение C_3 содержит все три перечисленных литерала, и числа v'_1, v'_2 и v_3 дают вклад 3 в цифру, соответствующую выражению C_3.) Целевое значение, равное 4, достигается в каждом разряде с меткой, отвечающей подвыражению в скобках, путем добавления в множество S' непустого множества из соответствующих фиктивных переменных $\left\{s_j, s'_j\right\}$. (В примере, проиллюстрированном в табл. 34.2, множество S' включает значения $\{s_1, s'_1, s'_2, s_3, s_4, s'_4\}$.) Поскольку сумма цифр по всем значениям множества S' во всех разрядах совпадает с соответствующими цифрами целевого значения t, и при суммировании не производится перенос значений из младших разрядов в старшие, то сумма значений множества S' равна t.

Теперь предположим, что имеется подмножество $S' \subseteq S$, сумма элементов которого равна t. Для каждого значения $i = 1, 2, \ldots, n$ это подмножество должно включать в себя ровно по одному из значений v_i или v'_i, потому что в противном случае сумма цифр в разрядах, соответствующих переменным, не была бы равна 1. Если $v_i \in S'$, то выполняем присваивание $x_i = 1$. В противном случае $v'_i \in S'$, и выполняется присваивание $x_i = 0$. Мы утверждаем, что в результате такого присваивания выполняется каждое подвыражение C_j, $j = 1, 2, \ldots, k$. Чтобы доказать это утверждение, заметим, что для того, чтобы сумма цифр в разряде с меткой C_j была равна 4, подмножество S' должно содержать хотя бы одно из значений v_i или v'_i, в котором 1 находится в разряде с меткой C_j, так как суммарный вклад фиктивных переменных s_j и s'_j не превышает 3. Если подмножество S' включает в себя значение v_i, в котором в этом разряде содержится 1, то в подвыражение C_j входит литерал x_i. Поскольку при $v_i \in S'$ выполняется присваивание $x_i = 1$, то подвыражение C_j выполняется. Если множество S' включает в себя значение v'_i, в котором в этом разряде содержится 1, то в подвыражение C_j входит литерал $\neg x_i$. Так как при $v'_i \in S'$ выполняется присваивание $x_i = 0$, то подвыражение C_j снова выполняется. Таким образом, в формуле ϕ выполняются все подвыражения в скобках, и на этом доказательство теоремы завершается. ■

Упражнения

34.5-1. В ***задаче об изоморфизме подграфу*** (subgraph-isomorphism problem) задаются два графа (G_1 и G_2) и спрашивается, изоморфен ли граф G_1 какому-нибудь подграфу графа G_2. Покажите, что эта задача NP-полная.

34.5-2. В ***задаче 0-1 целочисленного программирования*** (0-1 integer-programming problem) задается целочисленная матрица A размером $m \times n$ и целочисленный m-компонентный вектор b и спрашивается, существует ли

целочисленный n-компонентный вектор x, элементы которого являются элементами множества $\{0, 1\}$, удовлетворяющий неравенству $Ax \leqslant b$. Докажите, что задача 0-1 целочисленного программирования является NP-полной. (*Указание*: приведите к этой задаче задачу 3-CNF_SAT.)

34.5-3. ***Задача целочисленного линейного программирования*** (integer linear-programming problem) похожа на задачу 0-1 целочисленного программирования, описанную в упражнении 34.5-2, но в ней компоненты вектора x могут быть любыми целыми числами, а не только нулем и единицей. Исходя из предположения, согласно которому задача 0-1 целочисленного программирования является NP-полной, покажите, что задача целочисленного линейного программирования также NP-полная.

34.5-4. Покажите, что задача о сумме подмножества разрешима в течение полиномиального времени, если целевое значение t выражено в унарной системе счисления, т.е. представлено последовательностью из t единиц.

34.5-5. В ***задаче о разделении множества*** (set-partition problem) в качестве входных данных выступает множество чисел S. Спрашивается, можно ли это числа разбить на два множества A и $\bar{A} = S - A$ таким образом, чтобы $\sum_{x \in A} x = \sum_{x \in \bar{A}} x$. Покажите, что задача о разделении множества является NP-полной.

34.5-6. Покажите, что задача о гамильтоновом пути NP-полная.

34.5-7. ***Задача о самом длинном простом цикле*** (longest-simple-cycle problem) — это задача, в которой в заданном графе находится простой цикл (без повторения вершин) максимальной длины. Покажите, что эта задача — NP-полная.

34.5-8. В ***половинной задаче о 3-CNF выполнимости*** (half 3-CNF satisfiability problem) задается 3-CNF формула ϕ с n переменными и m подвыражениями в скобках, где m — четное. Нужно определить, существует ли набор значений переменных формулы ϕ, при котором результат ровно половины выражений в скобках равен 0, а результат второй половины этих выражений равен 1. Докажите, что половинная задача о 3-CNF выполнимости является NP-полной.

Задачи

34-1. **Независимое множество**

Независимым множеством (independent set) графа $G = (V, E)$ называется такое подмножество вершин $V' \subseteq V$, что каждое ребро из множества E инцидентно хотя бы одной вершине из множества V'. ***Задача о независимом множестве*** (independent-set problem) заключается в том, чтобы

найти в заданном графе G независимое множество максимального размера.

а) Сформулируйте задачу принятия решения, соответствующую задаче о независимом множестве, и докажите, что она является NP-полной. (*Указание*: приведите к этой задаче задачу о клике.)

б) Предположим, что для задачи принятия решения, определенной в части *а*, имеется подпрограмма в виде "черного ящика", решающая эту задачу. Сформулируйте алгоритм поиска независимого множества, имеющего максимальный размер. Время работы этого алгоритма должно выражаться полиномиальной функцией от величин $|V|$ и $|E|$. При этом предполагается, что каждый запрос подпрограммы учитывается как одна операция.

в) Несмотря на то, что задача о независимом множестве является NP-полной, некоторые особые случаи разрешимы в течение полиномиального времени.

г) Разработайте эффективный алгоритм, позволяющий решить задачу о независимом множестве, если степень каждой вершины графа G равна 2. Проанализируйте время работы этого алгоритма и докажите его корректность.

д) Разработайте эффективный алгоритм, позволяющий решить задачу о независимом множестве, если граф G — двудольный. Проанализируйте время работы этого алгоритма и докажите его корректность. (*Указание*: воспользуйтесь результатами, полученными в разделе 26.3.)

34-2. **Бонни и Клайд**

Бонни и Клайд только что ограбили банк. У них есть мешок денег, который нужно разделить. В каждом из описанных ниже сценариев требуется либо сформулировать алгоритм с полиномиальным временем работы, либо доказать, что задача NP-полная. В каждом случае в качестве входных данных выступает список, состоящий из n содержащихся в мешке элементов, а также стоимость каждого из них.

а) В мешке n монет двух различных номинаций: одни монеты стоят x долларов, а другие — y долларов. Деньги следует разделить поровну.

б) Имеется n монет произвольного количества различных номиналов, но при этом каждый номинал является неотрицательной целой степенью двойки, т.е. возможны такие номиналы: 1 доллар, 2 доллара, 4 доллара и т.д. Деньги следует разделить поровну.

в) Имеется n чеков, по невероятному совпадению выписанных на имя "Бонни или Клайд". Нужно разделить чеки таким образом, чтобы по ним можно было получить одинаковые суммы.

г) Как и в случае *в*, имеется n чеков, но на этот раз допускается расхождение в суммах, которое не должно превышать 100 долларов.

34-3. **Раскраска графов**

Если задан неориентированный граф $G = (V, E)$, то его ***k-раскрашиванием*** (k-coloring) называется функция $c : V \to \{1, 2, \ldots, k\}$, такая что $c(u) \neq c(v)$ для каждого ребра $(u, v) \in E$. Другими словами, числа $1, 2, \ldots, k$ представляют k цветов, и смежные вершины должны быть разного цвета. ***Задача о раскрашивании графов*** (graph-coloring problem) заключается в том, чтобы определить минимальное количество цветов, необходимых для раскрашивания заданного графа.

а) Сформулируйте эффективный алгоритм 2-раскрашивания графа, если такое раскрашивание возможно.

б) Переформулируйте задачу о раскрашивании графов в виде задачи принятия решения. Покажите, что эта задача разрешима в течение полиномиального времени тогда и только тогда, когда задача о раскрашивании графов разрешима в течение полиномиального времени.

в) Пусть язык 3-COLOR представляет собой множество графов, для которых возможно 3-раскрашивание. Покажите, что если задача 3-COLOR — NP-полная, то задача принятия решения из части *б* тоже NP-полная.

Чтобы доказать NP-полноту задачи 3-COLOR, воспользуемся приведением к этой задаче задачи 3-CNF_SAT. Для заданной формулы ϕ, содержащей m выражений в скобках и n переменных $x_1, x_2, \ldots, x_n$ конструируется граф $G = (V, E)$ описанным ниже способом. Множество V содержит по одной вершине для каждой переменной, по одной вершине для каждого отрицания переменной, по 5 вершин для каждого подвыражения в скобках и 3 специальных вершины: TRUE, FALSE и RED. Ребра в этом графе могут быть двух типов: "литеральные" ребра, которые не зависят от подвыражений в скобках, и "дизъюнктивные" ребра, которые от них зависят. Литеральные ребра образуют треугольник на специальных вершинах, а также образуют треугольник на вершинах x_i, $\neg x_i$ и RED для $i = 1, 2, \ldots, n$.

г) Докажите, что при любом 3-раскрашивании c графа, состоящего из литеральных ребер, из каждой пары вершин x_i, $\neg x_i$ одна окрашена

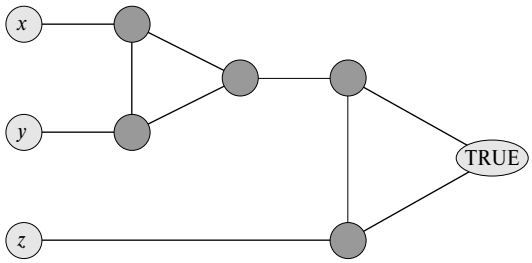


Рис. 34.18. Структурный элемент, который соответствует подвыражению $(x \vee y \vee z)$, использующийся в задаче 34-3

как $c\,(\text{TRUE})$, а другая — как $c\,(\text{FALSE})$. Покажите, что для любого набора значений функции ϕ существует 3-раскрашивание графа, содержащего только литеральные ребра.

Дизъюнктивные ребра нужны для того, чтобы наложить на 3-раскрашивание условия, соответствующие истинности дизъюнкций, составляющих формулу ϕ. Структурный элемент, показанный на рис. 34.18, соответствует подвыражению $(x \vee y \vee z)$. Структурный элемент, соответствующий каждому подвыражению, состоит из трех вершин для входящих в него литералов, пяти вспомогательных вершин (на рисунке они выделены темным цветом), соединенных с входящими в выражение литералами, и с особой вершиной TRUE, как показано на рисунке.

д) Докажите, что если каждая из вершин x, y и z окрашена в один из двух цветов — $c\,(\text{TRUE})$ или $c\,(\text{FALSE})$, — то для изображенного на рисунке структурного элемента правильное 3-раскрашивание возможно тогда и только тогда, когда цвет хотя бы одной из вершин x, y и z — $c\,(\text{TRUE})$.

е) Завершите доказательство NP-полноты задачи 3-COLOR.

34-4. **Расписание с прибылью и сроками выполнения**

Предположим, что имеется одна вычислительная машина и n заданий $a_1, a_2, \ldots, a_n$. Каждое задание a_j характеризуется временем выполнения t_j, прибылью p_j и конечным сроком выполнения d_j. В каждый момент времени машина может обрабатывать только одно задание, причем задание a_j после запуска должно без прерываний выполняться в течение времени t_j. Если задание a_j будет выполнено до наступления конечного срока его выполнения d_j, то будет получена прибыль p_j, но если конечный срок выполнения будет упущен, то и прибыли не будет. Сформулируем такую задачу оптимизации: пусть для множества n заданий

известны время обработки, прибыль и время выполнения; требуется составить расписание таким образом, чтобы были выполнены все задания и общая сумма прибыли была максимальной.

а) Сформулируйте эту задачу в виде задачи принятия решения.

б) Покажите, что эта задача принятия решения NP-полная.

в) Сформулируйте алгоритм с полиномиальным временем работы, позволяющий решить задачу принятия решения в предположении, что все времена выполнения выражаются целыми числами от 1 до n. (*Указание*: воспользуйтесь методами динамического программирования.)

г) Сформулируйте алгоритм с полиномиальным временем работы, позволяющий решить задачу оптимизации в предположении, что все времена выполнения выражаются целыми числами от 1 до n.

Заключительные замечания

Книга Гарея (Garey) и Джонсона (Johnson) [110] является замечательным учебником по вопросам NP-полноты; в ней подробно обсуждается эта теория и описываются многие задачи, NP-полнота которых была доказана до 1979 года. Из этой книги позаимствовано доказательство теоремы 34.13, а список NP-полных задач, описанных в разделе 34.5, взят из содержания этой книги. В период 1981-1992 гг. Джонсон написал серию из 23 статей в *Journal of Algorithms*, рассказывая о новых достижениях в исследованиях NP-полноты. В книгах Хопкрофта (Hopcroft), Мотвани (Motwani) и Ульмана (Ullman) [153], Льюиса (Lewis) и Пападимитриу (Papadimitriou) [204], Пападимитриу [236] и Сипсера (Sipser) [279] проблема NP-полноты удачно трактуется в контексте теории сложности. В книге Ахо (Aho), Хопкрофта и Ульмана [5] также описывается NP-полнота и приводятся примеры приведений, в том числе приведение задачи о гамильтоновом цикле к задаче о вершинном покрытии.

Класс P был введен в 1964 году Кобхемом (Cobham) [64] и независимо в 1965 году Эдмондсом (Edmonds) [84], который ввел также класс NP и выдвинул гипотезу, что $\mathrm{P} \neq \mathrm{NP}$. Понятие NP-полноты впервые было предложено Куком (Cook) [67] в 1971 году. Кук представил первое доказательство NP-полноты задач о выполнимости формул и 3-CNF выполнимости. Левин (Levin) [203] независимо пришел к этому понятию; он доказал NP-полноту задачи о мозаике. Карп (Karp) [173] в 1972 году предложил методику приведения и продемонстрировал богатое разнообразие NP-полных задач. В статье Крапа впервые доказывается NP-полнота задач о клике, о вершинном покрытии и о гамильтоновом цикле. С тех пор многими исследователями была доказана NP-полнота сотен задач. В 1995 году

на заседании, посвященном 60-летию Карпа, Пападимитриу в своем докладе заметил: “. . . каждый год выходит около 6000 статей, в заголовке, аннотации или списке ключевых слов которых содержится термин ‘NP-полный’. Он встречается чаще, чем термины ‘компилятор’, ‘база данных’, ‘экспертная система’, ‘нейронная сеть’ или ‘операционная система”’.

Недавняя работа по теории сложности пролила свет на вопрос о сложности приближенных компьютерных вычислений. В ней приводится новое определение класса NP с помощью “вероятностно проверяемых доказательств”. В этом определении подразумевается, что для таких задач, как задача о клике, о вершинном покрытии, задача о коммивояжере, в которой выполняется неравенство треугольника, и многих других получение хорошего приближенного решения является NP-сложным, поэтому оно не легче, чем получение оптимальных решений. Введение в эту область можно найти в диссертации Ароры (Arora) [19], в главе Ароры и Лунда (Lund) в [149], в обзорной статье Ароры [20], в книге под редакцией Мэйра (Mayr), Промеля (Promel) и Стиджера (Steger) [214], а также в обзорной статье Джонсона [167].

ГЛАВА 35

Приближенные алгоритмы

Многие задачи, представляющие практический интерес, являются NP-полными. Однако они слишком важны, чтобы отказаться от их решения лишь на том основании, что получить их оптимальное решение трудно. Если задача NP-полная, мало шансов найти алгоритм с полиномиальным временем работы, позволяющий найти ее точное решение. Несмотря на это надежда на точное решение остается. Во-первых, если объем входных данных небольшой, алгоритм, время работы которого выражается показательной функцией, вполне может подойти. Во-вторых, иногда удается выделить важные частные случаи, разрешимые в течение полиномиального времени. В-третьих, остается возможность найти в течение полиномиального времени решение, *близкое к оптимальному* (в наихудшем либо в среднем случае). На практике такие решения часто являются достаточно хорошими. Алгоритм, возвращающий решения, близкие к оптимальным, называется ***приближенным алгоритмом*** (approximation algorithm). В этой главе описаны приближенные алгоритмы с полиномиальным временем выполнения, предназначенные для решения некоторых NP-полных задач.

Оценка качества приближенных алгоритмов

Предположим, мы занимаемся задачей оптимизации, каждому из возможных решений которой сопоставляется положительная стоимость, и требуется найти решение, близкое к оптимальному. В зависимости от задачи, оптимальное решение можно определить либо как такое, которому соответствует максимально возможная стоимость, или такое, которому соответствует минимально возможная стоимость; другими словами, это может быть либо задача максимизации, либо задача минимизации.

Говорят, что алгоритм решения задачи обладает ***отношением, или коэффициентом аппроксимации (приближения)*** (approximation ratio) $\rho(n)$, если для произвольных входных данных размера n стоимость C решения, полученного в результате выполнения этого алгоритма, отличается от стоимости C^* оптимального решения не более чем в $\rho(n)$ раз:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leqslant \rho(n). \tag{35.1}$$

Алгоритм, в котором достигается коэффициент аппроксимации $\rho(n)$, будем называть $\rho(n)$-***приближенным алгоритмом*** ($\rho(n)$-approximation algorithm). Определения коэффициента аппроксимации и $\rho(n)$-приближенного алгоритма применимы и к задачам минимизации, и к задачам максимизации. Для задач максимизации выполняется неравенство $0 < C \leqslant C^*$, и отношение C^*/C равно величине, на которую стоимость оптимального решения больше стоимости приближенного решения. Аналогично, для задач минимизации выполняется неравенство $0 < C^* \leqslant C$, и отношение C/C^* дает значение, во сколько раз стоимость приближенного решения больше стоимости оптимального решения. Поскольку предполагается, что стоимости всех решений положительные, эти коэффициенты вполне определены. Коэффициент аппроксимации приближенного алгоритма не может быть меньше 1, поскольку из неравенства $C/C^* < 1$ следует неравенство $C^*/C > 1$. Таким образом, 1-приближенный алгоритм[1] выдает оптимальное решение, а приближенный алгоритм с бóльшим отношением аппроксимации может возвратить решение, которое намного хуже оптимального.

Для ряда задач разработаны приближенные алгоритмы с полиномиальным временем работы и малыми постоянными отношениями аппроксимации. Есть также задачи, для которых лучшие из известных приближенных алгоритмов с полиномиальным временем работы характеризуются коэффициентами аппроксимации, величина которых возрастает с ростом размера входных данных n. Примером такой задачи является задача о покрытии множества, представленная в разделе 35.3.

Некоторые NP-полные задачи допускают наличие приближенных алгоритмов с полиномиальным временем работы, коэффициент аппроксимации которых можно уменьшать за счет увеличения времени их работы. Другими словами, в них допускается компромисс между временем вычисления и качеством приближения. В качестве примера можно привести задачу о сумме подмножества, которая исследуется в разделе 35.5. Эта ситуация достаточно важна и заслуживает собственного имени.

Схема аппроксимации (approximation scheme) задачи оптимизации — это приближенный алгоритм, входные данные которого включают в себя не только пара-

[1] Если коэффициент аппроксимации не зависит от n, мы используем термины "отношение аппроксимации ρ" и "ρ-приближенный алгоритм", указывающие на отсутствие зависимости от n.

метры экземпляра задачи, но и такое значение $\varepsilon > 0$, что для любого фиксированного значения ε эта схема является $(1+\varepsilon)$-приближенным алгоритмом. Схему аппроксимации называют ***схемой аппроксимации с полиномиальным временем выполнения*** (polynomial-time approximation scheme), если для любого фиксированного значения $\varepsilon > 0$ работа этой схемы завершается в течение времени, выраженного полиномиальной функцией от размера n входных данных.

Время работы схемы аппроксимации с полиномиальным временем вычисления может очень быстро возрастать при уменьшении величины ε. Например, это время может вести себя как $O\left(n^{2/\varepsilon}\right)$. В идеале, если величина ε уменьшается на постоянный множитель, время, необходимое для достижения нужного приближения, не должно возрастать больше, чем на постоянный множитель. Другими словами, хотелось бы, чтобы время работы схемы выражалось полиномиальной функцией от величин $1/\varepsilon$ и n.

Говорят, что схема аппроксимации является ***схемой аппроксимации с полностью полиномиальным временем работы*** (fully polynomial-time approximation scheme), если время ее работы выражается полиномом от $1/\varepsilon$ и размера входных данных задачи n. Например, время работы такой схемы может вести себя как $O\left((1/\varepsilon)^2 n^3\right)$. В такой схеме любого уменьшения величины ε на постоянный множитель можно добиться за счет увеличения времени работы на соответствующий постоянный множитель.

Краткое содержание главы

В первых четырех разделах этой главы приведены некоторые примеры приближенных алгоритмов с полиномиальным временем работы, позволяющие получать приближенные решения NP-полных задач. В пятом разделе представлена схема аппроксимации с полностью полиномиальным временем работы. Начало раздела 35.1 посвящено исследованию задачи о вершинном покрытии, которая относится к классу NP-полных задач минимизации. Для этой задачи существует приближенный алгоритм, характеризующийся коэффициентом аппроксимации 2. В разделе 35.2 представлен приближенный алгоритм с коэффициентом аппроксимации 2, предназначенный для решения частного случая задачи коммивояжера, когда функция стоимости удовлетворяет неравенству треугольника. Также показано, что если неравенство треугольника не соблюдается, то для любой константы $\rho \geqslant 1$ ρ-приближенного алгоритма не существует, если не выполняется условие $\mathrm{P} = \mathrm{NP}$. В разделе 35.3 показано, как использовать жадный метод в качестве эффективного приближенного алгоритма для решения задачи о покрытии множества. При этом возвращается покрытие, стоимость которого в наихудшем случае превышает оптимальную на множитель, выражающийся логарифмической функцией. В разделе 35.4 представлены еще два приближенного алгоритма. В первом из них исследуется оптимизирующая версия задачи о 3-CNF выполнимости

и приводится простой рандомизированный алгоритм, который выдает решение, характеризующееся математическим ожиданием коэффициента аппроксимации, равным $8/7$. Затем изучается взвешенный вариант задачи о вершинном покрытии и описывается, как с помощью методов линейного программирования разработать 2-приближенный алгоритм. Наконец, в разделе 35.5 представлена схема аппроксимации с полностью полиномиальным временем выполнения, предназначенная для решения задачи о сумме подмножеств.

35.1 Задача о вершинном покрытии

Задача о вершинном покрытии определена в разделе 34.5.2. В этом же разделе доказано, что эта задача является NP-полной. ***Вершинное покрытие*** (vertex cover) неориентированного графа $G = (V, E)$ — это такое подмножество $V' \subseteq V$, что если (u, v) — ребро графа G, то либо $u \in V'$, либо $v \in V'$ (могут выполняться и оба эти соотношения). Размером вершинного покрытия называется количество содержащихся в нем вершин.

Задача о вершинном покрытии (vertex-cover problem) состоит в том, чтобы найти для заданного неориентированного графа вершинное покрытие минимального размера. Назовем такое вершинное покрытие ***оптимальным вершинным покрытием*** (optimal vertex cover). Эта задача представляет собой оптимизирующую версию NP-полной задачи принятия решений.

Несмотря на то, что найти оптимальное вершинное покрытие графа G может оказаться трудно, не так сложно найти вершинное покрытие, близкое к оптимальному. Приведенный ниже приближенный алгоритм принимает в качестве входных данных параметры неориентированного графа G и возвращает вершинное покрытие, размер которого превышает размер оптимального вершинного покрытия не более чем в два раза.

```
APPROX_VERTEX_COVER(G)
1  C ← ∅
2  E' ← E[G]
3  while E' ≠ ∅
4        do Пусть (u, v) — произвольное ребро из множества E'
5           C ← C ∪ {u, v}
6           Удаляем из множества E' все ребра, инцидентные
                  вершинам u или v
7  return C
```

Рассмотрим работу алгоритма APPROX_VERTEX_COVER. Как уже было сказано, вершинное покрытие, которое конструируется, содержится в переменной C. В строке 1 переменная C инициализируется пустым множеством. В строке 2

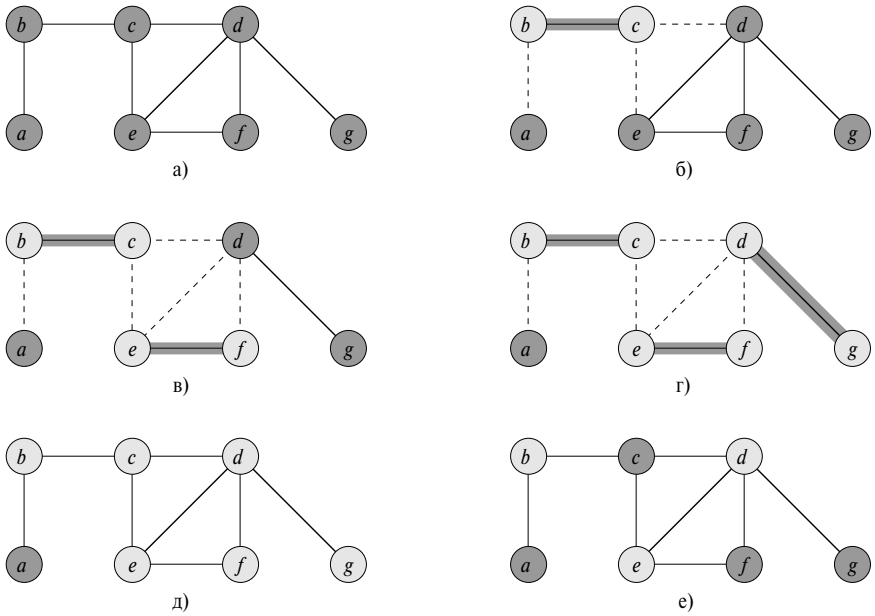


Рис. 35.1. Работа алгоритма APPROX_VERTEX_COVER

создается множество E', представляющее собой копию множества ребер графа $E[G]$. Цикл в строках 3-6 выбирает из множества E' ребро (u, v), добавляет его конечные точки u и v в множество C и удаляет из множества E' все ребра, которые покрываются вершиной u или вершиной v. Если для представления множества E' используются списки смежных вершин, время выполнения этого алгоритма равно $O(V + E)$.

Работа алгоритма APPROX_VERTEX_COVER проиллюстрирована на рис. 35.1. В части *а* рисунка изображен граф G, содержащий 7 вершин и 8 ребер. В части *б* ребро (b, c), выделенное серым цветом, — первое по счету ребро, выбранное алгоритмом APPROX_VERTEX_COVER. Вершины b и c, показанные светло-серой штриховкой, добавляются в множество C, в котором содержится создаваемое вершинное покрытие. Показанные пунктиром ребра (a, b), (c, e) и (c, d) удаляются, поскольку они уже покрыты вершинами из множества C. В части *в* рисунка выбрано ребро (e, f), а вершины e и f добавляются в множество C. В части *г* выбрано ребро (d, g), а вершины d и g добавляются в множество C. В части *д* показано множество C, которое представляет собой вершинное покрытие, полученное в результате выполнения алгоритма APPROX_VERTEX_COVER. Как видно из рисунка, оно состоит из шести вершин: b, c, d, e, f, g. В части *е* рисунка показано оптимальное вершинное покрытие для рассмотренного экземпляра задачи. Оно состоит всего из трех вершин: b, d и e.

Теорема 35.1. Алгоритм APPROX_VERTEX_COVER является 2-приближенным алгоритмом с полиномиальным временем работы.

Доказательство. Мы уже показали, что время работы алгоритма APPROX_VERTEX_COVER выражается полиномиальной функцией.

Множество вершин C, которое возвращается алгоритмом APPROX_VERTEX_COVER, является вершинным покрытием, поскольку алгоритм не выходит из цикла, пока каждое ребро $E[G]$ не будет покрыто некоторой вершиной из множества C.

Чтобы показать, что рассматриваемый алгоритм возвращает вершинное покрытие, размер которого превышает размер оптимального вершинного покрытия не более чем в два раза, обозначим через A множество ребер, выбранных в строке 4 алгоритма APPROX_VERTEX_COVER. Чтобы покрыть ребра множества A, каждое вершинное покрытие, в том числе и оптимальное покрытие C^*, должно содержать хотя бы одну конечную точку каждого ребра из множества A. Никакие два ребра из этого множества не имеют общих конечных точек, поскольку после того, как ребро выбирается в строке 4, все другие ребра с такими же конечными точками удаляются из множества E' в строке 6. Таким образом, никакие два ребра из множества A не покрываются одной и той же вершиной из множества C^*, из чего следует, что нижняя граница размера оптимального вершинного покрытия равна

$$|C^*| \geqslant |A| . \tag{35.2}$$

При каждом выполнении строки 4 выбирается ребро, ни одна из конечных точек которого пока еще не вошла в множество C. Это позволяет оценить сверху (фактически указать точную верхнюю границу) размера возвращаемого вершинного покрытия:

$$|C| = 2\,|A| . \tag{35.3}$$

Сопоставляя уравнения (35.2) и (35.3), получаем

$$|C| = 2\,|A| \leqslant 2\,|C^*| ,$$

что и доказывает теорему. ■

Еще раз вернемся к приведенному выше доказательству. На первый взгляд может показаться удивительным, как можно доказать, что размер вершинного покрытия, возвращенного процедурой APPROX_VERTEX_COVER, не более чем в два раза превышает размер оптимального вершинного покрытия, если не известно, чему равен размер оптимального вершинного покрытия. Это становится возможным благодаря использованию нижней границы оптимального вершинного покрытия. Как предлагается показать в упражнении 35.1-2, множество A, состоящее из ребер, выбранных в строке 4 процедуры APPROX_VERTEX_COVER, фактически является

максимальным паросочетанием вершин в графе G. (***Максимальное паросочетание*** (maximal matching) — это паросочетание, которое не является собственным подмножеством какого-нибудь другого паросочетания.) Как было показано при доказательстве теоремы 35.1, размер максимального паросочетания равен нижней границе размера оптимального вершинного покрытия. Алгоритм возвращает вершинное покрытие, размер которого не более чем в два раза превышает размер максимального паросочетания множества A. Составив отношение размера возвращаемого решения к полученной нижней границе, находим коэффициент аппроксимации. Эта методика будет использоваться и в следующих разделах.

Упражнения

35.1-1. Приведите пример графа, для которого процедура APPROX_VERTEX_COVER всегда возвращает неоптимальное решение.

35.1-2. Обозначим через A множество ребер, выбранных в строке 4 процедуры APPROX_VERTEX_COVER. Докажите, что множество A — максимальное паросочетание в графе G.

⋆ 35.1-3. Профессор предложил такую эвристическую схему решения задачи о вершинном покрытии. Одна за другой выбираются вершины с максимальной степенью, и для каждой из них удаляются все инцидентные ребра. Приведите пример, демонстрирующий, что коэффициент аппроксимации, предложенной профессором, превышает 2. (*Указание*: попытайтесь построить двудольный граф, вершины в левой части которого имеют одинаковые степени, а вершины в правой части — разные степени.)

35.1-4. Сформулируйте эффективный жадный алгоритм, позволяющий найти оптимальное вершинное покрытие дерева в течение линейного времени.

35.1-5. Из доказательства теоремы 34.12 известно, что задача о вершинном покрытии и NP-полная задача о клике являются взаимодополняющими в том смысле, что оптимальное вершинное покрытие — это дополнение к клику максимального размера в дополняющем графе. Следует ли из этого, что для задачи о клике существует приближенный алгоритм с полиномиальным временем решения, обладающий постоянным коэффициентом аппроксимации? Обоснуйте ваш ответ.

35.2 Задача о коммивояжере

Обратимся к задаче о коммивояжере, представленной в разделе 34.5.4. В этой задаче задается полный неориентированный граф $G = (V, E)$, каждому из ребер $(u, v) \in E$ которого сопоставляется неотрицательная целочисленная стоимость $c(u, v)$, и необходимо найти гамильтонов цикл (тур) минимальной стоимости

графа G. Введем дополнительное обозначение $c(A)$ — полную стоимость всех ребер подмножества $A \subseteq E$:

$$c(A) = \sum_{(u,v) \in A} c(u,v).$$

Во многих практических ситуациях наиболее дешевый переход из места u в место w — всегда по прямой, так что если по пути зайти в какой-нибудь промежуточный путь v, это не может привести к уменьшению стоимости. Выражаясь другими словами, если срезать путь, пропустив какой-нибудь промежуточный пункт, — это никогда не станет причиной увеличения стоимости. Формализуем это понятие, выдвинув утверждение, что функция стоимости c удовлетворяет ***неравенству треугольника*** (triangle inequality), если для всех вершин $u, v, w \in V$

$$c(u,w) \leqslant c(u,v) + c(v,w).$$

Неравенство треугольника имеет естественный характер и автоматически удовлетворяется во многих приложениях. Например, если вершины графа — точки на плоскости, и стоимость перехода от одной вершины к другой выражается обычным евклидовым расстоянием между ними, то неравенство треугольника выполняется. (Кроме евклидового расстояния существует множество других функций стоимости, удовлетворяющих неравенству треугольника.)

Как видно из упражнения 35.2-2, задача о коммивояжере является NP-полной даже в том случае, если потребовать, чтобы функция стоимости удовлетворяла неравенству треугольника. Таким образом, мало надежд найти алгоритм с полиномиальным временем работы, позволяющий получить точное решение этой задачи. Поэтому есть смысл заняться поиском хороших приближенных алгоритмов.

В разделе 35.2.1 исследуется 2-приближенный алгоритм, позволяющий решить задачу о коммивояжере, в которой выполняется неравенство треугольника. В разделе 35.2.2 будет показано, что без соблюдения неравенства треугольника приближенный алгоритм с полиномиальным временем работы, характеризующийся постоянным коэффициентом аппроксимации, не существует, если только не справедливо соотношение $\mathrm{P} = \mathrm{NP}$.

35.2.1 Задача о коммивояжере с неравенством треугольника

Воспользовавшись методикой предыдущего раздела, сначала вычислим минимальное остовное дерево, вес которого является нижней границей длины оптимального тура коммивояжера. Затем с помощью этого минимального остовного дерева создадим тур, стоимость которого не более чем в два раза превышает вес этого дерева при условии, что функция стоимости удовлетворяет неравенству

треугольника. Этот подход реализован в приведенном ниже алгоритме, в котором используется алгоритм построения минимального остовного дерева MST_PRIM, описанный в разделе 23.2.

APPROX_TSP_TOUR(G, c)

1 Выбирается вершина $r \in V[G]$ которая будет "корневой"
2 Из корня r с помощью алгоритма MST-PRIM(G, c, r) строится минимальное остовное дерево T для графа G
3 Пусть L — список вершин, которые посещаются при обходе вершин дерева T в прямом порядке
4 **return** Гамильтонов цикл H, который посещает вершины в порядке перечисления в списке L

Напомним, что при прямом обходе дерева рекурсивно посещаются все его вершины (см. раздел 12.1), причем вершина заносится в список при первом посещении, до посещения ее дочерних вершин.

Работа процедуры APPROX_TSP_TOUR проиллюстрирована на рис. 35.2. В части *а* рисунка показано заданное множество точек, расположенных в вершинах целочисленной решетки. Например, точка f находится на один шаг правее и на два шага выше от точки h. В качестве функции стоимости между двумя точками используется обычное евклидово расстояние. В части *б* рисунка изображено полученное в результате выполнения алгоритма MST_PRIM минимальное остовное дерево T, берущее свое начало в корневой вершине a. Вершинам присвоены метки таким образом, что они добавляются алгоритмом MST_PRIM в основное дерево в алфавитном порядке. В части *в* показан порядок посещения вершин при прямом обходе дерева T, начиная с вершины a. При полном обходе дерева вершины посещаются в порядке a, b, c, b, h, b, a, d, e, f, e, g, e, d, a. При прямом обходе дерева T составляется список вершин, посещенных впервые (на рисунке возле каждой такой вершины поставлена точка). Полученный в результате список имеет вид a, b, c, h, d, e, f, g. В части *г* рисунка представлен тур H, возвращенный алгоритмом APPROX_TSP_TOUR. Его полная стоимость составляет приблизительно 19.074. В части *д* рисунка изображен оптимальный тур H^*, длина которого примерно на 23% меньше (она приблизительно равна 14.715).

Согласно результатам упражнения 23.2-2, даже при простой реализации алгоритма MST_PRIM время работы алгоритма APPROX_TSP_TOUR равно $\Theta\left(V^2\right)$. Теперь покажем, что если функция стоимости в экземпляре задачи коммивояжера удовлетворяет неравенству треугольника, то алгоритм APPROX_TSP_TOUR возвращает тур, стоимость которого не более чем в два раза превышает стоимость оптимального тура.

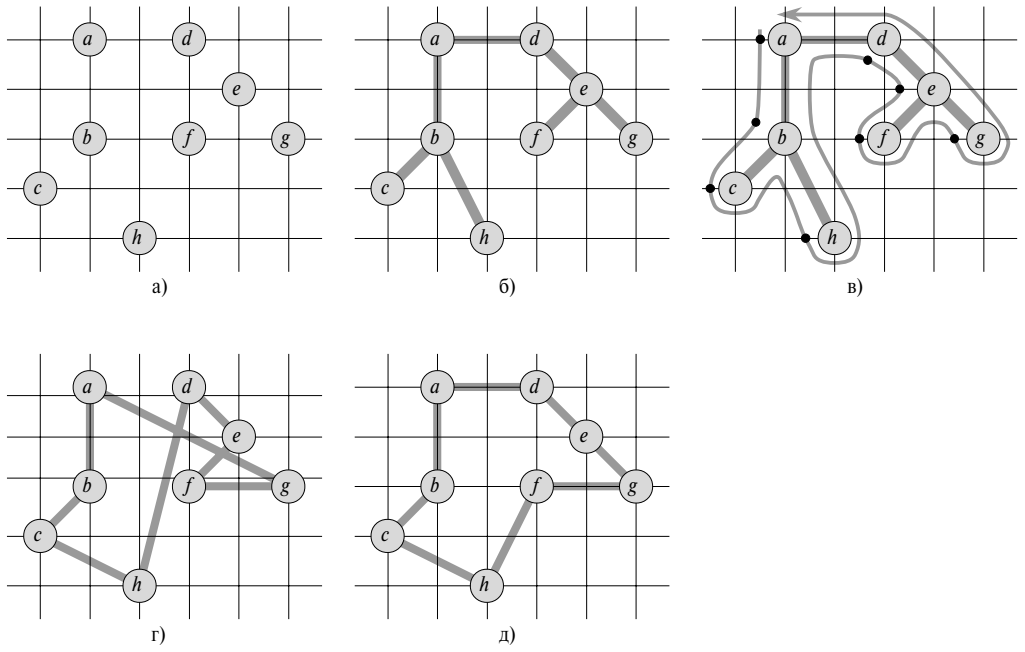


Рис. 35.2. Работа алгоритма APPROX_TSP_TOUR

Теорема 35.2. Алгоритм APPROX_TSP_TOUR является 2-приближенным алгоритмом с полиномиальным временем работы, позволяющим решить задачу коммивояжера, в которой удовлетворяется неравенство треугольника.

Доказательство. Ранее уже было показано, что время работы алгоритма APPROX_TSP_TOUR выражается полиномиальной функцией.

Обозначим через H^* тур, который является оптимальным для данного множества вершин. Поскольку путем удаления из этого тура одного ребра получается остовное дерево, вес минимального остовного дерева T равен нижней границе стоимости оптимального тура, т.е. выполняется неравенство

$$c(T) \leqslant c(H^*). \tag{35.4}$$

При ***полном обходе*** (full walk) дерева T составляется список вершин, которые посещаются впервые, если к ним происходит возврат после посещения поддерева. Обозначим этот обход через W. При полном обходе в рассматриваемом примере вершины посещаются в следующем порядке:

$$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a.$$

Поскольку при полном обходе каждое ребро дерева T проходится ровно по два раза, естественным образом обобщив определение стоимости c на множества,

в которых ребра встречаются по несколько раз, получаем равенство

$$c(W) = 2c(T). \tag{35.5}$$

Из соотношений (35.4) и (35.5) следует неравенство

$$c(W) \leqslant 2c(H^*). \tag{35.6}$$

Таким образом, стоимость обхода W превышает стоимость оптимального тура не более чем в два раза.

К сожалению, обход W в общем случае не является туром, поскольку он посещает некоторые вершины более одного раза. Однако, согласно неравенству треугольника, посещение любой из вершин в обходе W можно отменить, и при этом стоимость не возрастет. (Если из маршрута W удалить вершину v, которая посещается в этом маршруте на пути от вершины u к вершине w, то в полученном в результате такой операции упорядоченном списке вершин будет определяться переход непосредственно от вершины u к вершине w.) Путем неоднократного выполнения этой операции из обхода W можно исключить все посещения каждой вершины, кроме первого. В рассматриваемом примере упорядоченный список вершин принимает вид

$$a, b, c, h, d, e, f, g.$$

Этот порядок совпадает с тем, который получается при прямом обходе дерева T. Пусть H — цикл, соответствующий данному прямому обходу. Это гамильтонов цикл, так как каждая вершина посещается по одному разу. Именно этот цикл составляется алгоритмом APPROX_TSP_TOUR. Поскольку цикл H получается путем удаления вершин из полного обхода W, выполняется неравенство

$$c(H) \leqslant c(W). \tag{35.7}$$

Сопоставляя неравенства (35.6) и (35.7), получаем соотношение $c(H) \leqslant 2c(H^*)$, что и завершает доказательство теоремы. ■

Несмотря на то, что теорема 35.2 позволяет добиться неплохого коэффициента аппроксимации, обычно на практике алгоритм APPROX_TSP_TOUR — не лучший выбор для решения этой задачи. Существует другой приближенный алгоритм, который обычно дает намного лучшие практические результаты (см. ссылки в конце этой главы).

35.2.2 Общая задача о коммивояжере

Если опустить предположение о том, что функция стоимости c удовлетворяет неравенству треугольника, то нельзя найти туры с хорошим приближением за полиномиальное время, если не выполняется условие $\mathrm{P} = \mathrm{NP}$.

Теорема 35.3. Если $\mathrm{P} \neq \mathrm{NP}$, то для любой константы $\rho \geqslant 1$ не существует приближенного алгоритма с полиномиальным временем работы и коэффициентом аппроксимации ρ, позволяющего решить задачу о коммивояжере в общем виде.

Доказательство. Докажем теорему методом "от противного". Предположим, что для некоторого числа $\rho \geqslant 1$ существует приближенный алгоритм A с полиномиальным временем работы и коэффициентом аппроксимации ρ. Без потери общности предположим, что число ρ — целое, округлив его при необходимости. Затем покажем, как с помощью алгоритма A можно решать экземпляры задачи о гамильтоновом цикле (определенной в разделе 34.2) в течение полиномиального времени. Поскольку задача о гамильтоновом цикле NP-полная, согласно теореме 34.13, из ее разрешимости в течение полиномиального времени и теоремы 34.4 следует равенство $\mathrm{P} = \mathrm{NP}$.

Пусть $G = (V, E)$ — экземпляр задачи о гамильтоновом цикле. Необходим эффективный способ, позволяющий с помощью гипотетического приближенного алгоритма A определить, содержит ли граф G гамильтонов цикл. Преобразуем граф G в экземпляр задачи о коммивояжере. Пусть $G' = (V, E')$ — полный граф на множестве V, т.е.

$$E' = \{(u, v) : u, v \in V \text{ и } u \neq v\}.$$

Назначим каждому ребру из множества E' целочисленную стоимость

$$c(u, v) = \begin{cases} 1 & \text{если } (u, v) \in E, \\ \rho|V| + 1 & \text{в противном случае.} \end{cases}$$

Представление графа G' и функции c можно получить из представления графа G в течение времени, являющегося полиномиальной функцией от величин $|V|$ и $|E|$.

Теперь рассмотрим задачу о коммивояжере (G', c). Если исходный граф G содержит гамильтонов цикл H, то функция стоимости c сопоставляет каждому ребру цикла H единичную стоимость, а значит, экземпляр (G', c) содержит тур стоимостью $|V|$. С другой стороны, если граф G не содержит гамильтонового цикла, то в любом туре по графу G' должно использоваться некоторое ребро, отсутствующее в множестве E. Однако стоимость любого тура, в котором используется ребро, не содержащееся в множестве E, не меньше величины

$$(\rho|V| + 1) + (|V| + 1) = \rho|V| + |V| > \rho|V|.$$

Из-за большой стоимости ребер, отсутствующих в графе G, между стоимостью тура, который представляет собой гамильтонов цикл в графе G (она равна $|V|$), и стоимостью любого другого тура (его величина не меньше $\rho|V| + |V|$) существует интервал, величина которого не меньше $\rho|V|$.

Что произойдет, если применить к задаче о коммивояжере (G', c) алгоритм A? Поскольку этот алгоритм гарантированно возвращает тур, стоимость которого не более чем в ρ раз превышает стоимость оптимального тура, если граф G содержит гамильтонов цикл, алгоритм A должен его возвратить. Если же граф G не содержит гамильтоновых циклов, то алгоритм A возвращает тур, стоимость которого превышает величину $\rho|V|$. Поэтому с помощью алгоритма A задачу о гамильтоновом цикле можно решить в течение полиномиального времени. ■

Доказательство теоремы 35.3 — это пример общей методики доказательства того, что задачу нельзя хорошо аппроксимировать. Предположим, что заданной NP-сложной задаче X в течение полиномиального времени можно сопоставить такую задачу минимизации Y, что "да"-экземпляры задачи X будут соответствовать экземплярам задачи Y, стоимость которых не превышает k (где k — некоторая фиксированная величина), а "нет"-экземпляры задачи X будут соответствовать экземплярам задачи Y, стоимость которых превышает ρk. Затем нужно показать, что если не выполняется равенство $\mathrm{P} = \mathrm{NP}$, то не существует ρ-приближенного алгоритма с полиномиальным временем работы, позволяющего решить задачу Y.

Упражнения

35.2-1. Предположим, что полный неориентированный граф $G = (V, E)$, содержащий не менее трех вершин, характеризуется функцией стоимости c, удовлетворяющей неравенству треугольника. Докажите, что для всех $u, v \in V$ выполняется неравенство $c(u, v) \geqslant 0$.

35.2-2. Покажите, как в течение полиномиального времени один экземпляр задачи о коммивояжере можно преобразовать в другой экземпляр, функция стоимости которого удовлетворяет неравенству треугольника. Оба экземпляра должны содержать одно и то же множество оптимальных туров. Объясните, почему такое полиномиально-временное преобразование не противоречит теореме 35.3, предполагая, что $\mathrm{P} \neq \mathrm{NP}$.

35.2-3. Рассмотрим описанный ниже ***эвристический метод ближайшей точки*** (closest-point heuristic), позволяющий создавать приближенные туры в задаче о коммивояжере, функция стоимости которой удовлетворяет неравенству треугольника. Начнем построение с тривиального цикла, состоящего из одной произвольным образом выбранной вершины. На каждом этапе идентифицируется вершина u, не принадлежащая циклу, причем такая, расстояние от которой до цикла является минимальным. Предположим, что ближе всех к вершине u в цикле расположена вершина v. Цикл расширяется за счет включения в него вершины u сразу после вершины v. Описанные действия повторяются до тех пор, пока в цикл

не будут включены все вершины. Докажите, что этот эвристический метод возвращает тур, полная стоимость которого не более чем в два раза превышает полную стоимость оптимального тура.

35.2-4. ***Задача о коммивояжере с устранением узких мест*** (bottleneck travelling-salesman problem) — это задача поиска такого гамильтонового цикла, для которого минимизируется стоимость самого дорогостоящего входящего в этот цикл ребра. Предполагая, что функция стоимости удовлетворяет неравенству треугольника, покажите, что для этой задачи существует приближенный алгоритм с полиномиальным временем работы, коэффициент аппроксимации которого равен 3. (*Указание*: воспользовавшись рекурсией, покажите, что все узлы остовного дерева с узкими местами можно обойти ровно по одному разу, взяв полный обход дерева с пропусками узлов, если ограничить пропуски таким образом, чтобы не пропускать более двух последовательных промежуточных узлов (см. упражнение 23.-3). Покажите, что стоимость самого дорогостоящего ребра в остовном дереве с узкими местами не превышает стоимости самого дорогостоящего ребра в гамильтоновом цикле с узкими местами.)

35.2-5. Предположим, что вершины экземпляра задачи о коммивояжере расположены на плоскости, и что стоимость $c(u, v)$ равна евклидовому расстоянию между точками u и v. Покажите, что в оптимальном туре никогда не будет самопересечений.

35.3 Задача о покрытии множества

Задача о покрытии множества — это задача оптимизации, моделирующая многие задачи выбора ресурсов (resource-selection problems). Соответствующая ей задача принятия решения является обобщением NP-полной задачи о вершинном покрытии, следовательно, она — NP-сложная. Однако приближенный алгоритм, разработанный для задачи о вершинном покрытии, здесь неприменим, поэтому следует попытаться поискать другие подходы. Исследуем простой эвристический жадный метод, коэффициент аппроксимации которого выражается логарифмической функцией. Другими словами, по мере того как растет размер экземпляра задачи, размер приближенного решения тоже может возрастать относительно размера оптимального решения. Однако, так как логарифмическая функция возрастает достаточно медленно, этот приближенный алгоритм может тем не менее давать полезные результаты.

Экземпляр $(X, \mathcal{F})$ ***задачи о покрытии множества*** (set-covering problem) состоит из конечного множества X и такого семейства $\mathcal{F}$ подмножеств множества X, что каждый элемент множества X принадлежит хотя бы одному подмножеству

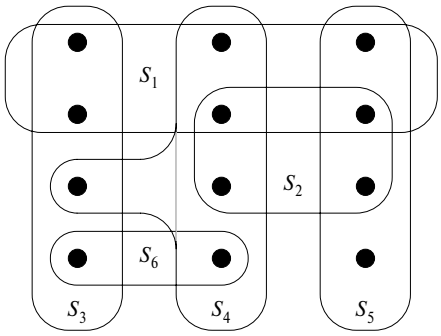


Рис. 35.3. Экземпляр $(X, \mathcal{F})$ задачи о покрытии множества, в котором множество X состоит из 12 черных точек, а $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$

из семейства $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S.$$

Говорят, что подмножество $S \in \mathcal{F}$ ***покрывает*** (covers) содержащиеся в нем элементы. Задача состоит в том, чтобы найти подмножество $\mathcal{C} \subseteq \mathcal{F}$ минимального размера, члены которого покрывают все множество X:

$$X = \bigcup_{S \in \mathcal{C}} S. \tag{35.8}$$

Говорят, что любое семейство $\mathcal{C}$, удовлетворяющее уравнению (35.8), ***покрывает*** (covers) множество X. Задача о покрытии множества иллюстрируется на рис. 35.3. Размер семейства $\mathcal{C}$ определяется как количество содержащихся в нем подмножеств, а не как суммарное количество отдельных элементов в этих множествах. В примере, проиллюстрированном на рис. 35.3, размер минимального покрытия множества равен 3. Жадный алгоритм возвращает вершинное покрытие, размер которого равен 4, выбирая множества S_1, S_4, S_5 и S_3 в том порядке, в котором они здесь перечислены. Покрытие множества минимального размера имеет вид $\mathcal{C} = \{S_3, S_4, S_5\}$.

Задача о покрытии множества является абстракцией многих часто возникающих комбинаторных задач. В качестве простого примера предположим, что множество X представляет набор знаний, необходимых для решения задачи, над которой работает определенный коллектив сотрудников. Нужно сформировать комитет, состоящий из минимально возможного количества сотрудников, причем такой, что при необходимости любой информации из множества X, оказывается, что в комитете есть сотрудник, обладающий этим знанием. Если преобразовать эту

задачу в задачу принятия решений, то в ней будет спрашиваться, существует ли покрытие, размер которого не превышает k, где k — дополнительный параметр, определенный в экземпляре задачи. Как предлагается показать в задаче 35.3-2, версия этой задачи в форме задачи принятия решений является NP-полной.

Жадный приближенный алгоритм

Жадный метод работает путем выбора на каждом этапе множества S, покрывающего максимальное количество элементов, оставшихся непокрытыми.

```
GREEDY_SET_COVER(X, F)
1  U ← X
2  C ← ∅
3  while U ≠ ∅
4        do выбирается подмножество S ∈ F, максимизирующее
                  величину |S ∩ U|
5           U ← U − S
6           C ← C ∪ {S}
7  return C
```

Опишем принцип действия алгоритма. На каждом этапе его работы множество U содержит оставшиеся непокрытыми элементы. Множество $\mathcal{C}$ содержит покрытие, которое конструируется. Строка 4 — это этап принятия решения в жадном методе. Выбирается подмножество S, покрывающее максимально возможное количество еще непокрытых элементов (с произвольным разрывом связей). После выбора подмножества S его элементы удаляются из множества U, а подмножество S помещается в семейство $\mathcal{C}$. Когда алгоритм завершает свою работу, множество $\mathcal{C}$ будет содержать подсемейство семейства $\mathcal{F}$, покрывающее множество X.

В примере, проиллюстрированном на рис. 35.3, алгоритм GREEDY_SET_COVER добавляет в семейство $\mathcal{C}$ множества S_1, S_4, S_5 и S_3 в порядке перечисления.

Алгоритм GREEDY_SET_COVER легко реализовать таким образом, чтобы время его работы выражалось полиномиальной функцией от величин $|X|$ и $|\mathcal{F}|$. Поскольку количество итераций цикла в строках 3-6 ограничено сверху величиной $\min(|X|, |\mathcal{F}|)$, а тело цикла можно реализовать таким образом, чтобы его выполнение завершалось в течение времени $O(|X|\,|\mathcal{F}|)$, существует реализация, завершающая свою работу в течение времени $O(|X|\,|\mathcal{F}|\min(|X|, |\mathcal{F}|))$. В упражнении 35.3-3 предлагается разработать алгоритм с линейным временем работы.

Анализ

Теперь покажем, что жадный алгоритм возвращает покрытие множества, не слишком сильно превышающее оптимальное покрытие. Для удобства в этой главе

d-е по порядку гармоническое число $H_d = \sum_{i=1}^{d} 1/i$ (см. раздел А.1) будет обозначаться как $H(d)$. В качестве граничного условия введем определение $H(0) = 0$.

Теорема 35.4. Алгоритм GREEDY_SET_COVER — это $\rho(n)$-приближенный алгоритм с полиномиальным временем работы, где $\rho(n) = H(\max\{|S| : S \in \mathcal{F}\})$.

Доказательство. Уже было показано, что алгоритм GREEDY_SET_COVER выполняется в течение полиномиального времени.

Чтобы показать, что GREEDY_SET_COVER — это $\rho(n)$-приближенный алгоритм, присвоим каждому из выбранных алгоритмом множеств стоимость 1, распределим эту стоимость по всем элементам, покрытым за первый раз, а потом с помощью этих стоимостей получим нужное соотношение между размером оптимального покрытия множества $\mathcal{C}^*$ и размером покрытия $\mathcal{C}$, возвращенного алгоритмом. Обозначим через S_i i-е подмножество, выбранное алгоритмом GREEDY_SET_COVER; добавление подмножества S_i в множество $\mathcal{C}$ приводит к увеличению стоимости на 1. Равномерно распределим стоимость, соответствующую выбору подмножества S_i, между элементами, которые впервые покрываются этим подмножеством. Обозначим через c_x стоимость, выделенную элементу $x \in X$. Стоимость выделяется каждому элементу только один раз, когда этот элемент покрывается впервые. Если элемент x первый раз покрывается подмножеством S_i, то выполняется равенство

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.$$

На каждом шаге алгоритма присваивается единичная стоимость, поэтому

$$|\mathcal{C}| = \sum_{x \in X} c_x.$$

Стоимость, присвоенная оптимальному покрытию, равна

$$\sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x,$$

и поскольку каждый элемент $x \in X$ принадлежит хотя бы одному множеству $S \in \mathcal{C}^*$, справедливо соотношение

$$\sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \geqslant \sum_{x \in X} c_x.$$

Объединяя два приведенных выше неравенства, получаем соотношение

$$|\mathcal{C}| \leqslant \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x. \tag{35.9}$$

Оставшаяся часть доказательства основана на приведенном ниже ключевом неравенстве, которое скоро будет доказано. Для любого множества S, принадлежащего семейству $\mathcal{F}$, выполняется неравенство

$$\sum_{x \in S} c_x \leqslant H\left(|S|\right). \tag{35.10}$$

Из неравенств (35.9) и (35.10) следует соотношение

$$|\mathcal{C}| \leqslant \sum_{S \in \mathcal{C}^*} H\left(|S|\right) \leqslant |\mathcal{C}^*| \cdot H\left(\max\left\{|S| : S \in \mathcal{F}\right\}\right),$$

которое и доказывает теорему.

Осталось доказать неравенство (35.10). Рассмотрим произвольное множество $S \in \mathcal{F}$ и индекс $i = 1, 2, \ldots, |\mathcal{C}|$, и введем величину

$$u_i = |S - (S_1 \cup S_2 \cup \cdots \cup S_i)|,$$

которая равна количеству элементов множества S, оставшихся непокрытыми после того, как в алгоритме были выбраны множества $S_1, S_2, \ldots, S_i$. Определим величину $u_0 = |S|$, равную количеству элементов множества S (которые изначально непокрыты). Пусть k — минимальный индекс, при котором выполняется равенство $u_k = 0$, т.е. каждый элемент множества S покрывается хотя бы одним из множеств $S_1, S_2, \ldots, S_k$. Тогда $u_{i-1} \geqslant u_i$, и при $i = 1, 2, \ldots, k$ $u_{i-1} - u_i$ элементов множества S впервые покрываются множеством S_i. Таким образом,

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.$$

Заметим, что

$$|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geqslant |S - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1},$$

поскольку при жадном выборе множества S_i гарантируется, что множество S не может покрыть больше новых элементов, чем множество S_i (в противном случае вместо множества S_i было бы выбрано множество S). Таким образом, мы получаем неравенство

$$\sum_{x \in S} c_x \leqslant \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}}.$$

Теперь ограничим эту величину следующим образом:

$$\begin{aligned}
\sum_{x \in S} c_x &\leqslant \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} = \\
&= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \leqslant \\
&\leqslant \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} = && \text{(так как } j \leqslant u_{i-1}\text{)} \\
&= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) = \\
&= \sum_{i=1}^{k} \left(H\left(u_{i-1}\right) - H\left(u_i\right) \right) = \\
&= H\left(u_0\right) - H\left(u_k\right) = && \text{(поскольку сумма сворачивается)} \\
&= H\left(u_0\right) - H\left(0\right) = \\
&= H\left(u_0\right) = && \text{(поскольку } H\left(0\right) = 0\text{)} \\
&= H\left(|S|\right),
\end{aligned}$$

что завершает доказательство неравенства (35.10). ■

Следствие 35.5. Алгоритм GREEDY_SET_COVER является $(\ln|X| + 1)$-приближенным алгоритмом с полиномиальным временем работы.

Доказательство. Достаточно воспользоваться неравенством (A.14) и теоремой 35.4. ■

В некоторых приложениях величина $\max\{|S| : S \in \mathcal{F}\}$ — это небольшая константа, поэтому решение, которое возвращается алгоритмом GREEDY_SET_COVER, больше оптимального на множитель, не превышающий малую константу. Одно из таких приложений — получение с помощью описанного выше эвристического метода приближенного вершинного покрытия графа, степень вершин которого не превышает 3. В этом случае решение, найденное алгоритмом GREEDY_SET_COVER, не более чем в $H\left(3\right) = 11/6$ раз больше оптимального решения, т.е. оно немного лучше решения, предоставляемого алгоритмом APPROX_VERTEX_COVER.

Упражнения

35.3-1. Каждое из перечисленных слов рассматривается как набор букв: {`arid`, `dash`, `drain`, `heard`, `lost`, `nose`, `shun`, `slate`, `snare`, `thread`}.

Приведите покрытие множества, которое будет возвращено алгоритмом GREEDY_SET_COVER, если преимущество имеет слово, которое находится в словаре выше других.

35.3-2. Покажите, что задача о покрытии множества в форме задачи принятия решения является NP-полной. Для этого приведите к ней задачу о вершинном покрытии.

35.3-3. Покажите, как реализовать процедуру GREEDY_SET_COVER, чтобы она выполнялась в течение времени $O\left(\sum_{S \in \mathcal{F}} |S|\right)$.

35.3-4. Покажите, что приведенное ниже неравенство (которое является более слабым по сравнению с тем, что было использовано в теореме 35.4) выполняется тривиальным образом:

$$|\mathcal{C}| \leqslant |\mathcal{C}^*| \max\{|S| : S \in \mathcal{F}\}.$$

35.3-5. В зависимости от принципа, по которому осуществляется выбор в строке 4, алгоритм GREEDY_SET_COVER может возвращать несколько разных решений. Разработайте процедуру BAD_SET_COVER_INSTANCE(n), возвращающую n-элементный экземпляр задачи о покрытии множества, для которого процедура GREEDY_SET_COVER при разной организации выбора в строке 4 могла бы возвращать различные решения, количество которых выражалось бы показательной функцией от n.

35.4 Рандомизация и линейное программирование

В этом разделе исследуются два метода, весьма полезных для разработки приближенных алгоритмов: рандомизация и линейное программирование. Далее приводится простой рандомизированный алгоритм, позволяющий создать оптимизирующую версию решения задачи о 3-CNF выполнимости, после чего с помощью методов линейного программирования разрабатывается приближенный алгоритм для взвешенной версии задачи о вершинном покрытии. В данном разделе эти два мощных метода рассматриваются лишь поверхностно. В сносках даются ссылки для дальнейшего изучения этой проблемы.

Рандомизированный приближенный алгоритм для задачи о MAX-3-CNF выполнимости

Рандомизированные алгоритмы можно создавать не только для поиска точных решений, но и для поиска приближенных решений. Говорят, что рандомизированный алгоритм решения задачи имеет ***коэффициент аппроксимации*** (approximation ratio) $\rho(n)$, если для любых входных данных размера n *математическое*

ожидание стоимости C решения, полученного с помощью этого рандомизированного алгоритма, не более чем в $\rho(n)$ раз превышает стоимость оптимального решения C^*:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leqslant \rho(n). \tag{35.11}$$

Рандомизированный алгоритм, позволяющий получить коэффициент аппроксимации $\rho(n)$, называют ***рандомизированным $\rho(n)$-приближенным алгоритмом*** (randomized $\rho(n)$-approximation algorithm). Другими словами, рандомизированный приближенный алгоритм похож на детерминистический приближенный алгоритм с тем отличием, что его коэффициент аппроксимации выражается через математическое ожидание.

Как видно из раздела 34.4, отдельно взятый экземпляр задачи о 3-CNF выполнимости может не выполняться. Чтобы он был выполнимым, должен существовать такой вариант присвоения переменных, при котором каждое выражение в скобках принимает значение 1. Если экземпляр невыполнимый, может возникнуть потребность оценить, насколько он "близок" к выполнимому. Другими словами, может возникнуть желание определить, какие значения следует присвоить переменным, чтобы выполнялось максимально возможное количество выражений в скобках. Назовем задачу, которая получилась в результате, задачей о ***MAX-3-CNF выполнимости*** (MAX-3-CNF satisfiability). Входные данные этой задачи совпадают с входными данными задачи о 3-CNF выполнимости, а цель состоит в том, чтобы найти присваиваемые переменным значения, при которых максимальное количество подвыражений в скобках принимает значение 1. Теперь покажем, что если значения присваиваются каждой переменной случайным образом, причем значения 0 и 1 присваивается с вероятностью $1/2$, то получится рандомизированный $8/7$-приближенный алгоритм. В соответствии с определением 3-CNF выполнимости, приведенном в разделе 34.4, требуется, чтобы в каждом выражении в скобках содержалось ровно три различных литерала. Затем предполагается, что ни одно из выражений в скобках не содержит одновременно переменной и ее отрицания. (В упражнении 35.4-1 предлагается отказаться от этого предположения.)

Теорема 35.6. Для заданного экземпляра задачи о MAX-3-CNF выполнимости с n переменными $x_1, x_2, \ldots, x_n$ и m выражениями в скобках рандомизированный алгоритм, в котором каждой переменной независимо с вероятностью $1/2$ присваивается значение 1 и с вероятностью $1/2$ — значение 0, является рандомизированным $8/7$-приближенным алгоритмом.

Доказательство. Предположим, что каждой переменной независимо с вероятностью $1/2$ присваивается значение 1 и с вероятностью $1/2$ — значение 0. Определим для $i = 1, 2, \ldots, m$ индикаторную случайную величину

$$Y_i = \mathrm{I}\{i\text{-е подвыражение в скобках выполняется}\}$$

так, что равенство $Y_i = 1$ выполняется, если хоть одному из литералов, содержащихся в i-м выражении в скобках, присвоено значение 1. Поскольку ни один из литералов не входит более одного раза в одно и то же выражение в скобках, и поскольку предполагается, что одни и те же скобки не содержат одновременно переменную и ее отрицание, присвоение значений трем переменным в каждых скобках выполняется независимым образом. Выражение в скобках не выполняется только тогда, когда всем трем его литералам присваивается значение 0, поэтому

$$\Pr\{i\text{-е подвыражение не выполняется}\} = (1/2)^3 = 1/8.$$

Тогда

$$\Pr\{i\text{-е подвыражение выполняется}\} = 1 - 1/8 = 7/8.$$

Поэтому, согласно лемме 5.1, $\mathrm{E}[Y_i] = 7/8$. Пусть всего выполняется Y подвыражений в скобках, т.е. $Y = \sum_{i=1}^{m} Y_i$. Тогда мы получаем

$$\mathrm{E}[Y] = \mathrm{E}\left[\sum_{i=1}^{m} Y_i\right] = \sum_{i=1}^{m} \mathrm{E}[Y_i] = \sum_{i=1}^{m} 7/8 = 7m/8,$$

где второе равенство следует из линейности математического ожидания. Очевидно, что верхняя граница количества выполняющихся подвыражений в скобках равна m, поэтому коэффициент аппроксимации не превышает значения $m/(7m/8) = 8/7$. ■

Аппроксимация взвешенного вершинного покрытия с помощью линейного программирования

В ***задаче о вершинном покрытии с минимальным весом*** (minimum-weight vertex-cover problem) задается неориентированный граф $G = (V, E)$, в котором каждой вершине $v \in V$ назначается положительный вес $w(v)$. Вес любого вершинного покрытия $V' \subseteq V$ определяется как $w(V') = \sum_{v \in V'} w(v)$. В задаче нужно найти вершинное покрытие с минимальным весом.

К этой задаче нельзя применить ни алгоритм, который использовался для поиска невзвешенного вершинного покрытия, ни рандомизированное решение, так как оба эти метода могут дать решение, далекое от оптимального. Однако с помощью задачи линейного программирования можно вычислить нижнюю границу веса, который может возникать в задаче о вершинном покрытии минимального веса. Затем мы "округлим" это решение и с его помощью получим вершинное покрытие.

Предположим, что каждой вершине $v \in V$ сопоставляется переменная $x(v)$, и поставим условие, чтобы $x(v) \in \{0, 1\}$ для каждой вершины $v \in V$. Равенство $x(v) = 1$ интерпретируется как принадлежность вершины v вершинному покрытию, а равенство $x(v) = 0$ — как ее отсутствие в вершинном покрытии. Тогда

ограничение, согласно которому для любого ребра (u, v) хотя бы одна из вершин u и v должна входить в вершинное покрытие, можно наложить с помощью неравенства $x(u) + x(v) \geqslant 1$. Такой подход приводит нас к ***0-1 целочисленной задаче линейного программирования*** (0-1 integer program) поиска вершинного покрытия с минимальным весом:

$$\text{Минимизировать} \sum_{v \in V} w(v)\, x(v) \tag{35.12}$$

$$\text{при условиях} \quad x(u) + x(v) \geqslant 1 \qquad \text{для всех } (u, v) \in E \tag{35.13}$$

$$x(v) \in \{0, 1\} \qquad \text{для всех } v \in V. \tag{35.14}$$

В частном случае, когда все веса $w(v)$ равны 1, мы получаем NP-сложную оптимизирующую версию задачи о вершинном покрытии. Из упражнения 34.5-2 известно, что обычный поиск величин $x(v)$, удовлетворяющих условиям (35.13) и (35.14), — NP-сложная задача, и что непосредственная польза извлекается не из такой формулировки. Предположим, однако, что вместо ограничения $x(v) \in \{0, 1\}$ накладывается условие $0 \leqslant x(v) \leqslant 1$. Тогда мы получим ***ослабленную задачу линейного программирования*** (linear-programming relaxation):

$$\text{Минимизировать} \sum_{v \in V} w(v)\, x(v) \tag{35.15}$$

$$\text{при условиях} \quad x(u) + x(v) \geqslant 1 \qquad \text{для всех } (u, v) \in E \tag{35.16}$$

$$x(v) \leqslant 1 \qquad \text{для всех } v \in V \tag{35.17}$$

$$x(v) \geqslant 0 \qquad \text{для всех } v \in V. \tag{35.18}$$

Любое допустимое решение 0-1 целочисленной задачи линейного программирования, определенной выражениями (35.12)–(35.14), также является допустимым решением задачи линейного программирования, определенной выражениями (35.15)–(35.18). Поэтому оптимальное решение задачи линейного программирования является нижней границей оптимального решения 0-1 целочисленной задачи, а следовательно, нижней границей оптимального решения задачи о вершинном покрытии с минимальным весом.

В приведенной ниже процедуре с помощью решения сформулированной выше задачи линейного программирования строится приближенное решение задачи о вершинном покрытии с минимальным весом:

```
APPROX_MIN_WEIGHT_VC(G, w)
1  C ← ∅
2  Вычисляется x̄, оптимальное решение задачи линейного
   программирования (35.15)–(35.18)
3  for (для) каждой вершины v ∈ V
```

```
4       do if x̄(v) ⩾ 1/2
5             then C ← C ∪ {v}
6   return C
```

Опишем работу процедуры APPROX_MIN_WEIGHT_VC. В строке 1 инициализируется пустое вершинное покрытие. В строке 2 формулируется и решается задача линейного программирования, определенная соотношениями (35.15)–(35.18). В оптимальном решении каждой вершине v сопоставляется величина $0 \leqslant \bar{x}(v) \leqslant 1$. С помощью этой величины в строках 3–5 определяется, какие вершины добавятся в вершинное покрытие C. Если $\bar{x}(v) \geqslant 1/2$, вершина v добавляется в покрытие C; в противном случае она не добавляется. В результате "округляется" каждая дробная величина, входящая в решение задачи линейного программирования, и получается решение 0-1 целочисленной задачи, определенной соотношениями (35.12)–(35.14). Наконец, в строке 6 возвращается вершинное покрытие C.

Теорема 35.7. Алгоритм APPROX_MIN_WEIGHT_VC — это 2-приближенный алгоритм с полиномиальным временем выполнения, позволяющий решить задачу о вершинном покрытии с минимальным весом.

Доказательство. Поскольку существует алгоритм с полиномиальным временем решения, позволяющий решить задачу линейного программирования в строке 2, и поскольку цикл **for** в строках 3–5 завершает свою работу в течение полиномиального времени, алгоритм APPROX_MIN_WEIGHT_VC выполняется в течение полиномиального времени.

Теперь покажем, что он является 2-приближенным алгоритмом. Пусть C^* — оптимальное решение задачи о вершинном покрытии с минимальным весом, а z^* — значение оптимального решения задачи линейного программирования, определенной соотношениями (35.15)–(35.18). Поскольку оптимальное вершинное покрытие — допустимое решение этой задачи, величина z^* должна быть нижней границей величины $w(C^*)$, т.е. выполняется неравенство

$$z^* \leqslant w(C^*) \tag{35.19}$$

Далее, утверждается, что при округлении дробных значений переменных $\bar{x}(v)$ получается множество C, которое является вершинным покрытием, удовлетворяющим неравенству $w(C) \leqslant 2z^*$. Чтобы убедиться, что C — вершинное покрытие, рассмотрим произвольное ребро $(u, v) \in E$. Согласно ограничению (35.16), должно выполняться неравенство $x(u) + x(v) \geqslant 1$, из которого следует, что хотя бы одна из величин $\bar{x}(u)$ и $\bar{x}(v)$ не меньше $1/2$. Поэтому хотя бы одна из вершин u и v будет включена в вершинное покрытие, а следовательно, будут покрыты все ребра.

Теперь рассмотрим, чему равен вес этого покрытия. Имеем следующую цепочку соотношений:

$$\begin{aligned} z^* &= \sum_{v \in V} w(v)\,\bar{x}(v) \geqslant \\ &\geqslant \sum_{v \in V : \bar{x}(v) \geqslant 1/2} w(v)\,\bar{x}(v) \geqslant \\ &\geqslant \sum_{v \in V : \bar{x}(v) \geqslant 1/2} w(v) \cdot \frac{1}{2} = \\ &= \sum_{v \in C} w(v) \cdot \frac{1}{2} = \\ &= \frac{1}{2} \sum_{v \in C} w(v) = \\ &= \frac{1}{2} w(C)\,. \end{aligned} \tag{35.20}$$

Объединение неравенств (35.19) и (35.20) дает нам соотношение

$$w(C) \leqslant 2z^* \leqslant 2w(C^*)\,,$$

так что алгоритм APPROX_MIN_WEIGHT_VC является 2-приближенным алгоритмом. ■

Упражнения

35.4-1. Покажите, что даже если бы выражение в скобках могло содержать одновременно переменную и ее отрицание, то в результате независимого присвоения каждой переменной значения 1 с вероятностью $1/2$ и значения 0 с вероятностью $1/2$ все равно получился бы рандомизированный $8/7$-приближенный алгоритм.

35.4-2. ***Задача о MAX-CNF выполнимости*** (MAX-CNF satisfiability problem) похожа на задачу о MAX-3-CNF выполнимости с тем исключением, что в каждом выражении в скобках не обязательно содержится по 3 литерала. Разработайте рандомизированный 2-приближенный алгоритм, позволяющий решить задачу о MAX-CNF выполнимости.

35.4-3. В задаче MAX_CUT задается невзвешенный неориентированный граф $G = (V, E)$. Определим разрез $(S, V - S)$, как это было сделано в главе 23, и ***вес*** (weight) этого разреза, как количество пересекающих его ребер. Требуется найти разрез с максимальным весом. Предположим, что каждая вершина v случайно и независимо с вероятностью $1/2$ помещается в множество S или $V - S$. Покажите, что этот алгоритм является рандомизированным 2-приближенным алгоритмом.

35.4-4. Покажите, что ограничение (35.17) избыточно в том смысле, что если опустить его из задачи линейного программирования, определенной соотношениями (35.15)–(35.18), то любое оптимальное решение получившейся в результате задачи для каждой вершины $v \in V$ должно удовлетворять неравенству $x(v) \leqslant 1$.

35.5 Задача о сумме подмножества

Экземпляр задачи о сумме подмножества имеет вид пары (S, t), где S — множество $\{x_1, x_2, \ldots, x_n\}$ положительных целых чисел, а t — положительное целое число. В этой задаче принятия решений спрашивается, существует ли подмножество множества S, сумма элементов которого равна целевому значению t. Эта задача является NP-полной (см. раздел 34.5.5).

Задача оптимизации, связанная с этой задачей принятия решений, возникает в различных практических приложениях. В такой задаче требуется найти подмножество множества $\{x_1, x_2, \ldots, x_n\}$, сумма элементов которого принимает максимально возможное значение, не большее t. Например, пусть имеется грузовик, грузоподъемность которого не превышает t кг, и n различных ящиков, которые нужно перевезти. Вес i-го ящика равен x_i кг. Требуется загрузить грузовик максимально возможным весом, но так, чтобы не превысить его грузоподъемности.

В этом разделе приведен алгоритм, позволяющий решить задачу оптимизации в течение экспоненциального времени. Затем показано, как модифицировать приведенный алгоритм, чтобы он превратился в схему аппроксимации с полностью полиномиальным временем решения. (Напомним, что время работы схемы аппроксимации с полностью полиномиальным временем выполнения выражается полиномиальной функцией от величины $1/\varepsilon$ и от размера входных данных.)

Точный алгоритм с экспоненциальным временем работы

Предположим, что для каждого подмножества S' множества S вычисляется сумма его элементов, после чего из всех подмножеств, сумма элементов которых не превышает t, выбирается подмножество, для которого эта сумма меньше других отличается от t. Очевидно, что такой алгоритм возвратит оптимальное решение, но время его работы выражается показательной функцией. Для реализации этого алгоритма можно было бы воспользоваться итеративной процедурой, в которой в i-й итерации вычислялись бы суммы элементов всех подмножеств множества $\{x_1, x_2, \ldots, x_i\}$ на основе вычисленных ранее сумм всех подмножеств множества $\{x_1, x_2, \ldots, x_{i-1}\}$. Если пойти этим путем, то легко понять, что нет смысла продолжать обработку некоторого подмножества S' после того, как сумма его элементов превысит t, поскольку никакое надмножество S' не может быть оптимальным решением. Реализуем эту стратегию.

В качестве входных данных процедуры EXACT_SUBSET_SUM, с псевдокодом которой мы скоро ознакомимся, выступает множество $S = \{x_1, x_2, \ldots, x_n\}$ и целевое значение t. В этой процедуре итеративно вычисляется список L_i, содержащий суммы всех подмножеств множества $\{x_1, x_2, \ldots, x_i\}$, которые не превышают t. Затем возвращается максимальное значение из списка L_n.

Если L — список положительных целых чисел, а x — еще одно положительное целое число, тогда через $L + x$ будет обозначаться список целых чисел, полученный из списка L путем увеличения каждого его элемента на величину x. Например, если $L = \langle 1, 2, 3, 5, 9 \rangle$, то $L + 2 = \langle 3, 4, 5, 7, 11 \rangle$. Воспользуемся этим обозначением и для множеств:

$$S + x = \{s + x : s \in S\} .$$

Кроме того, нам понадобится вспомогательная процедура MERGE_LISTS(L, L'), которая возвращает отсортированный список, представляющий собой объединение входных списков L и L' с исключением повторяющихся значений. Время выполнения процедуры MERGE_LISTS, как и время выполнения процедуры MERGE, используемой для сортировки слиянием и описанной в разделе 2.3.1, ведет себя как $O(|L| + |L'|)$. (Псевдокод процедуры MERGE_LISTS опускается.)

```
EXACT_SUBSET_SUM(S, t)
1  n ← |S|
2  L_0 ← ⟨0⟩
3  for i ← 1 to n
4       do L_i ← MERGE_LISTS(L_{i-1}, L_{i-1} + x_i)
5          Из списка L_i удаляются все элементы, большие t
6  return Максимальный элемент из списка L_n
```

Рассмотрим, как работает процедура EXACT_SUBSET_SUM. Обозначим через P_i множество всех значений, которые можно получить, выбрав (возможно, пустое) подмножество множества $\{x_1, x_2, \ldots, x_i\}$ и просуммировав его элементы. Например, если $S = \{1, 4, 5\}$, то

$$\begin{aligned} P_1 &= \{0, 1\} , \\ P_2 &= \{0, 1, 4, 5\} , \\ P_3 &= \{0, 1, 4, 5, 6, 9, 10\} . \end{aligned}$$

Воспользовавшись тождеством

$$P_i = P_{i-1} \cup (P_{i-1} + x_i) , \tag{35.21}$$

методом математической индукции по i можно доказать (см. упражнение 35.5-1), что L_i — отсортированный список, содержащий все элементы множества P_i, значение которых не превышает t. Поскольку длина списка L_i может достигать

значения 2^i, в общем случае время выполнения алгоритма EXACT_SUBSET_SUM ведет себя как показательная функция, хотя в некоторых случаях, когда величина t представляет собой полином от $|S|$ или все содержащиеся в множестве S числа ограничены сверху полиномиальными величинами от $|S|$, это время также полиномиально зависит от $|S|$.

Схема аппроксимации с полностью полиномиальным временем работы

Схему аппроксимации с полностью полиномиальным временем работы, позволяющую получить приближенное решение задачи о сумме подмножеств, можно составить путем "сокращения" каждого списка L_i после его создания. Идея заключается в том, что если два значения в списке L мало отличаются друг от друга, то для получения приближенного решения нет смысла явно обрабатывать оба эти значения. Точнее говоря, используется некоторый параметр сокращения δ, удовлетворяющий неравенству $0 < \delta < 1$. Чтобы ***сократить*** (trim) список L по параметру δ, нужно удалить из этого списка максимальное количество элементов таким образом, чтобы в полученном в результате этого сокращения списке L' для каждого удаленного из списка L элемента y содержался элемент z, аппроксимирующий элемент y, так что

$$\frac{y}{1+\delta} \leqslant z \leqslant y. \tag{35.22}$$

Можно считать, что элемент z "представляет" элемент y в списке L', т.е. каждый элемент y представлен элементом z, удовлетворяющим уравнению (35.22). Например, если $\delta = 0.1$ и

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle ,$$

то путем сокращения списка L можно получить список

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle ,$$

где удаленное значение 11 представлено значением 10, удаленные значения 21 и 22 представлены значением 20, а удаленное значение 24 представлено значением 23. Поскольку каждый элемент измененной версии списка является одновременно элементом исходного списка, сокращение может значительно уменьшить количество элементов, сохраняя в списке близкие (несколько меньшие) представляющие значения каждого удаленного элемента.

Приведенная ниже процедура по заданным параметрам L и δ сокращает список $L = \langle y_1, y_2, \ldots, y_m \rangle$ в течение времени $\Theta(m)$; при этом предполагается, что элементы списка L отсортированы в монотонно возрастающем порядке. На выходе процедуры получается сокращенный отсортированный список.

```
TRIM(L, δ)
1  m ← |L|
2  L' ← ⟨y_1⟩
3  last ← y_1
4  for i ← 2 to m
5       do if y_i > last ·(1 + δ)          ▷ y_i ≥ last в силу
                                           ▷ отсортированности списка L
6     then элемент y_i добавляется в конец списка L'
7          last ← y_i
8  return L'
```

Элементы списка L сканируются в монотонно возрастающем порядке, и в список L', который возвращается, элемент помещается, только если это первый элемент списка L или если его нельзя представить последним помещенным в список L' элементом.

Располагая процедурой TRIM, схему аппроксимации можно построить следующим образом. В качестве входных данных в эту процедуру передается множество $S = \{x_1, x_2, \ldots, x_n\}$, состоящее из n целых чисел (расположенных в произвольном порядке), целевое значение t и "параметр аппроксимации" ε, где

$$0 < \varepsilon < 1. \tag{35.23}$$

Процедура возвращает значение z, величина которого отличается от оптимального решения не более чем в $1 + \varepsilon$ раз.

```
APPROX_SUBSET_SUM(S, t, ε)
1  n ← |S|
2  L_0 ← ⟨0⟩
3  for i ← 1 to n
4       do L_i ← MERGE_LISTS(L_{i−1}, L_{i−1} + x_i)
5          L_i ← TRIM(L_i, ε/2n)
6          Из списка L_i удаляются все элементы, большие t
7  Пусть z* — максимальное значение в списке L_n
8  return z*
```

В строке 2 список L_0 инициализируется таким образом, чтобы в нем содержался только элемент 0. Цикл **for** в строках 3–6 вычисляет отсортированный список L_i, содержащий соответствующим образом сокращенную версию множества P_i, из которого удалены все элементы, превышающие величину t. Поскольку список L_i создается на основе списка L_{i-1}, мы должны гарантировать, что повторное сокращение не вносит слишком большой неточности. Скоро станет понятно, что процедура APPROX_SUBSET_SUM возвращает корректную аппроксимацию, если она существует.

В качестве примера рассмотрим экземпляр задачи, где $S = \langle 104, 102, 201, 101 \rangle$, $t = 308$ и $\varepsilon = 0.40$. Параметр δ равен $\varepsilon/8 = 0.05$. В процедуре APPROX_SUBSET_SUM вычисляются приведенные ниже значения (слева от которых указаны номера соответствующих строк):

Строка 2: $L_0 = \langle 0 \rangle$,

Строка 4: $L_1 = \langle 0, 104 \rangle$,
Строка 5: $L_1 = \langle 0, 104 \rangle$,
Строка 6: $L_1 = \langle 0, 104 \rangle$,

Строка 4: $L_2 = \langle 0, 102, 104, 206 \rangle$,
Строка 5: $L_2 = \langle 0, 102, 206 \rangle$,
Строка 6: $L_2 = \langle 0, 102, 206 \rangle$,

Строка 4: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$,
Строка 5: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$,
Строка 6: $L_3 = \langle 0, 102, 201, 303 \rangle$,

Строка 4: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$,
Строка 5: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$,
Строка 6: $L_4 = \langle 0, 101, 201, 302 \rangle$.

Алгоритм возвращает значение $z^* = 302$, которое приближает оптимальное решение $307 = 104 + 102 + 101$ в пределах погрешности $\varepsilon = 40\%$; фактически погрешность составляет 2%.

Теорема 35.8. Процедура APPROX_SUBSET_SUM является схемой аппроксимации с полностью полиномиальным временем выполнения, позволяющей решить задачу о сумме подмножеств.

Доказательство. В результате сокращения списка L_i в строке 5 и удаления из этого списка всех элементов, превышающих значение t, поддерживается свойство, что каждый элемент списка L_i также является элементом списка P_i. Поэтому значение z^*, которое возвращается в строке 8, на самом деле является суммой некоторого подмножества множества S. Обозначим оптимальное решение задачи о сумме подмножества $y^* \in P_n$. Далее, из строки 6 известно, что $z^* \leqslant y^*$. Согласно неравенству (35.1), нужно показать, что $y^*/z^* \leqslant 1+\varepsilon$. Необходимо также показать, что время работы этого алгоритма выражается полиномиальной функцией и от $1/\varepsilon$, и от размера входных данных.

Воспользовавшись индукцией по i, можно показать, что для каждого не превышающего t элемента y из списка P_i существует элемент $z \in L_i$, такой что

$$\frac{y}{(1+\varepsilon/2n)^i} \leqslant z \leqslant y \tag{35.24}$$

(см. упражнение 35.5-2). Неравенство (35.24) должно выполняться для $y^* \in P_n$, поэтому существует элемент $z \in L_n$, удовлетворяющий неравенству

$$\frac{y^*}{(1+\varepsilon/2n)^n} \leqslant z \leqslant y^*$$

и, следовательно,

$$\frac{y^*}{z} \leqslant \left(1+\frac{\varepsilon}{2n}\right)^n. \tag{35.25}$$

Поскольку существует элемент $z \in L_n$, удовлетворяющий неравенству (35.25), это неравенство должно быть справедливым для элемента z^*, который имеет самое большое значение в списке L_n, т.е.

$$\frac{y^*}{z^*} \leqslant \left(1+\frac{\varepsilon}{2n}\right)^n. \tag{35.26}$$

Осталось показать, что $y^*/z^* \leqslant 1+\varepsilon$. Для этого покажем, что $(1+\varepsilon/2n)^n \leqslant$ $\leqslant 1+\varepsilon$. Согласно (3.13), $\lim_{n\to\infty}(1+\varepsilon/2n)^n = e^{\varepsilon/2}$. Поскольку можно показать, что

$$\frac{d}{dn}\left(1+\frac{\varepsilon}{2n}\right)^n > 0, \tag{35.27}$$

функция $(1+\varepsilon/2n)^n$ по мере увеличения n возрастает до своего предела $e^{\varepsilon/2}$, и справедлива цепочка неравенств

$$\begin{aligned}\left(1+\frac{\varepsilon}{2n}\right)^n &\leqslant e^{\frac{\varepsilon}{2}} \leqslant \\ &\leqslant 1+\frac{\varepsilon}{2}+\left(\frac{\varepsilon}{2}\right)^2 \leqslant && \text{(в соответствии с (3.12))} \\ &\leqslant 1+\varepsilon && \text{(с учетом (35.23)).}\end{aligned} \tag{35.28}$$

Объединение неравенств (35.26) и (35.28) завершает анализ коэффициента аппроксимации.

Чтобы показать, что процедура APPROX_SUBSET_SUM — это схема аппроксимации с полностью полиномиальным временем выполнения, оценим границу длины списка L_i. После сокращения последовательные элементы z и z' в списке L_i должны удовлетворять соотношению $z'/z > 1+\varepsilon/2n$. Другими словами, они должны отличаться не менее чем в $1+\varepsilon/2n$ раз. Поэтому каждый список содержит

значение 0, возможно, значение 1 и до $\left\lfloor \log_{1+\varepsilon/2n} t \right\rfloor$ дополнительных значений. Количество элементов в каждом списке L_i не превышает

$$\begin{aligned}
\log_{1+\varepsilon/2n} t + 2 &= \frac{\ln t}{\ln(1+\varepsilon/2n)} + 2 \leqslant \\
&\leqslant \frac{2n(1+\varepsilon/2n)\ln t}{\varepsilon} + 2 \leqslant && \text{(в соответствии с (3.16))} \\
&\leqslant \frac{4n \ln t}{\varepsilon} + 2 && \text{(с учетом (35.23)).}
\end{aligned}$$

Эта граница выражается полиномиальной функцией от размера входных данных, который равен сумме количества битов $\lg t$, необходимых для представления числа t, и количества битов, необходимых для представления множества S, которое, в свою очередь, полиномиально зависит от n и от $1/\varepsilon$. Поскольку время выполнения процедуры APPROX_SUBSET_SUM выражается полиномиальной функцией от длины списка L_i, эта процедура является схемой аппроксимации с полностью полиномиальным временем выполнения. ■

Упражнения

35.5-1. Докажите уравнение (35.21). Затем покажите, что после выполнения строки 5 процедуры EXACT_SUBSET_SUM список L_i является отсортированным и содержит все элементы списка P_i, значения которых не превышают t.

35.5-2. Докажите неравенство (35.24).

35.5-3. Докажите неравенство (35.27).

35.5-4. Как следовало бы модифицировать представленную в этом разделе схему аппроксимации, чтобы она позволяла найти наименьшее значение, превышающее или равное заданной величине t, суммы элементов некоторого подмножества заданного входного списка?

Задачи

35-1. **Расфасовка по контейнерам**

Предположим, имеется множество, состоящее из n предметов, причем размер i-го предмета s_i удовлетворяет неравенству $0 < s_i < 1$. Нужно упаковать все предметы в контейнеры единичного размера, использовав при этом минимальное количество контейнеров. Каждый контейнер вмещает произвольное количество объектов, лишь бы их суммарный размер не превышал 1.

а) Докажите, что задача по определению минимального количества необходимых контейнеров является NP-полной. (*Указание*: приведите к ней задачу о сумме подмножества.)

При эвристическом методе ***выбора первого подходящего*** (first-fit) по очереди выбираются все предметы, и каждый помещается в первый же контейнер, в который этот предмет может поместиться. Пусть $S = \sum_{i=1}^{n} s_i$.

б) Докажите, что оптимальное количество контейнеров, необходимых для упаковки всех предметов, не меньше $\lceil S \rceil$.

в) Докажите, что при использовании эвристического метода выбора первого подходящего, не более чем один контейнер остается заполненным меньше чем наполовину.

г) Докажите, что количество контейнеров, которые используются в эвристическом методе выбора первого подходящего, никогда не превышает величину $\lceil 2S \rceil$.

д) Докажите, что коэффициент аппроксимации эвристического метода выбора первого подходящего равен 2.

е) Представьте эффективную реализацию эвристического метода выбора первого подходящего и проанализируйте время работы полученного алгоритма.

35-2. **Приближенный размер максимальной клики**

Пусть $G = (V, E)$ — неориентированный граф. Определим для произвольного числа $k \geqslant 1$ неориентированный граф $G^{(k)} = \left(V^{(k)}, E^{(k)}\right)$, где $V^{(k)}$ — множество всех упорядоченных k-кортежей вершин из множества V, а $E^{(k)}$ определяется таким образом, что элемент $(v_1, v_2, \ldots, v_k)$ смежен элементу $(w_1, w_2, \ldots, w_k)$ тогда и только тогда, когда при любом значении $1 \leqslant i \leqslant k$ каждая вершина v_i является смежной в графе G вершине w_i (либо $v_i = w_i$).

а) Докажите, что размер максимальной клики в графе $G^{(k)}$ равен k-й степени размера максимальной клики в графе G.

б) Докажите, что если существует приближенный алгоритм, позволяющий найти клику максимального размера и обладающий постоянным коэффициентом аппроксимации, то для решения данной задачи существует схема аппроксимации с полностью полиномиальным временем работы.

35-3. **Взвешенная задача о покрытии множества**

Предположим, задача о покрытии множества обобщается таким образом, что каждому множеству S_i из семейства $\mathcal{F}$ сопоставляется вес w_i, а вес

покрытия $\mathcal{C}$ вычисляется как $\sum_{S_i \in \mathcal{C}} w_i$. Нужно найти покрытие с минимальным весом. (В разделе 35.3 рассматривается случай, когда $w_i = 1$ для всех i.)

Покажите, что жадный эвристический подход, который применяется в задаче о покрытии множества, можно естественным образом обобщить так, чтобы он позволял получить приближенное решение любого экземпляра взвешенной задачи о покрытии множества. Покажите, что коэффициент аппроксимации этого эвристического подхода равен $H(d)$, где d — максимальный размер произвольного множества S_i.

35-4. **Паросочетание максимальной мощности**

Напомним, что в неориентированном графе G паросочетанием называется такое множество ребер, в котором никакие два ребра не инцидентны одной и той же вершине. Из раздела 26.3 мы узнали, как найти максимальное паросочетание в двудольном графе. В настоящей задаче будет производиться поиск паросочетаний в неориентированных графах общего вида (т.е. в графах, которые не обязательно являются двудольными).

а) ***Максимальным паросочетанием*** (maximal matching) называется паросочетание, которое не является собственным подмножеством никакого другого паросочетания. Покажите, что максимальное паросочетание не обязательно совпадает с паросочетанием максимальной мощности. Для этого приведите пример неориентированного графа G, максимальное паросочетание M в котором не является паросочетанием максимальной мощности. (Имеется граф всего лишь с четырьмя вершинами, обладающий указанным свойством.)

б) Рассмотрим неориентированный граф $G = (V, E)$. Сформулируйте жадный алгоритм поиска максимального паросочетания в графе G, время работы которого было бы равно $O(E)$.

В этой задаче внимание сосредотачивается на поиске приближенного алгоритма с полиномиальным временем работы, позволяющего найти паросочетание максимальной мощности. Время работы самого быстрого из известных на сегодняшний день алгоритмов, предназначенных для поиска паросочетания максимальной мощности, превышает линейное (хотя и является полиномиальным); рассматриваемый же здесь приближенный алгоритм завершает свою работу строго в течение линейного времени. Вы должны будете показать, что жадный алгоритм поиска максимального паросочетания с линейным временем выполнения, разработанный в части *б*, для задачи о паросочетании максимальной мощности является 2-приближенным алгоритмом.

в) Покажите, что размер паросочетания максимальной мощности в графе G представляет собой нижнюю границу размера произвольного вершинного покрытия в этом графе.

г) Рассмотрим максимальное паросочетание M в графе $G = (V, E)$. Пусть

$$T = \{v \in V : \text{ некоторое ребро из } M \text{ инцидентно } v\}.$$

Что можно сказать о подграфе графа G, порожденном теми вершинами графа G, которые не принадлежат T?

д) На основании результатов части *г* обоснуйте вывод о том, что величина $2|M|$ равна размеру вершинного покрытия графа G.

е) Воспользовавшись результатами решения частей *в* и *д* задачи, докажите, что сформулированный в части *б* жадный алгоритм является 2-приближенным алгоритмом для задачи о паросочетании максимальной мощности.

35-5. Расписание работы параллельной вычислительной машины

В задаче о ***расписании работы параллельной вычислительной машины*** (parallel-machine-scheduling problem) исходные данные представляют собой набор из n заданий $J_1, J_2, \ldots, J_n$, каждое из которых характеризуется временем обработки p_k. Для выполнения этих заданий в нашем распоряжении имеется m идентичных машин $M_1, M_2, \ldots, M_m$. Нужно составить ***расписание***, в котором для каждого задания J_k следует указать машину, на которой это задание будет выполняться, и выделенный ему интервал времени. Каждое задание должно непрерывно выполняться только на одной машине M_i в течение времени p_k, и в это время на машине M_i не может выполняться никакое другое задание. Обозначим через C_k ***время завершения*** (completion time) задания J_k, т.е. момент времени, когда прекращается обработка задания J_k. Для каждого расписания определяется его ***момент завершения*** (makespan), равный $C_{\max} = \max_{1 \leqslant j \leqslant n} C_j$. В задаче нужно найти расписание с минимальным моментом завершения.

Например, предположим, что имеется две машины M_1 и M_2 и что нужно выполнить четыре задания J_1, J_2, J_3, J_4, для которых $p_1 = 2$, $p_2 = 12$, $p_3 = 4$ и $p_4 = 5$. Можно предложить расписание, при котором на машине M_1 сначала выполняется задание J_1, а затем — задание J_2, а на машине M_2 сначала выполняется задание J_4, а затем — задание J_3. В этом расписании $C_1 = 2$, $C_2 = 14$, $C_3 = 9$, $C_4 = 5$ и $C_{\max} = 14$. В оптимальном расписании на машине M_1 выполняется задание J_2, а на машине M_2 — задания J_1, J_3 и J_4. В этом расписании $C_1 = 2$, $C_2 = 12$, $C_3 = 6$, $C_4 = 11$ и $C_{\max} = 12$.

В данной задаче о расписании работы параллельной вычислительной машины обозначим через $C^*_{\max}$ время завершения оптимального расписания.

а) Покажите, что оптимальное время завершения по величине не меньше самого большого времени обработки, т.е. что выполняется неравенство

$$C^*_{\max} \geqslant \max_{1 \leqslant k \leqslant n} p_k.$$

б) Покажите, что оптимальное время завершения по величине не меньше средней загрузки машин, т.е. что справедливо неравенство

$$C^*_{\max} \geqslant \frac{1}{m} \sum_{1 \leqslant k \leqslant n} p_k.$$

Предположим, что для составления расписания параллельных вычислительных машин используется следующий жадный алгоритм: как только машина освобождается, на ней начинает выполняться очередное задание, еще не внесенное в расписание.

в) Предложите псевдокод, реализующий этот жадный алгоритм. Чему равно время работы этого алгоритма?

г) Покажите, что для расписания, которое возвращается жадным алгоритмом, выполняется неравенство

$$C_{\max} \leqslant \frac{1}{m} \sum_{1 \leqslant k \leqslant n} p_k + \max_{1 \leqslant k \leqslant n} p_k.$$

Обоснуйте вывод, согласно которому этот алгоритм является 2-приближенным алгоритмом с полиномиальным временем выполнения.

Заключительные замечания

Несмотря на то, что методы, которые не обязательно вычисляют точные решения, были известны несколько тысяч лет назад (например, методы приближенного вычисления числа π), понятие приближенного алгоритма имеет намного более короткую историю. Заслуги по формализации концепции приближенного алгоритма с полиномиальным временем работы Хохбаум (Hochbaum) приписывает Гарею (Garey), Грэхему (Graham) и Ульману (Ullman) [109], а также Джонсону (Johnson) [166]. Первый такой алгоритм (ему посвящена задача 35-5) часто приписывается Грэхему [129].

Со времени публикации этой ранней работы были разработаны тысячи приближенных алгоритмов, позволяющих решать самые разнообразные задачи. По этой теме имеется большое количество литературы. Недавно вышедшие книги Осиэлло (Ausiello) и др. [25], Хохбаума [149] и Вазирани (Vazirani) [305] полностью посвящены приближенным алгоритмам. Это же можно сказать об обзорах Шмойса (Shmoys) [277] и Клейна (Klein) и Юнга (Young) [181]. В нескольких других книгах, таких как книги Гарея и Джонсона [110], а также Пападимитриу (Papadimitriou) и Штейглица (Steiglitz) [237], также значительное внимание уделяется приближенным алгоритмам. В книге Лоулера (Lawler), Ленстры (Lenstra), Ринноя Кана (Rinnooy Kan) и Шмойса [197] подробно рассматриваются приближенные алгоритмы, предназначенные для задачи о коммивояжере.

В книге Пападимитриу и Штейглица авторство алгоритма APPROX_VERTEX_COVER приписывается Ф. Гаврилу (F. Gavril) и М. Яннакакису (M. Yannakakis). Большое количество усилий было направлено на исследование задачи о вершинном покрытии (в книге Хохбаума [149] приведено 16 различных приближенных алгоритмов, предназначенных для решения этой задачи), однако значение всех коэффициентов аппроксимации не меньше $2 - o(1)$.

Алгоритм APPROX_TSP_TOUR был предложен в статье Розенкранца (Rosenkrantz), Стирнса (Stearns) и Льюиса (Lewis) [261]. Кристофидис (Christofides) усовершенствовал этот алгоритм и предложил 3/2-приближенный алгоритм, позволяющий решить задачу о коммивояжере с неравенством треугольника. Арора (Arora) [21] и Митчелл (Mitchell) [223] показали, что если точки находятся на евклидовой плоскости, то существует схема аппроксимации с полиномиальным временем работы. Теорема 35.3 доказана Сани (Sahni) и Гонзалезом (Gonzalez) [264].

Анализ жадного эвристического подхода к задаче о покрытии множества построен по аналогии с доказательством более общего результата, опубликованном в статье Чватала (Chvatal) [61]; представленный здесь основной результат доказан Джонсоном [166] и Ловасом (Lovasz) [206].

Описание алгоритма APPROX_SUBSET_SUM и его анализ с некоторыми изменениями взяты из статьи Ибарры (Ibarra) и Кима (Kim) [164], где приведены приближенные алгоритмы, предназначенные для решения задач о рюкзаке и о сумме подмножества.

Рандомизированный алгоритм решения задачи о MAX-3-CNF выполнимости можно найти в неявном виде в работе Джонсона [166]. Автором алгоритма, предназначенного для решения задачи о взвешенном вершинном покрытии, является Хохбаум [148]. Раздел 35.4 дает лишь поверхностное представление о тех возможностях, которые открываются благодаря использованию рандомизации и линейного программирования при разработке приближенных алгоритмов. Сочетание этих двух идей привело к появлению метода под названием “рандомизированное округление”, в котором задача сначала формулируется как целочисленная задача линейного программирования. После этого решается ослабленный вариант задачи,

а переменные в этом решении интерпретируются как вероятности. Затем эти вероятности используются для решения исходной задачи. Впервые этот метод был предложен Рагаваном (Raghavan) и Томсоном (Thompson) [255], после чего он нашел широкое применение. (Для ознакомления с этой темой см. обзорную статью Мотвани (Motwani), Наора (Naor) и Рагавана [227].) К другим заслуживающим внимания идеям в этой области, предложенным в последнее время, относится метод прямой двойственности (primal dual) (см. обзор [116]), поиск разреженных разрезов (sparse cuts) для использования в алгоритмах разбиения [199], а также применение полуопределенного программирования [115].

Как упоминалось в заключительных замечаниях к главе 34, последние достижения в области вероятностно проверяемых доказательств позволяют найти нижние границы аппроксимируемости многих задач, в том числе некоторых из тех, которые рассматриваются в настоящей главе. В дополнение к приведенным здесь ссылкам заметим, что глава из книги Ароры и Ланда (Lund) [22] содержит хорошее описание отношения между вероятностно проверяемыми доказательствами и степенью сложности приближенных алгоритмов решения различных задач.

ЧАСТЬ VIII

Приложения: математические основы

Введение

Анализ алгоритмов требует использования серьезного математического аппарата. Иногда достаточно знаний из простейшего курса высшей математики, но зачастую используемые в данной книге математические концепции и методы могут оказаться новыми для вас. В части I мы уже познакомились с асимптотическими обозначениями и решением рекуррентных соотношений; в этой части вы найдете ряд других математических концепций и методов, используемых при анализе алгоритмов. Как упоминалось во введении к части I, вы могли быть знакомы со многими рассматриваемыми здесь вопросами еще до того, как приступили к чтению данной книги, так что материал, поданный в приложениях, носит в первую очередь справочный характер. Тем не менее, здесь, как и в обычных главах, приведены упражнения и задачи, которые помогут вам повысить свою квалификацию в рассматриваемых областях математики.

В приложении А рассматриваются методы вычисления и оценки рядов, часто встречающихся при анализе тех или иных алгоритмов. Многие из приводимых здесь формул можно найти в различных учебниках по математике, но гораздо удобнее, когда все эти формулы собраны в одном месте.

В приложении Б приведены основные определения и обозначения, используемые при работе с множествами, отношениями, функциями, графами и деревьями. В этом приложении вы также найдете некоторые основные свойства этих математических объектов.

Приложение В начинается с элементарных принципов комбинаторики — перестановок, сочетаний и т.п. Остальной материал приложения посвящен основам теории вероятности. Большинство алгоритмов в этой книге не требуют использования теории вероятности при анализе, так что вы можете пропустить эту часть приложения. Вы сможете вернуться к нему позже, при желании детально разобраться с вероятностным анализом алгоритмов. В этом случае вы убедитесь, что данное приложение можно рассматривать и как хорошо организованный справочник.

ПРИЛОЖЕНИЕ А

Ряды

Когда алгоритм содержит итеративную управляющую конструкцию, такую как цикл **while** или **for**, время работы можно выразить в виде суммы значений времени выполнения отдельных итераций. Например, в разделе 2.2 указывалось, что j-я итерация алгоритма сортировки вставкой выполняется за время, в худшем случае пропорциональное j. Суммируя значения времени, затраченного на выполнение отдельных итераций, мы получим сумму, или ряд

$$\sum_{j=2}^{n} j.$$

Вычисление этой суммы приводит нас к границе для времени работы алгоритма, равной $\Theta\left(n^2\right)$ в худшем случае. Этот пример свидетельствует о важности понимания и умения вычислять суммы и оценивать их границы.

В разделе А.1 перечислены некоторые основные формулы, связанные с суммированием, а в разделе А.2 вы ознакомитесь с некоторыми методами оценки сумм. Формулы в разделе А.1 приведены без доказательств, однако в разделе А.2 в качестве иллюстрации описываемые здесь методы использованы для доказательства некоторых из формул раздела А.1. Остальные доказательства можно найти в различных учебниках по математике.

A.1 Суммы и их свойства

Для данной последовательности чисел $a_1, a_2, \ldots$ конечная сумма $a_1 + a_2 + \cdots + a_n$, где n — неотрицательное целое число, кратко записывается как

$$\sum_{k=1}^{n} a_k.$$

Если $n = 0$, значение суммы считается равным 0. Значение конечного ряда всегда определено и не зависит от порядка слагаемых.

Для данной последовательности чисел $a_1, a_2, \ldots$ бесконечная сумма $a_1 + a_2 + \ldots$ кратко записывается следующим образом:

$$\sum_{k=1}^{\infty} a_k,$$

что рассматривается как

$$\lim_{n\to\infty} \sum_{k=1}^{n} a_k.$$

Если данный предел не существует, ряд ***расходится*** (diverges); в противном случае ряд ***сходится*** (converges). Члены сходящегося ряда не могут быть суммированы в произвольном порядке. Однако можно переставлять члены ***абсолютно сходящегося ряда***, т.е. ряда $\sum_{k=1}^{\infty} a_k$, для которого сходится также ряд $\sum_{k=1}^{\infty} |a_k|$.

Линейность

Для любого действительного числа c и любых конечных последовательностей $a_1, a_2, \ldots, a_n$ и $b_1, b_2, \ldots, b_n$

$$\sum_{k=1}^{n} (ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k.$$

Свойство линейности справедливо также для бесконечных сходящихся рядов.

Это свойство может использоваться при работе с суммами, в которые входят асимптотические обозначения. Например,

$$\sum_{k=1}^{n} \Theta\left(f\left(k\right)\right) = \Theta\left(\sum_{k=1}^{n} f\left(k\right)\right).$$

В этом уравнении Θ-обозначение в левой части применяется к переменной k, а в правой — к n. Аналогичные действия применимы и к бесконечным сходящимся рядам.

Арифметическая прогрессия

Сумма

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n$$

называется ***арифметической прогрессией*** и равна

$$\sum_{k=1}^{n} k = \frac{1}{2} n\,(n+1) = \tag{A.1}$$

$$= \Theta\left(n^2\right). \tag{A.2}$$

Суммы квадратов и кубов

Для сумм квадратов и кубов справедливы следующие формулы:

$$\sum_{k=0}^{n} k^2 = \frac{n\,(n+1)\,(2n+1)}{6}, \tag{A.3}$$

$$\sum_{k=0}^{n} k^3 = \frac{n^2\,(n+1)^2}{4}. \tag{A.4}$$

Геометрическая прогрессия

Для действительного $x \neq 1$ сумма

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n$$

называется ***геометрической прогрессией*** и равна

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}. \tag{A.5}$$

В случае бесконечного ряда и $|x| < 1$ получается бесконечно убывающая геометрическая прогрессия, сумма которой равна

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}. \tag{A.6}$$

Гармонический ряд

Для натурального n, n-е ***гармоническое число*** представляет собой

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1). \tag{A.7}$$

(Мы докажем это соотношение в разделе A.2.)

Интегрирование и дифференцирование рядов

Новые формулы могут быть получены путем интегрирования или дифференцирования приведенных выше формул. Например, дифференцируя обе части уравнения суммы бесконечной геометрической прогрессии (A.6) и умножая на x, мы получим для $|x| < 1$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}. \tag{A.8}$$

Суммы разностей

Для любой последовательности $a_0, a_1, \ldots, a_n$ справедливо соотношение

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0, \tag{A.9}$$

поскольку каждый из членов $a_1, a_2, \ldots, a_{n-1}$ прибавляется и вычитается ровно один раз. Такие ряды именуют ***телескопическими*** (telescopes). Аналогично,

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n.$$

В качестве примера такого ряда рассмотрим сумму

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)}.$$

Поскольку каждый ее член можно записать как

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1},$$

мы получаем

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = \sum_{k=1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1} \right) = 1 - \frac{1}{n}.$$

Произведения

Конечное произведение $a_1 a_2 \ldots a_n$ может быть кратко записано следующим образом:

$$\prod_{k=1}^{n} a_k.$$

Если $n = 0$, значение произведения считается равным 1. Мы можем преобразовать формулу с произведением в формулу с суммой при помощи тождества

$$\lg\left(\prod_{k=1}^{n} a_k\right) = \sum_{k=1}^{n} \lg a_k.$$

Упражнения

А.1-1. Найдите простое выражение для $\sum_{k=1}^{n} (2k-1)$.

⋆ А.1-2. Покажите, используя формулу для гармонического ряда, что

$$\sum_{k=1}^{n} 1/(2k-1) = \ln\left(\sqrt{n}\right) + O(1).$$

А.1-3. Покажите, что для $0 < |x| < 1$ справедливо соотношение $\sum_{k=0}^{\infty} k^2 x^k = x(1+x)/(1-x)^3$.

⋆ А.1-4. Покажите, что $\sum_{k=0}^{\infty} (k-1)/2^k = 0$.

⋆ А.1-5. Вычислите сумму $\sum_{k=1}^{\infty} (2k+1)\, x^{2k}$.

А.1-6. Используя свойство линейности суммирования, докажите, что

$$\sum_{k=1}^{n} O(f_k(n)) = O\left(\sum_{k=1}^{n} f_k(n)\right).$$

А.1-7. Вычислите произведение $\prod_{k=1}^{n} 2 \cdot 4^k$.

⋆ А.1-8. Вычислите произведение $\prod_{k=2}^{n} \left(1 - 1/k^2\right)$.

А.2 Оценки сумм

Имеется множество методов оценки величин сумм, которые описывают время работы того или иного алгоритма. Здесь мы рассмотрим только наиболее распространенные из них.

Математическая индукция

Основным путем для вычисления сумм рядов является метод математической индукции. В качестве примера докажем, что сумма арифметической прогрессии $\sum_{k=1}^{n} k$ равна $n(n+1)/2$. Можно легко убедиться, что эта формула верна для $n = 1$. Сделаем предположение, что эта формула верна для некоторого n и докажем, что в этом случае она верна и для $n+1$:

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1) = \frac{1}{2} n(n+1) + (n+1) = \frac{1}{2}(n+1)(n+2).$$

Математическая индукция может использоваться не только для точных значений сумм, но и для того, чтобы показать корректность оценок. В качестве примера рассмотрим доказательство того, что сумма геометрической прогрессии $\sum_{k=0}^{n} 3^k$ равна $O(3^n)$. Точнее, докажем, что $\sum_{k=0}^{n} 3^k \leqslant c3^n$ для некоторой константы c. При $n = 0$ формула имеет вид $\sum_{k=0}^{0} 3^k = 1 \leqslant c \cdot 1$, что справедливо при $c \geqslant 1$. Полагая, что граница верна для n, докажем ее справедливость для $n+1$. Имеем:

$$\sum_{k=0}^{n+1} 3^k = \sum_{k=0}^{n} 3^k + 3^{n+1} \leqslant c3^n + 3^{n+1} = \left(\frac{1}{3} + \frac{1}{c}\right) c3^{n+1} \leqslant c3^{n+1},$$

что справедливо при $(1/3 + 1/c) \leqslant 1$ или, что то же самое, при $c \geqslant 3/2$. Таким образом, $\sum_{k=0}^{n} 3^k = O(3^n)$, что и требовалось показать.

При использовании асимптотических обозначений для доказательства по индукции следует быть предельно внимательным и осторожным. Рассмотрим следующее "доказательство" того, что $\sum_{k=1}^{n} k = O(n)$. Очевидно, что $\sum_{k=1}^{1} k = O(1)$. Исходя из справедливости оценки для n, докажем ее для $n+1$:

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1) \overset{\textit{Неверно!!}}{=} O(n) + O(n+1) = O(n+1).$$

Ошибка в том, что "константа", скрытая в $O(n)$, растет вместе с n, а значит, константой не является. Мы не смогли показать, что одна и та же константа работает для *всех* n.

Почленное сравнение

Иногда можно получить неплохую верхнюю оценку ряда, заменив каждый его член бóльшим (иногда для этого можно воспользоваться наибольшим членом). Например, вот как можно оценить верхнюю границу арифметической прогрессии (A.1):

$$\sum_{k=1}^{n} k \leqslant \sum_{k=1}^{n} n = n^2.$$

В общем случае

$$\sum_{k=1}^{n} a_k \leqslant n a_{\max},$$

где $a_{\max} = \max_{1 \leqslant k \leqslant n} a_k$.

Если ряд в действительности может быть ограничен геометрической прогрессией, можно получить более точную оценку его суммы. Итак, пусть ряд $\sum_{k=0}^{n} a_k$ обладает тем свойством, что для всех $k \geqslant 0$ $a_{k+1}/a_k \leqslant r$, где $0 < r < 1$ — некоторая константа. В таком случае, т.к. $a_k \leqslant a_0 r^k$, сумма ряда ограничивается сверху суммой бесконечной геометрической прогрессии:

$$\sum_{k=0}^{n} a_k \leqslant \sum_{k=0}^{\infty} a_0 r^k = a_0 \sum_{k=0}^{\infty} r^k = \frac{a_0}{1-r}.$$

Применим этот метод, например, к ряду $\sum_{k=1}^{\infty} \left(k/3^k\right)$. Для того чтобы сумма начиналась с $k = 0$, перепишем ряд как $\sum_{k=0}^{\infty} \left((k+1)/3^{k+1}\right)$. Первый член a_0 этого ряда равен $1/3$, а отношение соседних членов ряда r для всех $k \geqslant 0$ равно

$$\frac{(k+2)/3^{k+2}}{(k+1)/3^{k+1}} = \frac{1}{3} \cdot \frac{k+2}{k+1} \leqslant \frac{2}{3}.$$

Таким образом, мы получаем следующую оценку:

$$\sum_{k=1}^{\infty} \frac{k}{3^k} = \sum_{k=0}^{\infty} \frac{k+1}{3^{k+1}} \leqslant \frac{1}{3} \cdot \frac{1}{1-2/3} = 1.$$

Распространенной ошибкой при использовании этого метода является выяснение того, что отношение двух последовательных членов ряда меньше 1 и на этом основании делается вывод об ограниченности ряда геометрической прогрессией. В качестве контрпримера можно привести гармонический ряд, который расходится, так как

$$\sum_{k=1}^{\infty} \frac{1}{k} = \lim_{n \to \infty} \sum_{k=1}^{n} \frac{1}{k} = \lim_{n \to \infty} \Theta\left(\lg n\right) = \infty.$$

Несмотря на то, что отношение $(k+1)$-го и k-го членов этого ряда равно $k/(k+1) < 1$, ряд расходится. Для того чтобы ряд был ограничен геометрической прогрессией, надо показать наличие *константы* $r < 1$, такой что отношение двух соседних членов никогда не превосходит значения r. В гармоническом ряде такой константы не существует, поскольку отношение двух соседних членов может быть сколь угодно близким к 1.

Разбиение рядов

Еще один способ получения оценки сложной суммы состоит в представлении ряда как суммы двух или большего числа рядов путем разделения на части всего диапазона индексов, и оценке сумм частей по отдельности. Предположим, например, что мы ищем нижнюю границу арифметической прогрессии $\sum_{k=1}^{n} k$, для которой, как мы уже выяснили, верхняя граница равна $O\left(n^2\right)$. Можно попытаться ограничить каждый член суммы наименьшим, но поскольку наименьший член этого ряда — 1, мы получим в качестве нижней границы n, что слишком далеко от найденной верхней границы.

Лучший результат для нижней границы можно получить, если сначала разбить ряд на две части. Предположим для удобства, что n — четное число. Тогда

$$\sum_{k=1}^{n} k = \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k \geqslant \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} (n/2) = (n/2)^2 = \Omega\left(n^2\right),$$

что является асимптотически точной оценкой, поскольку $\sum_{k=1}^{n} k = O\left(n^2\right)$.

В рядах, рассматриваемых при анализе алгоритмов, зачастую можно разбить ряд на части и просто проигнорировать некоторое конечное число начальных членов. Обычно этот метод применим, если каждый член a_k ряда $\sum_{k=0}^{n} a_k$ не зависит от n. Тогда для любой константы $k_0 > 0$ можно записать

$$\sum_{k=0}^{n} a_k = \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^{n} a_k = \Theta(1) + \sum_{k=k_0}^{n} a_k,$$

поскольку начальные члены суммы — постоянные величины, и в отдельный ряд выделено постоянное их количество. После такого разделения для поиска границ ряда $\sum_{k=k_0}^{n} a_k$ можно использовать другие методы. Описанная методика применима и к бесконечным рядам. Например, для поиска асимптотической верхней границы ряда

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k}$$

заметим, что отношение последовательных членов ряда

$$\frac{(k+1)^2/2^{k+1}}{k^2/2^k} = \frac{(k+1)^2}{2k^2} \leqslant \frac{8}{9},$$

при $k \geqslant 3$. Таким образом, можно оценить сумму исходного ряда следующим образом:

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k} = \sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \leqslant \sum_{k=0}^{2} \frac{k^2}{2^k} + \frac{9}{8} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k = O(1),$$

поскольку количество членов первой суммы — константа, а вторая представляет собой бесконечно убывающую геометрическую прогрессию.

Метод разбиения ряда может быть применен для определения асимптотических границ в гораздо более сложных ситуациях. Например, таким способом мы можем получить границу $O(\lg n)$ гармонического ряда (А.7):

$$H_n = \sum_{k=1}^{n} \frac{1}{k}.$$

Идея заключается в разбиении диапазона от 1 до n на $\lfloor \lg n \rfloor + 1$ частей, с ограничением каждой части единицей. Каждая часть состоит из членов, начинающихся от $1/2^i$ и заканчивающихся членом $1/2^{i+1}$ (исключая сам этот член), что дает нам

$$\sum_{k=1}^{n} \frac{1}{k} \leqslant \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i + j} \leqslant \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i} = \sum_{i=0}^{\lfloor \lg n \rfloor} 1 \leqslant \lg n + 1. \qquad \text{(А.10)}$$

Приближение интегралами

Если сумма может быть выражена как $\sum_{k=m}^{n} f(k)$, где $f(k)$ — монотонно возрастающая функция, мы можем оценить значение ряда при помощи интегралов:

$$\int_{m-1}^{n} f(x)\,dx \leqslant \sum_{k=m}^{n} f(k) \leqslant \int_{m}^{n+1} f(x)\,dx. \qquad \text{(А.11)}$$

Пояснение такой оценки показано на рис. А.1. На рисунке в прямоугольниках показаны их площади, а общая площадь прямоугольников представляет значение суммы. Значение интеграла равно заштрихованной площади под кривой. На рис. А.1*а* показано, что $\int_{m-1}^{n} f(k) \leqslant \sum_{k=m}^{n} f(k)$, а на рис. А.1*б* — что $\sum_{k=m}^{n} f(k) \leqslant \int_{m}^{n+1} f(x)\,dx$.

Если функция $f(k)$ монотонно убывающая, то аналогично можно показать, что

$$\int_{m}^{n+1} f(x)\,dx \leqslant \sum_{k=m}^{n} f(k) \leqslant \int_{m-1}^{n} f(x)\,dx. \qquad \text{(А.12)}$$

Приближение интегралами (А.12) дает нам точную асимптотическую оценку n-го гармонического числа. Нижнюю границу мы получаем следующим образом:

$$\sum_{k=1}^{n} \frac{1}{k} \geqslant \int_{1}^{n+1} \frac{dx}{x} = \ln(n+1). \qquad \text{(А.13)}$$

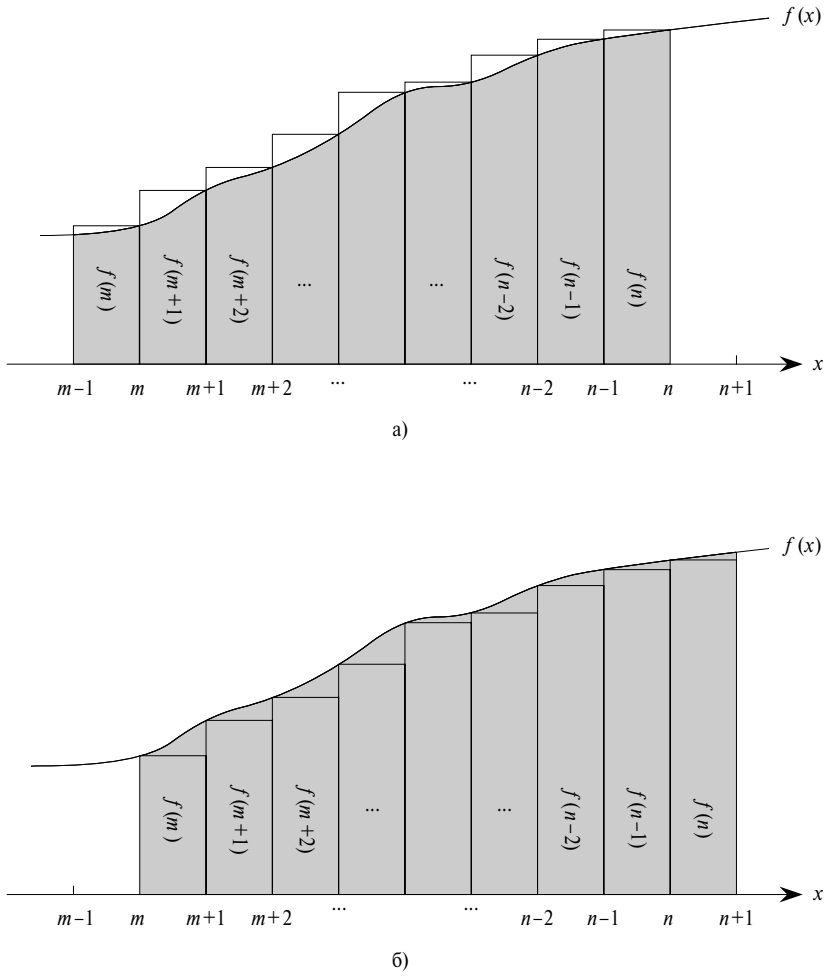


Рис. А.1. Приближение $\sum_{k=m}^{n} f\,(k)$ интегралами

Для получения верхней границы воспользуемся неравенством

$$\sum_{k=2}^{n} \frac{1}{k} \leqslant \int_{1}^{n} \frac{dx}{x} = \ln n,$$

откуда, прибавляя первый член ряда, находим, что

$$\sum_{k=1}^{n} \frac{1}{k} \leqslant \ln n + 1. \tag{A.14}$$

Упражнения

А.2-1. Покажите, что сумма $\sum_{k=1}^{n} 1/k^2$ ограничена сверху константой.

А.2-2. Найдите асимптотическую верхнюю границу суммы $\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil$.

А.2-3. Применяя разбиение ряда, покажите, что n-я частичная сумма гармонического ряда есть $\Omega(\lg n)$.

А.2-4. Найдите приближенное значение $\sum_{k=1}^{n} k^3$ при помощи интегралов.

А.2-5. Почему мы не можем применить интегральное приближение (А.12) для поиска верхней границы n-го гармонического числа непосредственно к сумме $\sum_{k=1}^{n} 1/k$?

Задачи

А-1. **Оценки сумм**

Дайте асимптотически точные оценки приведенных ниже сумм. Считаем, что $r \geqslant 0$ и $s \geqslant 0$ — константы.

а) $\sum_{k=1}^{n} k^r$.

б) $\sum_{k=1}^{n} \lg^s k$.

в) $\sum_{k=1}^{n} k^r \lg^s k$.

Заключительные замечания

Книга Кнута (Knuth) [182] — отличное справочное пособие по изложенному здесь материалу. Основные свойства рядов можно найти во множестве книг, например, в книге Апостола (Apostol) [18] или Томаса (Thomas) и Финни (Finney) [296].

ПРИЛОЖЕНИЕ Б

Множества и прочие художества

Во многих главах книги нам приходится сталкиваться с элементами дискретной математики. Здесь вы более подробно ознакомитесь с обозначениями, определениями и элементарными свойствами множеств, отношений, функций, графов и деревьев. Читатели, знакомые с этим материалом, могут пропустить данное приложение.

Б.1 Множества

Множество (set) представляет собой набор различимых объектов, которые называются ***членами*** или ***элементами***. Если объект x является членом множества S, мы записываем это как $x \in S$ (читается "x принадлежит S"). Если x не принадлежит S, мы записываем $x \notin S$. Множество можно описать путем явного перечисления его элементов в виде списка, заключенного в фигурные скобки. Например, мы можем определить множество S как содержащее числа 1, 2 и 3, и только их, записав $S = \{1, 2, 3\}$. Поскольку 2 является элементом множества S, мы можем записать $2 \in S$, а так как 4 не является элементом S, верна запись $4 \notin S$. Множество не может содержать один и тот же элемент дважды[1]; кроме того, элементы множества не упорядочены. Два множества A и B ***равны*** (что записывается как $A = B$), если они содержат одни и те же элементы. Например, $\{1, 2, 3\} = \{3, 2, 1\}$.

Для часто встречающихся множеств используются специальные обозначения:

[1]Модификация множества, которое может содержать несколько одинаковых элементов, называется **мультимножеством** (multiset).

- $\emptyset$ обозначает ***пустое множество***, т.е. множество, не содержащее ни одного элемента;
- **Z** обозначает множество ***целых чисел***, т.е. множество $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$;
- **R** обозначает множество ***действительных чисел***;
- **N** обозначает множество ***натуральных чисел***, т.е. множество $\{1, 2, 3, \ldots\}$[2].

Если все элементы множества A содержатся во множестве B, т.е. из $x \in A$ вытекает $x \in B$, то мы записываем, что $A \subseteq B$ и говорим, что множество A является ***подмножеством*** B. Множество A является ***истинным подмножеством*** (proper subset) множества B (что записывается как $A \subset B$), если $A \subseteq B$, но $A \neq B$. (Многие авторы используют обозначение $A \subset B$ не только для истинных подмножеств, но и для отношения обычного подмножества.) Для любого множества A справедливо $A \subseteq A$; для двух множеств A и B $A = B$ тогда и только тогда, когда $A \subseteq B$ и $B \subseteq A$. Для любых трех множеств A, B и C из $A \subseteq B$ и $B \subseteq C$ следует $A \subseteq C$. Для любого множества A справедливо соотношение $\emptyset \subseteq A$.

Иногда множество определяется посредством другого множества. Так, имея множество A, мы можем определить множество $B \subseteq A$, указав свойство, которым обладают элементы множества B. Например, множество четных чисел можно определить следующим образом: $\{x : x \in \mathbf{Z} \text{ и } x/2 - \text{целое число}\}$. Двоеточие в такой записи читается как “такой что” (некоторые авторы вместо двоеточия используют вертикальную черту).

Для данных двух множеств A и B можно определить новые множества при помощи следующих ***операций над множествами***.

- ***Пересечением*** множеств A и B является множество

$$A \cap B = \{x : x \in A \text{ и } x \in B\}.$$

- ***Объединением*** множеств A и B является множество

$$A \cup B = \{x : x \in A \text{ или } x \in B\}.$$

- ***Разностью*** множеств A и B является множество

$$A - B = \{x : x \in A \text{ и } x \notin B\}.$$

Операции над множествами обладают следующими свойствами.

- ***Свойства пустого множества***

$$A \cap \emptyset = \emptyset,$$
$$A \cup \emptyset = A.$$

[2] В зарубежной литературе (в частности, в американской) под натуральными числами подразумеваются числа $0, 1, 2, \ldots$. Здесь мы придерживаемся принятого в отечественной математике понятия натуральных чисел. — *Прим. ред.*

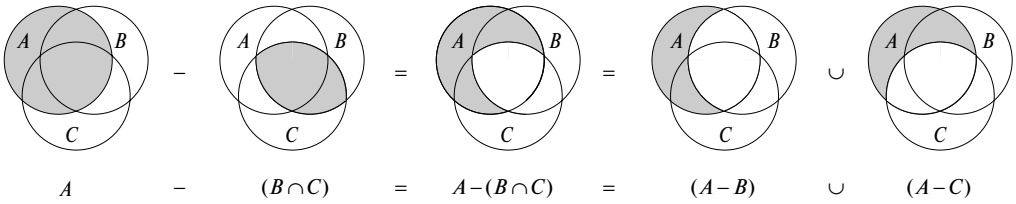


Рис. Б.1. Диаграмма Венна, иллюстрирующая первый закон де Моргана

- ***Свойства идемпотентности***

$$A \cap A = A,$$
$$A \cup A = A.$$

- ***Свойства коммутативности***

$$A \cap B = B \cap A,$$
$$A \cup B = B \cup A.$$

- ***Свойства ассоциативности***

$$A \cap (B \cap C) = (A \cap B) \cap C,$$
$$A \cup (B \cup C) = (A \cup B) \cup C.$$

- ***Свойства дистрибутивности***

$$\begin{aligned} A \cap (B \cup C) &= (A \cap B) \cup (A \cap C) \\ A \cup (B \cap C) &= (A \cup B) \cap (A \cup C)\,. \end{aligned} \qquad \text{(Б.1)}$$

- ***Свойства поглощения***

$$A \cap (A \cup B) = A,$$
$$A \cup (A \cap B) = A.$$

- ***Законы де Моргана***

$$\begin{aligned} A - (B \cap C) &= (A - B) \cup (A - C)\,, \\ A - (B \cup C) &= (A - B) \cap (A - C)\,. \end{aligned} \qquad \text{(Б.2)}$$

Первый закон де Моргана проиллюстрирован на рис. Б.1 при помощи ***диаграммы Венна***, графического представления множеств в виде областей на плоскости.

Часто все рассматриваемые множества являются подмножествами некоторого большего множества U, называемого ***универсумом*** (universe). Например, если мы

работаем с различными множествами целых чисел, то в качестве универсума можно рассматривать множество **Z**. Для данного универсума U можно определить ***дополнение*** (complement) множества A как $\bar{A} = U - A$. Для любого множества $A \subseteq U$ выполняются следующие соотношения:

$$\begin{aligned} \overline{\bar{A}} &= A, \\ A \cap \bar{A} &= \emptyset, \\ A \cup \bar{A} &= U. \end{aligned}$$

Из законов де Моргана следует, что для любых двух множеств $B, C \subseteq U$ имеют место равенства

$$\begin{aligned} \overline{B \cap C} &= \overline{B} \cup \overline{C}, \\ \overline{B \cup C} &= \overline{B} \cap \overline{C}. \end{aligned}$$

Множества A и B являются ***непересекающимися*** (disjoint), если они не имеют общих элементов, т.е. если $A \cap B = \emptyset$. Семейство $\mathcal{S} = \{S_i\}$ непустых множеств образует ***разбиение*** (partition) множества S, если

- множества ***попарно не пересекаются***, т.е. из $S_i, S_j \in \mathcal{S}$ и $i \neq j$ следует $S_i \cap S_j = \emptyset$, и
- их объединение равно S, т.е.

$$S = \bigcup_{S_i \in \mathcal{S}} S_i.$$

Другими словами, семейство $\mathcal{S}$ образует разбиение множества S, если любой элемент $s \in S$ принадлежит в точности одному из множеств $S_i \in \mathcal{S}$.

Количество элементов множества называется его ***мощностью*** (cardinality) или размером, и обозначается как $|S|$. Два множества имеют одинаковую мощность, если между их элементами можно установить взаимно однозначное соответствие. Мощность пустого множества равна $0 : |\emptyset| = 0$. Если мощность множества представляет собой целое неотрицательное число, то такое множество называется ***конечным***; в противном случае множество является ***бесконечным***. Бесконечное множество, элементы которого могут быть поставлены во взаимно однозначное соответствие натуральным числам, называются ***счетными*** (countably infinite), в противном случае мы имеем дело с ***несчетными*** (uncountably) множествами. Множество целых чисел **Z** счетно, в то время как множество действительных чисел **R** — нет.

Для любых двух конечных множеств A и B справедливо следующее тождество:

$$|A \cup B| = |A| + |B| - |A \cap B|, \tag{Б.3}$$

из которого следует неравенство

$$|A \cup B| \leqslant |A| + |B| .$$

Если множества A и B не пересекаются, то $|A \cap B| = 0$ и, следовательно, $|A \cup B| = |A| + |B|$. Если $A \subseteq B$, то $|A| \leqslant |B|$.

Конечное множество из n элементов называется ***n-элементным***. В англоязычной литературе одноэлементное множество имеет специальное название ***singleton***, а подмножество из k элементов именуется ***k-subset***.

Множество всех подмножеств множества S, включая пустое подмножество и само множество S, обозначается как 2^S и называется ***степенным множеством*** (power set) S. Например, $2^{\{a,b\}} = \{\emptyset, \{a\}, \{b\}, \{a,b\}\}$. Мощность степенного множества конечного множества S имеет мощность $2^{|S|}$.

Иногда мы сталкиваемся с множествоподобными структурами, элементы которых упорядочены. ***Упорядоченная пара*** (ordered pair) состоит из двух элементов a и b, обозначается как (a, b) и формально может быть определена как $\{a, \{a, b\}\}$. Упорядоченные пары (a, b) и (b, a) различны.

Декартово произведение (Cartesian product) двух множеств A и B обозначается $A \times B$ и представляет собой множество всех упорядоченных пар, в которых первый элемент пары является элементом A, а второй — элементом B. Говоря более строго,

$$A \times B = \{(a, b) : a \in A \text{ и } b \in B\} .$$

Например, $\{a, b\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)\}$. Если A и B являются конечными множествами, то мощность их декартова произведения равна

$$|A \times B| = |A| \cdot |B| . \qquad \text{(Б.4)}$$

Декартово произведение n множеств $A_1, A_2, \ldots, A_n$ представляет собой множество кортежей из n элементов (***n-tuples***)

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) : a_i \in A_i, i = 1, 2, \ldots, n\} ,$$

мощность которого в случае, если все множества конечны, равна

$$|A_1 \times A_2 \times \cdots \times A_n| = |A_1| \cdot |A_2| \cdots |A_n| .$$

Можно также определить декартову степень как следующее множество:

$$A^n = A \times A \times \cdots \times A,$$

мощность которого в случае конечности множества A составляет $|A^n| = |A|^n$.

Упражнения

Б.1-1. Изобразите диаграмму Венна, которая иллюстрирует первое свойство дистрибутивности (Б.1).

Б.1-2. Докажите обобщение закона де Моргана для любого конечного набора множеств:

$$\overline{A_1 \cap A_2 \cap \ldots \cap A_n} = \overline{A_1} \cup \overline{A_2} \cup \ldots \cup \overline{A_n},$$
$$\overline{A_1 \cup A_2 \cup \ldots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \ldots \cap \overline{A_n}.$$

⋆ Б.1-3. Докажите обобщение равенства (Б.3), именуемое ***принципом включений и исключений*** (principle of inclusion and exclusion):

$$\begin{aligned} |A_1 \cup A_2 \cup \cdots \cup A_n| = \\ = |A_1| + |A_2| + \cdots + |A_n| - \\ - |A_1 \cap A_2| - |A_1 \cap A_3| - \cdots + \qquad \text{(все пары)} \\ + |A_1 \cap A_2 \cap A_3| + \cdots + \qquad \text{(все тройки)} \\ \vdots \\ + (-1)^{n-1} |A_1 \cap A_2 \cap \ldots \cap A_n| . \end{aligned}$$

Б.1-4. Покажите, что множество нечетных натуральных чисел счетно.

Б.1-5. Покажите, что для любого конечного множества S степенное множество 2^S содержит $2^{|S|}$ элементов (т.е. имеется $2^{|S|}$ различных подмножеств множества S).

Б.1-6. Дайте индуктивное определение кортежа из n элементов, используя понятие упорядоченной пары.

Б.2 Отношения

Бинарным отношением (binary relation) R между элементами двух множеств A и B называется подмножество декартова произведения $A \times B$. Если $(a, b) \in \in R$, мы можем записать это как $a\ R\ b$. Когда мы говорим, что R есть бинарное отношение на множестве A, имеется в виду, что R является подмножеством $A \times A$. Например, отношение "меньше" на множестве натуральных чисел представляет собой множество $\{(a, b) : a, b \in \mathbf{N} \text{ и } a < b\}$. Под n-арным отношением на множествах $A_1, A_2, \ldots, A_n$ понимается подмножество декартова произведения $A_1 \times A_2 \times \ldots \times A_n$.

Бинарное отношение $R \subseteq A \times A$ является ***рефлексивным***, если для всех $a \in A$ справедливо $a\ R\ a$. Например, отношения "$=$" или "$\leqslant$" на множестве $\mathbf{N}$ рефлексивны, но отношение "$<$" таковым не является.

Отношение R ***симметрично***, если из $a\ R\ b$ вытекает $b\ R\ a$ для всех $a, b \in A$. Например, симметричным является отношение "$=$", но не "$<$" или "$\leqslant$".

Отношение R ***транзитивно***, если из $a\ R\ b$ и $b\ R\ c$ следует $a\ R\ c$ для всех $a, b, c \in A$. Например, транзитивными являются отношения "$=$", "$<$" и "$\leqslant$", но отношение $R = \{(a, b) : a, b \in \mathbf{N}$ и $a = b - 1\}$ таковым не является, поскольку из $3\ R\ 4$ и $4\ R\ 5$ не следует $3\ R\ 5$.

Отношение, которое является одновременно рефлексивным, симметричным и транзитивным, называется ***отношением эквивалентности***. Например, на множестве натуральных чисел отношение "$=$" является отношением эквивалентности, а отношение "$<$" — нет. Если R — отношение эквивалентности на множестве A, то можно определить ***класс эквивалентности*** элемента $a \in A$ как множество $[a] = \{b \in A : a\ R\ b\}$, т.е. множество всех элементов, эквивалентных a. Например, если мы определим отношение $R = \{(a, b) : a, b \in \mathbf{N}$ и $(a + b) -$ четное число$\}$, то R будет отношением эквивалентности, поскольку $a+a$ четно (рефлексивность), если четно $a + b$, то четно и $b + a$ (симметричность), и из четности $a + b$ и $b + c$ вытекает четность $a + c$ (транзитивность). Класс эквивалентности числа 4 есть $[4] = \{2, 4, 6, \ldots\}$, а класс эквивалентности числа 3 — $[3] = \{1, 3, 5, \ldots\}$. Основная теорема о классах эквивалентности заключается в следующем.

Теорема Б.1 (Отношения эквивалентности тождественны разбиениям). Для любого отношения эквивалентности на множестве A классы эквивалентности образуют разбиение A, и обратно, любое разбиение A определяет отношение эквивалентности на A, для которого множества в разбиении являются классами эквивалентности. ■

Доказательство. Для доказательства первого утверждения теоремы надо показать, что классы эквивалентности R непусты, попарно не пересекаются и их объединение дает множество A. Из рефлексивности R вытекает $a \in [a]$, так что класс эквивалентности не является пустым; кроме того, поскольку каждый элемент $a \in A$ является элементом класса эквивалентности $[a]$, объединение всех классов эквивалентности равно A. Остается показать, что классы эквивалентности попарно не пересекаются, т.е. что если два класса эквивалентности $[a]$ и $[b]$ имеют общий элемент c, то это один и тот же класс эквивалентности. В этом случае $a\ R\ c$ и $b\ R\ c$, откуда в силу симметричности и транзитивности $a\ R\ b$. Таким образом, для произвольного элемента $x \in [a]$ имеем $x\ R\ a$ и, как следствие, $x\ R\ b$, так что $[a] \subseteq [b]$. Аналогично, $[b] \subseteq [a]$, так что $[a] = [b]$.

Для доказательства второй части теоремы рассмотрим разбиение $A\ \mathcal{A} = \{A_i\}$, и определим следующее отношение:

$$R = \{(a, b) :\ \text{существует такое } i, \text{ что } a \in A_i \text{ и } b \in A_i\}\,.$$

Покажем, что R есть отношение эквивалентности на A. Это отношение рефлексивно, т.е. из $a \in A_i$ следует $a\ R\ a$. Симметричность R следует из того, что если

$a\ R\ b$, то и a, и b принадлежат одному и тому же множеству A_i, так что $b\ R\ a$. Если $a\ R\ b$ и $b\ R\ c$, то все три элемента принадлежат одному и тому же множеству, так что $a\ R\ c$, т.е. отношение R транзитивно. Чтобы показать, что множества в разбиении являются классами эквивалентности R, заметим, что если $a \in A_i$, то из $x \in [a]$ вытекает $x \in A_i$, а из $x \in A_i$ вытекает $x \in [a]$. ■

Бинарное отношение R на множестве A является ***антисимметричным***, если из $a\ R\ b$ и $b\ R\ a$ следует $a = b$. Например, антисимметричным на множестве натуральных чисел является отношение "$\leqslant$", поскольку если $a \leqslant b$ и $b \leqslant a$, то $a = b$. Отношение, являющееся одновременно рефлексивным, антисимметричным и транзитивным, является и отношением ***частичного порядка*** (partial order), а множество, на котором определено такое отношение, — ***частично упорядоченным множеством***. Например, отношение "быть потомком" на множестве людей является отношением частичного порядка (если рассматривать человека как собственного потомка).

В частично упорядоченном множестве A может не быть единственного "наибольшего" элемента a, такого что $b\ R\ a$ для всех $b \in A$. Вместо этого могут быть несколько ***максимальных*** элементов a, обладающих тем свойством, что не существует таких $b \in A$, отличных от a, что $a\ R\ b$. Например, в наборе ящиков разного размера может быть несколько максимальных ящиков, которые не могут поместиться ни в один другой ящик, так что одного "наибольшего" ящика, в котором могут поместиться все остальные, не существует[3].

Отношение частичного порядка R является отношением ***полного*** (total order), или ***линейного*** (linear order), порядка, если для всех $a, b \in A$ имеем $a\ R\ b$ или $b\ R\ a$, т.е. если любая пара элементов A может быть связана отношением R. Например, отношение "$\leqslant$" на множестве натуральных чисел является отношением линейного порядка, в то время как отношение "быть потомком" на множестве людей таковым не является, поскольку могут быть люди, которые не являются потомками друг друга.

Упражнения

Б.2-1. Докажите, что отношение подмножества "$\subseteq$" на множестве всех подмножеств $\mathbf{Z}$ является отношением частичного, но не полного порядка.

Б.2-2. Покажите, что для любого положительного n отношение "тождественности по модулю n" является отношением эквивалентности на множестве целых чисел. (Мы говорим, что $a \equiv b \,(\mathrm{mod}\, n)$, если существует такое целое число q, что $a - b = qn$.) На сколько классов эквивалентности разбивает множество целых чисел это отношение?

[3] Для того чтобы отношение "может поместиться в" было отношением частичного порядка, надо считать, что ящик может поместиться сам в себя.

Б.2-3. Приведите пример отношения, которое

а) рефлексивно и симметрично, но не транзитивно;

б) рефлексивно и транзитивно, но не симметрично;

в) симметрично и транзитивно, но не рефлексивно.

Б.2-4. Пусть S — конечное множество, а R — отношение эквивалентности на $S \times S$. Покажите, что если R — антисимметрично, то все классы эквивалентности содержат по одному элементу.

Б.2-5. Профессор полагает, что всякое симметричное и транзитивное отношение R должно быть рефлексивным. Он предлагает следующее доказательство: из $a\ R\ b$ в силу симметрии следует $b\ R\ a$, а применение транзитивности к этому выводу дает $a\ R\ a$. Прав ли профессор?

Б.3 Функции

Для данных двух множеств A и B ***функция*** f представляет собой бинарное отношение на $A \times B$, такое что для каждого $a \in A$ существует ровно одно $b \in B$, такое что $(a, b) \in f$. Множество A называется ***областью определения*** (domain) функции f, а множество B — ***областью значений*** (codomain). Иногда для функции используется запись $f : A \to B$; кроме того, если $(a, b) \in f$, то мы записываем $b = f(a)$, поскольку значение b однозначно определяется по значению a.

Интуитивно функцию f можно рассматривать как операцию, которая ставит в соответствие каждому элементу A элемент B. Никакому элементу A не может быть поставлено в соответствие два различных элемента B, однако один и тот же элемент B может соответствовать разным элементам A. Например, бинарное отношение

$$f = \{(a, b) : a, b \in \mathbf{N} \text{ и } b = a \bmod 2\}$$

является функцией $f : \mathbf{N} \to \{0, 1\}$, поскольку для каждого натурального числа a имеется ровно одно число b из $\{0, 1\}$, такое что $b = a \bmod 2$. Например, $0 = f(0)$, $1 = f(1)$, $0 = f(2)$ и т.д. Бинарное же отношение

$$g = \{(a, b) : a, b \in \mathbf{N} \text{ и } a + b - \text{четно}\}$$

функцией не является, поскольку и $(1, 3)$, и $(1, 5)$ являются элементами g, так что элементу $a = 1$ соответствуют более одного элемента b, такого что $(a, b) \in g$.

Если у нас имеется функция $f : A \to B$, и $b = f(a)$, то a называется ***аргументом*** функции f, а b — ***значением*** функции f от данного аргумента a. Можно определить функцию, указывая ее значения для каждого элемента из области определения. Например, можно определить $f(n) = 2n$ для $n \in \mathbf{N}$, что означает $f = \{(n, 2n) : n \in \mathbf{N}\}$. Две функции f и g называются ***равными***, если у них

одинаковые области определения и значений и если для любого a из области определения $f(a) = g(a)$.

Конечная последовательность длины n представляет собой функцию f, область определения которой — множество из n целых чисел $\{0, 1, \ldots, n-1\}$. Зачастую конечная последовательность записывается как список ее значений: $\langle f(0), f(1), \ldots, f(n-1)\rangle$. ***Бесконечной последовательностью*** называется функция, область определения которой — целые неотрицательные числа. Например, последовательность чисел Фибоначчи, определенная рекуррентным соотношением (3.21), представляет собой бесконечную последовательность $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\rangle$.

Если область определения функции f является декартовым произведением, дополнительные скобки вокруг аргументов в записи обычно опускаются. Например, если у нас есть функция $f : A_1 \times A_2 \times \cdots \times A_n \to B$, то мы можем записать $b = f(a_1, a_2, \ldots, a_n)$, а не $b = f((a_1, a_2, \ldots, a_n))$. Кроме того, в этом случае ***аргументом*** называется каждый элемент a_i, хотя технически единым аргументом функции f является кортеж из n элементов $(a_1, a_2, \ldots, a_n)$.

Если $b = f(a)$ для некоторой функции $f : A \to B$, то иногда говорят, что b есть ***образ*** (image) a. Образ подмножества $A' \subseteq A$ определяется как

$$f(A') = \{b \in B : b = f(a) \text{ для некоторого } a \in A'\}.$$

Множество значений (range) функции f представляет собой образ ее области определения, т.е. $f(A)$. Например, множеством значений функции $f : \mathbf{N} \to \mathbf{N}$, определенной как $f(n) = 2n$, является

$$f(\mathbf{N}) = \{m : m = 2n \text{ для некоторого } n \in \mathbf{N}\}.$$

Функция называется ***наложением*** или ***сюръекцией*** (surjection), если ее множество значений совпадает с областью значений. Например, функция $f(n) = \lfloor n/2 \rfloor$ представляет собой наложение множества неотрицательных целых чисел на множество неотрицательных целых чисел, так как каждый элемент этого множества служит значением f для некоторого аргумента. Функция же $f(n) = 2n$ не является наложением $\mathbf{N}$ на $\mathbf{N}$, поскольку, например, число 3 не является значением f ни для какого аргумента; однако эта функция является наложением $\mathbf{N}$ на множество четных натуральных чисел. Сюръекцию $f : A \to B$ иногда называют ***отображением*** A ***на*** B.

Функция $f : A \to B$ называется ***вложением*** или ***инъекцией*** (injection), если различным аргументам f соответствуют различные значения, т.е. из $a \neq a'$ вытекает $f(a) \neq f(a')$. Например, функция $f(n) = 2n$ является вложением множества $\mathbf{N}$ в $\mathbf{N}$, так как каждое четное число b является отображением не более одного элемента из области определения, а именно $b/2$. Функция $f(n) = \lfloor n/2 \rfloor$ вложением не является, поскольку значение 1, например, получается для двух

аргументов — 2 и 3. Инъекции в англоязычной литературе иногда называются функциями однозначного соответствия (one-to-one function).

Функция $f : A \to B$ называется ***биекцией*** (bijection), если она является одновременно инъекцией и сюръекцией. Например, функция $f(n) = (-1)^n \lceil n/2 \rceil$ является биекцией, отображающей множество неотрицательных целых чисел на множество целых чисел:

$$\begin{aligned} 0 &\to 0 \\ 1 &\to -1 \\ 2 &\to 1 \\ 3 &\to -2 \\ 4 &\to 2 \\ &\vdots \end{aligned}$$

Данная функция является инъекцией, поскольку ни один элемент множества целых чисел не является образом более одного аргумента. Она также является сюръекцией, поскольку каждое целое число является образом некоторого элемента множества неотрицательных целых чисел. Следовательно, рассматриваемая функция является биекцией. Биекции называют также ***взаимно однозначными соответствиями*** (one-to-one correspondence), поскольку они делят все элементы областей определения и значений на пары. Биекция множества A относительно себя самого иногда называется ***перестановкой*** множества A.

Если функция f является биекцией, ***обратная*** (inverse) к ней функция определяется следующим образом:

$$f^{-1}(b) = a \text{ тогда и только тогда, когда } f(a) = b.$$

Например, вот функция, обратная к $f(n) = (-1)^n \lceil n/2 \rceil$:

$$f^{-1}(m) = \begin{cases} 2m & \text{если } m \geqslant 0, \\ -2m - 1 & \text{если } m < 0. \end{cases}$$

Упражнения

Б.3-1. Пусть A и B — конечные множества, а $f : A \to B$ — некоторая функция. Покажите, что

а) если f — инъекция, то $|A| \leqslant |B|$;

б) если f — сюръекция, то $|A| \geqslant |B|$.

Б.3-2. Пусть имеется функция $f(x) = x + 1$. Будет ли она биекцией, если ее область определения и область значений — натуральные числа? А если ее область определения и область значений — целые числа?

Б.3-3. Дайте определение обратного к бинарному отношению. (Если отношение является биекцией, то определение должно давать обратную функцию.)

⋆ Б.3-4. Приведите пример биекции $f : \mathbf{Z} \to \mathbf{Z} \times \mathbf{Z}$.

Б.4 Графы

В этом разделе будут рассмотрены два типа графов — ориентированные и неориентированные. Следует иметь в виду, что терминологию в этой области еще нельзя назвать вполне устоявшейся, так что в литературе можно встретить определения, отличающиеся от приведенных здесь, хотя в основном эти отличия незначительны. Вопрос о представлении графов в памяти компьютера рассматривался в разделе 22.1.

Ориентированный граф (directed graph) G определяется как пара (V, E), где V — конечное множество, а E — бинарное отношение на V. Множество V называется ***множеством вершин*** (vertex set) графа G, а его элементы — ***вершинами***. Множество E называется ***множеством ребер*** графа G, а его элементы — ***ребрами***. На рис. Б.2*а* изображен ориентированный граф с множеством вершин $\{1, 2, 3, 4, 5, 6\}$. Вершины на рисунке показаны кружками, а ребра — стрелками. Обратите внимание на возможность существования ***ребер-циклов***, или ***петель*** (self-loops), т.е. ребер, соединяющих вершину с самой собой.

В ***неориентированном графе*** (undirected graph) $G = (V, E)$ множество ребер E состоит из *неупорядоченных* пар вершин, т.е. ребро является множеством $\{u, v\}$, где $u, v \in V$ и $u \neq v$. По соглашению для ребер используется запись (u, v), причем и (u, v), и (v, u) обозначают одно и то же ребро неориентированного графа. В неориентированном графе петли запрещены, так что каждое ребро содержит две разные вершины. На рис. Б.2*б* показан неориентированный граф с множеством вершин $\{1, 2, 3, 4, 5, 6\}$.

Многие определения выглядят одинаково и для ориентированных, и для неориентированных графов, хотя некоторые отличия, естественно, имеются. Если (u, v) — ребро направленного графа $G = (V, E)$, то ребро ***выходит из*** (incident

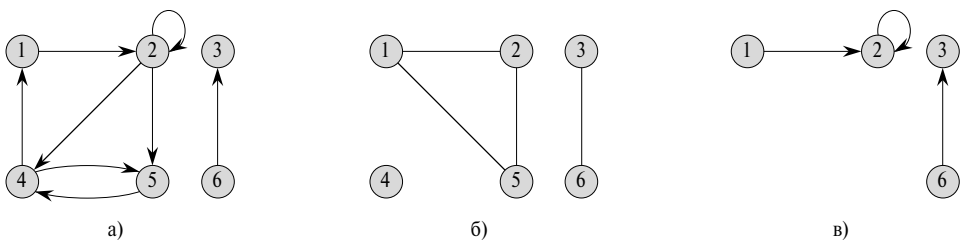


Рис. Б.2. Ориентированные и неориентированные графы

from, leave) вершины u и ***входит в*** (incident to, enter) вершину v. Например, на рис. Б.2*а* из вершины 2 выходят ребра $(2,2)$, $(2,4)$ и $(2,5)$, а входят в вершину 2 ребра $(1,2)$ и $(2,2)$. Если (u,v) — ребро неориентированного графа $G=(V,E)$, то оно ***соединяет вершины*** (incident on) u и v и называется ***инцидентным*** этим вершинам.

Если в графе G имеется ребро (u,v), то говорят, что вершина v ***смежна*** (adjacent) с вершиной u. Для неориентированных графов отношение смежности является симметричным (в случае ориентированных графов это утверждение неверно). Если вершина v смежна с вершиной u, то пишут $u \to v$. На рис. Б.2*а* и рис. Б.2*б* вершина 2 является смежной с вершиной 1, поскольку ребро $(1,2)$ имеется в обоих графах. Вершина 1 смежна с вершиной 2 только на рис. Б.2*б*.

Степенью (degree) вершины в неориентированном графе называется число ребер, соединяющих ее с другими вершинами. Например, вершина 2 на рис. Б.2*б* имеет степень 2. Вершина, степень которой равна 0 (как, например, у вершины 4 на рис. Б.2*б*), называется ***изолированной*** (isolated). В ориентированном графе различают ***исходящую степень*** (out-degree), которая равна количеству выходящих из вершины ребер, и ***входящую степень*** (in-degree), которая равна количеству входящих в вершину ребер. ***Степень*** (degree) вершины в ориентированном графе равна сумме ее входящей и исходящей степеней. Так, на рис. Б.2*а* вершина 2 имеет входящую степень 2, исходящую — 3, так что степень данной вершины равна 5.

Путь (***маршрут***) ***длины*** k от вершины u к вершине u' в графе $G=(V,E)$ представляет собой последовательность $\langle v_0, v_1, v_2, \ldots, v_k\rangle$ вершин, такую что $u = v_0$, $u' = v_k$ и $(v_{i-1}, v_i) \in E$ для $i = 1,2,\ldots,k$. Длиной пути называется количество составляющих его ребер. Путь ***содержит*** (contains) вершины $v_0, v_1, v_2, \ldots, v_k$ и ребра (v_0,v_1), (v_1,v_2), $\ldots$, (v_{k-1},v_k). Всегда имеется путь нулевой длины из вершины в нее саму. Если имеется путь p из вершины u в вершину u', то говорят, что вершина u' ***достижима*** (reachable) из u по пути p, что иногда в ориентированном графе G записывается как $u \overset{p}{\rightsquigarrow} u'$. Путь является ***простым*** (simple), если все вершины пути различны. Например, на рис. Б.2*а* путь $\langle 1,2,5,4\rangle$ является простым путем длины 3; путь $\langle 2,5,4,5\rangle$ простым не является.

Подпуть (subpath) пути $p = \langle v_0, v_1, \ldots, v_k\rangle$ представляет собой непрерывную подпоследовательность его вершин, т.е. для любых $0 \leqslant i \leqslant j \leqslant k$ последовательность вершин $\langle v_i, v_{i+1}, \ldots, v_j\rangle$ является подпутем p.

В ориентированном графе путь $\langle v_0, v_1, \ldots, v_k\rangle$ образует ***цикл*** (cycle), если $v_0 = v_k$ и путь содержит по крайней мере одно ребро. Цикл называется ***простым***, если кроме того все вершины $v_1, v_2, \ldots, v_k$ различны. Петля является циклом с длиной 1. Два пути — $\langle v_0, v_1, \ldots, v_{k-1}, v_0\rangle$ и $\langle v'_0, v'_1, \ldots, v'_{k-1}, v'_0\rangle$ — образуют один и тот же цикл, если существует такое целое j, что $v'_i = v_{(i+j) \bmod k}$ для $i = 0,1,\ldots,k-1$ (один цикл получен из другого сдвигом). На рис. Б.2*а* путь

$\langle 1,2,4,1\rangle$ образует тот же цикл, что и пути $\langle 2,4,1,2\rangle$ и $\langle 4,1,2,4\rangle$. Этот цикл простой, цикл $\langle 1,2,4,5,4,1\rangle$ таковым не является. Цикл $\langle 2,2\rangle$, образованный ребром $(2,2)$, представляет собой петлю. Ориентированный граф, не содержащий петель, называется ***простым***. В неориентированном графе путь $\langle v_0, v_1, \ldots, v_k\rangle$ образует (***простой***) ***цикл***, если $k \geqslant 3$, $v_0 = v_k$ и все вершины $v_1, v_2, \ldots, v_k$ различны. Например, на рис. Б.2*б* путь $\langle 1,2,5,1\rangle$ является циклом. Граф без циклов называется ***ациклическим*** (acyclic).

Неориентированный граф является ***связным*** (connected), если любая его вершина достижима из другой по некоторому пути. Для неориентированного графа отношение "быть достижимым из" является отношением эквивалентности на множестве вершин. Классы эквивалентности называются ***связными компонентами*** (connected components) графа. Например, на рис. Б.2*б* имеются три связных компонента: $\{1,2,5\}$, $\{3,6\}$ и $\{4\}$. Каждая вершина в $\{1,2,5\}$ достижима из другой вершины этого множества. Неориентированный граф считается связным тогда и только тогда, когда он состоит из единственного связного компонента.

Ориентированный граф называется ***сильно связным*** (strongly connected), если любые его две вершины достижимы друг из друга. Любой ориентированный граф можно разбить на ***сильно связные компоненты*** (strongly connected components), которые определяются как классы эквивалентности отношения взаимной достижимости. Ориентированный граф считается сильно связным тогда и только тогда, когда он состоит из единственного сильно связного компонента. Граф на рис. Б.2*а* состоит из трех таких компонентов: $\{1,2,4,5\}$, $\{3\}$ и $\{6\}$. Все пары в множестве $\{1,2,4,5\}$ являются взаимно достижимыми. Вершины $\{3,6\}$ не образуют сильно связный компонент, поскольку из вершины 3 нельзя достичь вершины 6.

Два графа $G = (V,E)$ и $G' = (V',E')$ ***изоморфны*** (isomorphic), если существует биекция $f : V \to V'$, такая что $(u,v) \in E$ тогда и только тогда, когда $(f(u), f(v)) \in E'$. Другими словами, мы можем перенумеровать вершины G, превратив их в вершины G', сохранив при этом ребра между вершинами в неизменном состоянии. На рис. Б.3*а* показана пара изоморфных графов G и G' с множествами вершин $V = \{1,2,3,4,5,6\}$ и $V' = \{u,v,w,x,y,z\}$ соответственно. Отображение V на V' выглядит следующим образом: $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$. Графы на рис. Б.3*б* неизоморфны. Хотя оба изображенных здесь графа имеют по 5 вершин и 7 ребер, граф в верхней части рисунка имеет вершину степени 4, которой нет у нижнего графа.

Мы говорим, что граф $G' = (V',E')$ является ***подграфом*** (subgraph) $G = (V,E)$, если $V' \subseteq V$ и $E' \subseteq E$. Если в графе $G = (V,E)$ выбрано подмножество $V' \subseteq V$, то подграфом графа G, ***порожденным*** (induced) множеством вершин V', является граф $G' = (V',E')$, где

$$E' = \left\{(u,v) \in E : u,v \in V'\right\}.$$

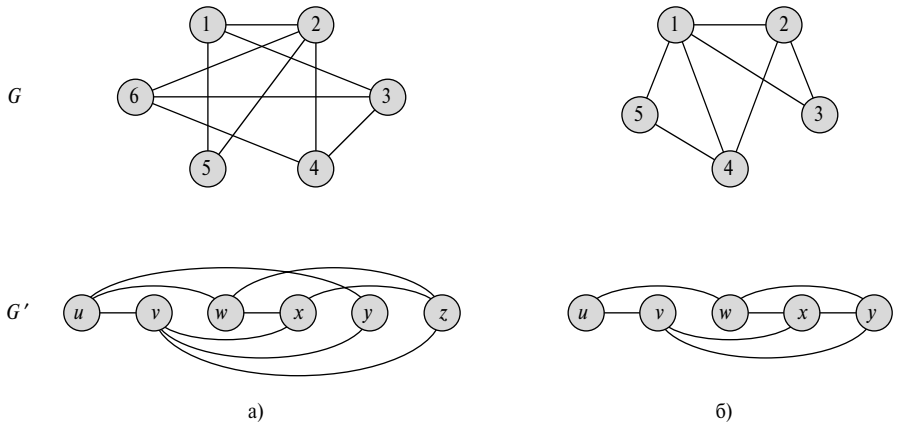


Рис. Б.3. Изоморфные и неизоморфные графы

Подграф, порожденный множеством вершин $\{1, 2, 3, 6\}$ на рис. Б.2*а*, показан на рис. Б.2*в*, он содержит следующее множество ребер: $\{(1, 2), (2, 2), (6, 3)\}$.

Для данного неориентированного графа $G = (V, E)$ его ***ориентированная версия*** (directed version) представляет собой ориентированный граф $G' = (V, E')$, где $(u, v) \in E'$ тогда и только тогда, когда $(u, v) \in E$. Другими словами, каждое неориентированное ребро (u, v) графа G заменяется в ориентированной версии двумя ориентированными ребрами (u, v) и (v, u). Для ориентированного графа $G = (V, E)$ его ***неориентированная версия*** (undirected version) представляет собой неориентированный граф $G' = (V, E')$, где $(u, v) \in E'$ тогда и только тогда, когда $u \neq v$ и $(u, v) \in E$. Другими словами, неориентированная версия содержит ребра графа G "с удаленными стрелочками", причем петли из неориентированной версии убираются. Поскольку в неориентированном графе ребра (u, v) и (v, u) идентичны, неориентированная версия ориентированного графа содержит ребро только по разу, даже если в ориентированном графе между вершинами u и v имеются два ориентированных ребра (u, v) и (v, u). В ориентированном графе ***соседом*** (neighbor) вершины u называется любая вершина, которая становится смежной с u в неориентированной версии, т.е. v является соседом u, если $u \neq v$ и либо $(u, v) \in E$, либо $(v, u) \in E$. В неориентированном графе смежные вершины являются соседями.

Определенные виды графов имеют свои специальные названия. ***Полным*** (complete) графом называется неориентированный граф, в котором каждая пара вершин образована смежными вершинами, т.е. который содержит все возможные ребра. ***Двудольным*** (bipartite) называется неориентированный граф $G = (V, E)$, в котором множество V может быть разделено на два множества V_1 и V_2, такие что из $(u, v) \in E$ следует, что либо $u \in V_1$ и $v \in V_2$, либо $u \in V_2$ и $v \in V_1$.

Ациклический неориентированный граф называется ***лесом*** (forest), а связный ациклический неориентированный граф — (***свободным***) ***деревом*** (free tree).

Имеется еще два варианта графов, с которыми вы можете встретиться. Это ***мультиграф*** (multigraph), который похож на неориентированный граф, но может содержать как петли, так и по несколько ребер между вершинами. ***Гиперграф*** (hypergraph) также похож на неориентированный граф, но он содержит ***гиперребра***, которые могут соединять произвольное количество вершин. Многие алгоритмы, разработанные для обычных ориентированных и неориентированных графов, могут быть обобщены для работы с такими графоподобными структурами.

Сжатием (contraction) неориентированного графа $G = (V, E)$ по ребру $e = (u, v)$ называется граф $G' = (V', E')$, где $V' = V - \{u, v\} \cup \{x\}$ (x — новая вершина). Множество ребер E' образуется из E путем удаления ребра (u, v). Кроме того, для каждой вершины w, инцидентной к u или v, удаляются ребра (u, w) и (v, w) (если они имеются в E) и добавляется новое ребро (x, w).

Упражнения

Б.4-1. На приеме каждый гость подсчитывает, сколько рукопожатий он сделал. По окончании приема вычисляется сумма рукопожатий каждого из гостей. Покажите, что полученная сумма четна, доказав следующую ***лемму о рукопожатиях***: если $G = (V, E)$ — неориентированный граф, то сумма степеней всех вершин равна удвоенному числу ребер $\sum_{v \in V} \text{degree}(v) = 2|E|$.

Б.4-2. Покажите, что если ориентированный или неориентированный граф содержит путь из вершины u в вершину v, то в нем есть простой путь между u и v. Покажите, что если в ориентированном графе есть цикл, то в нем есть простой цикл.

Б.4-3. Покажите, что для любого связного неориентированного графа $G = (V, E)$ выполняется соотношение $|E| \geqslant |V| - 1$.

Б.4-4. Проверьте, что отношение “быть достижимым из” в неориентированном графе является отношением эквивалентности на множестве вершин графа. Какие из трех свойств отношения эквивалентности выполняются для отношения “быть достижимым из” в ориентированном графе?

Б.4-5. Изобразите неориентированную версию графа на рис. Б.2*а* и ориентированную версию графа на рис. Б.2*б*.

⋆ Б.4-6. Покажите, что гиперграф можно представить как двудольный граф, в котором отношение смежности соответствует отношению инцидентности в гиперграфе. (*Указание*: одно множество вершин двудольного графа должно соответствовать вершинам гиперграфа, а второе множество — гиперребрам.)

Б.5 Деревья

Как и слово “граф”, слово “дерево” также употребляется в нескольких родственных смыслах. Здесь представлены основные определения и математические свойства некоторых видов деревьев. Вопросы представления деревьев в памяти компьютера рассматриваются в разделах 10.4 и 22.1.

Б.5.1 Свободные деревья

Как уже говорилось в разделе Б.4, ***свободные деревья*** (free tree), или ***деревья без выделенного корня***, представляют собой связный ациклический неориентированный граф. Прилагательное “свободный” зачастую опускается, когда мы говорим о графе, являющемся деревом. Многие разработанные для деревьев алгоритмы могут работать и с лесом. Пример дерева показан на рис. Б.4*а*, леса — на рис. Б.4*б*. Лес не является деревом в силу того, что представляющий его граф не является связным. Граф на рис. Б.4*в* содержит цикл, а потому не может быть ни деревом, ни лесом.

В следующей теореме указано несколько важных свойств деревьев.

Теорема Б.2 (Свойства свободных деревьев). Пусть $G = (V, E)$ — неориентированный граф. Тогда следующие утверждения равносильны.

1. G — свободное дерево.
2. Любые две вершины G соединяются при помощи единственного простого пути.
3. G — связный граф, но при удалении из E любого ребра перестает быть таковым.
4. G — связный граф, и $|E| = |V| - 1$.
5. G — ациклический граф, и $|E| = |V| - 1$.
6. G — ациклический граф, но при добавлении любого ребра в E получается граф, содержащий цикл.

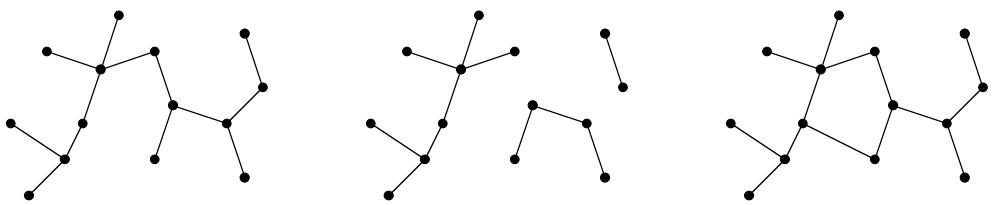


Рис. Б.4. Примеры дерева, леса и графа, который не является деревом или лесом из-за наличия цикла

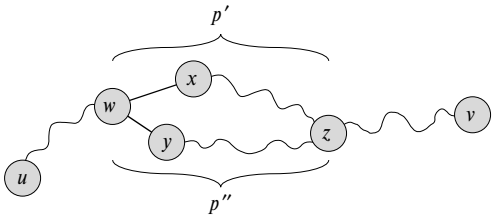


Рис. Б.5. Иллюстрация к доказательству теоремы Б.2

Доказательство. (1)⇒(2). Поскольку дерево является связным, для любых двух вершин G имеется как минимум один соединяющий их простой путь. Пусть u и v — вершины, соединенные двумя простыми путями p_1 и p_2, как показано на рис. Б.5. Пусть w — вершина, в которой пути впервые расходятся, т.е. следующие за w вершины на путях p_1 и p_2, соответственно, x и y, причем $x \neq y$. Пусть z — первая вершина, в которой пути вновь сходятся, т.е. z — первая после w вершина на пути p_1, которая также принадлежит пути p_2. Пусть p' — подпуть p_1 от w через x до z, а p'' — подпуть p_2 от w через y до z. Пути p' и p'' не имеют общих точек, кроме конечных. Тогда путь, образованный объединением p' и пути, обратного к p'', образует цикл. Это противоречит нашему предположению о том, что G является деревом. Таким образом, если G — дерево, то между вершинами не может быть больше одного соединяющего их простого пути.

(2)⇒(3). Если любые две вершины G соединяются единственным простым путем, то G — связный граф. Пусть (u, v) — ребро из E. Это ребро представляет собой путь от u до v, а значит, это единственный путь от u до v. Если мы удалим из графа путь (u, v), пути от u до v не будет, а граф перестанет быть связным.

(3)⇒(4). По условию граф G является связным, а из упражнения Б.4-3 нам известно, что $|E| \geqslant |V| - 1$. Докажем по индукции, что $|E| \leqslant |V| - 1$. Связный граф с одной или двумя вершинами имеет ребер на одно меньше, чем вершин. Предположим, что G имеет $n \geqslant 3$ вершин и что для меньшего числа вершин выполнение условия $|E| \leqslant |V| - 1$ доказано. Удаление произвольного ребра из G разделяет граф на $k \geqslant 2$ связных компонентов (на самом деле $k = 2$). Каждый компонент удовлетворяет условию (3) теоремы, т.к. в противном случае этому условию не удовлетворяет само дерево G. Тогда, по индукции, количество ребер во всех компонентах не превышает $|V| - k \leqslant |V| - 2$. Добавление удаленного ребра дает нам неравенство $|E| \leqslant |V| - 1$.

(4)⇒(5). Предположим, что граф G является связным и что $|E| = |V| - 1$. Мы должны показать, что граф G ациклический. Предположим, что граф содержит цикл, состоящий из k вершин $v_1, v_2, \ldots, v_k$; без потери общности можно считать, что это простой цикл. Пусть $G_k = (V_k, E_k)$ — подграф G, состоящий из данного

цикла. Заметим, что $|V_k| = |E_k| = k$. Если $k < |V|$, то должна существовать вершина $v_{k+1} \in V - V_k$, смежная с некоторой вершиной $v_i \in V_k$, что следует из связности G. Определим подграф $G_{k+1} = (V_{k+1}, E_{k+1})$ графа G как подграф, у которого $V_{k+1} = V_k \cup \{v_{k+1}\}$ и $E_{k+1} = E_k \cup \{(v_i, v_{k+1})\}$. Заметим, что $|V_{k+1}| = |E_{k+1}| = k + 1$. Если $k + 1 < |V|$, мы можем продолжить наше построение, аналогично определяя G_{k+2} и т.д. до тех пор, пока не получим $G_n = (V_n, E_n)$, такой что $n = |V|$, $V_n = V$ и $|E_n| = |V_n| = |V|$. Поскольку G_n — подграф G, $E_n \subseteq E$, следовательно, $|E| \geqslant |V|$, что противоречит предположению $|E| = |V| - 1$. Следовательно, граф G ациклический.

(5)⇒(6). Предположим, что граф G ациклический и что $|E| = |V| - 1$. Пусть k — количество связных компонентов графа G. По определению каждый связный компонент представляет собой свободное дерево; поскольку из (1) вытекает (5), сумма всех ребер во всех связных компонентах G равна $|V| - k$. Следовательно, k должно быть равно 1, а G должно быть деревом. Поскольку из (1) вытекает (2), любые две вершины G соединены единственным простым путем. Таким образом, добавление любого ребра к G создает цикл.

(6)⇒(1). Предположим, что G — ациклический граф, но добавление любого ребра в E приводит к образованию цикла. Мы должны показать, что G — связный граф. Пусть u и v — произвольные вершины графа G. Если u и v не являются смежными, добавление ребра (u, v) создает цикл, в котором все ребра, кроме (u, v), принадлежат G. Таким образом, существует путь из u в v, но поскольку u и v выбраны произвольно, G оказывается связным графом. ■

Б.5.2 Деревья с корнем и упорядоченные деревья

Дерево с корнем (rooted tree) представляет собой свободное дерево, в котором выделена одна вершина, именуемая ***корнем*** (root) дерева. Зачастую вершины в дереве с корнем называют ***узлами***[4] (nodes) дерева. На рис. Б.6*а* показано дерево с корнем, состоящее из 12 узлов с корнем в узле 7.

Рассмотрим узел x в дереве T с корнем r. Любой узел y на (единственном) пути от r к x называется ***предком*** (ancestor) x. Если y является предком x, то x является ***потомком*** (descendant) y (каждый узел является собственным предком и потомком). Если y — предок x и $x \neq y$, то y — ***истинный предок*** (proper ancestor) x, а x, соответственно, ***истинный потомок*** (proper descendant) y. ***Поддеревом с корнем в узле*** x (subtree rooted at x) называется дерево, порожденное потомками x, корнем которого является узел x. Например, на рис. Б.6*а* поддерево с корнем в узле 8 содержит узлы 8, 6, 5 и 9.

Если (y, x) — последнее ребро на пути от корня r дерева T к узлу x, то узел y является ***родительским*** (parent) по отношению к x, а x — ***ребенком*** (child),

[4]Термин “узел” зачастую используется в теории графов как синоним термина “вершина”. Мы же будем использовать термин “узел” только для вершины дерева с корнем.

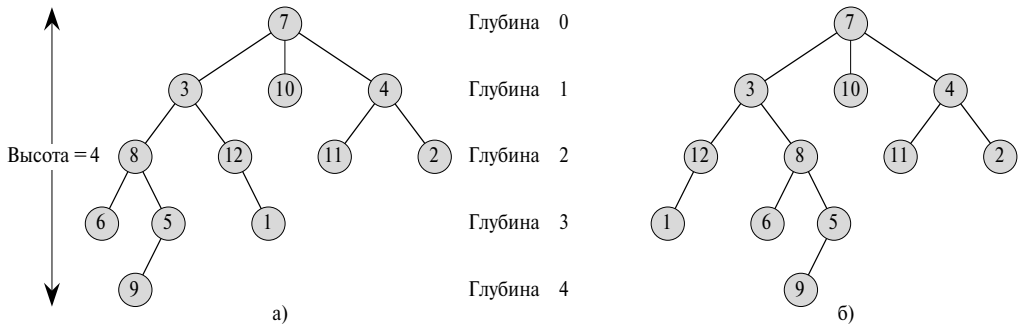


Рис. Б.6. Деревья с корнем и упорядоченные деревья

или ***дочерним*** узлом по отношению к узлу y. Корень дерева — единственный узел, не имеющий родительского узла. Если два узла имеют общий родительский узел, мы будем называть такие узлы ***родственными***, или ***братьями*** (siblings). Узел, у которого нет дочерних узлов, называется ***внешним узлом*** (external node) или ***листом*** (leaf). Узел, не являющийся листом, называется ***внутренним узлом*** (internal node).

Количество дочерних узлов узла x называется его ***степенью***[5] (degree). Длина пути от корня r к узлу x называется ***глубиной*** (depth) узла x в дереве. ***Высота*** (height) узла в дереве равна количеству ребер в самом длинном простом нисходящем пути от узла к листу; высотой дерева называется высота его корня. Высота дерева равна также наибольшей глубине узла дерева.

Упорядоченное дерево (ordered tree) представляет собой дерево с корнем, в котором дочерние узлы каждого узла упорядочены. Т.е. если узел имеет k дочерних узлов, то существует первый, второй, ..., k-й дочерние узлы. Два дерева, приведенные на рис. Б.6*а* и рис. Б.6*б*, отличаются, если рассматривать их как упорядоченные, но одинаковы, если трактовать их как обычные деревья с корнем.

Б.5.3 Бинарные и позиционные деревья

Бинарные деревья определяются рекурсивно. ***Бинарное дерево*** (binary tree) T представляет собой конечное множество узлов, которое

- либо не содержит узлов,
- либо состоит из трех непересекающихся множеств узлов: ***корневой*** узел, бинарное дерево, называемое ***левым поддеревом*** (left subtree), и бинарное дерево, называемое ***правым поддеревом*** (right subtree).

[5] Заметим, что степень узла зависит от того, рассматриваем ли мы дерево с корнем или свободное дерево. Степень вершины в свободном дереве, как и в любом неориентированном графе, равна количеству смежных вершин. В дереве с корнем степень равна количеству дочерних узлов — родительский узел при вычислении степени не учитывается.

Бинарное дерево, которое не содержит узлов, называется ***пустым*** (empty tree) или ***нулевым*** (null tree), и иногда обозначается как NIL. Если левое поддерево непустое, его корень называется ***левым ребенком*** (left child) корня всего дерева; аналогично, корень непустого правого поддерева называется ***правым ребенком*** (right child). Если поддерево является пустым, мы говорим, что соответствующий ребенок ***отсутствует*** (absent). Пример бинарного дерева можно увидеть на рис. Б.7*а*.

Бинарное дерево представляет собой не просто упорядоченное дерево, в котором каждый узел имеет степень не более 2. Например, в бинарном дереве в случае узла с одним дочерним имеет значение, какой именно этот дочерний узел — левый или правый. В упорядоченных деревьях такое различие в случае одного дочернего узла не делается. На рис. Б.7*б* показано бинарное дерево, отличающееся от приведенного на рис. Б.7*а* позицией одного узла. Если рассматривать эти деревья как просто упорядоченные, то они являются идентичными.

Пустующие места в бинарном дереве можно заполнить фиктивными листьями, как показано на рис. Б.7*в*, где они изображены квадратами. Так получается ***полностью бинарное дерево*** (full binary tree): каждый узел либо представляет собой лист, либо имеет степень 2. Узлы со степенью 1 в таком дереве отсутствуют.

Информация о позициях узлов, которая отличает упорядоченные деревья от бинарных, может быть расширена на случай деревьев с более чем 2 дочерними узлами в каждом узле. В ***позиционном дереве*** (positional tree) все дочерние узлы данного узла пронумерованы различными натуральными числами. Если у данного узла среди дочерних нет узла с номером i, то i-й дочерний узел у данного узла ***отсутствует*** (absent). Позиционное дерево называется ***k-арным*** (k-ary) деревом, если в нем не имеется дочерних узлов с номером, превышающим k. Таким образом, бинарное дерево представляет собой k-арное дерево при $k = 2$.

Полным k-арным деревом (complete k-ary tree) называется k-арное дерево, у которого все листья имеют одну и ту же глубину, а все внутренние узлы — одну и ту же степень k. Так, на рис. Б.8 показано полное бинарное дерево высота

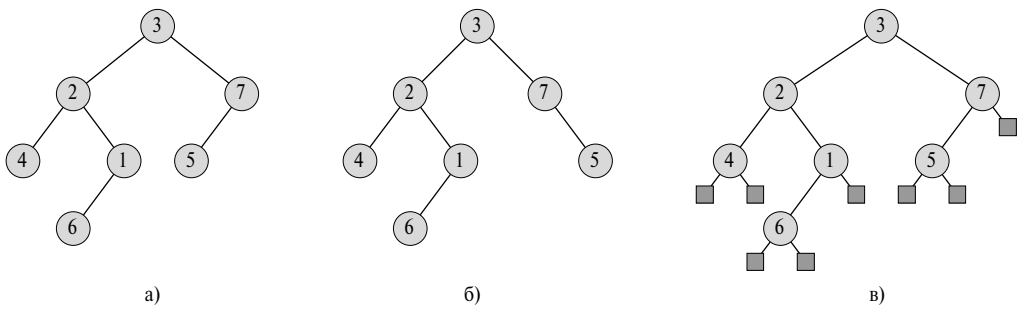


Рис. Б.7. Бинарные деревья

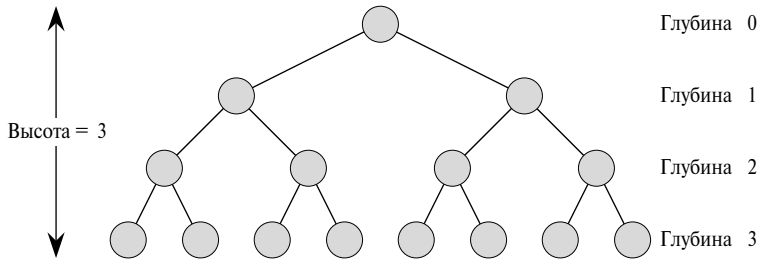


Рис. Б.8. Полное бинарное дерево, которое имеет высоту 3, 8 листьев и 7 внутренних узлов

которого равна 3. Сколько листьев содержится в полном k-арном дереве, высота которого равна h? Корень имеет k дочерних узлов на глубине 1, каждый из которых содержит по k дочерних узлов с глубиной 2 и т.д. Таким образом, количество листьев на глубине h равно k^h. Соответственно, высота полного k-арного дерева с n листьями равна $\log_k n$. Количество внутренних узлов полного k-арного дерева высоты h равно

$$1 + k + k^2 + \cdots + k^{h-1} = \sum_{i=0}^{h-1} k^i = \frac{k^h - 1}{k - 1}$$

в соответствии с (А.5). Таким образом, полное бинарное дерево содержит $2^h - 1$ внутренних узлов.

Упражнения

Б.5-1. Нарисуйте все свободные деревья, состоящие из 3 вершин A, B и C. Изобразите все корневые деревья с этими же узлами и узлом A в качестве корня. Нарисуйте все упорядоченные деревья с узлами A, B и C и узлом A в качестве корня. Изобразите все бинарные деревья с этими же узлами и узлом A в качестве корня.

Б.5-2. Пусть $G = (V, E)$ — ориентированный ациклический граф, в котором имеется вершина $v_0 \in V$, такая что имеется единственный путь от v_0 к любой другой вершине $v \in V$. Докажите, что неориентированная версия графа G является деревом.

Б.5-3. Покажите по индукции, что количество узлов степени 2 в любом непустом бинарном дереве на 1 меньше количества листьев.

Б.5-4. Покажите с использованием метода математической индукции, что непустое бинарное дерево с n узлами имеет высоту как минимум $\lfloor \lg n \rfloor$.

⋆ Б.5-5. Определим ***длину внутреннего пути*** (internal path length) полностью бинарного дерева как сумму глубин всех внутренних узлов дерева. Аналогично, под ***длиной внешнего пути*** (external path length) будем подразумевать сумму глубин всех листьев дерева. Рассмотрим полностью бинарное дерево с n внутренними узлами, длиной внутреннего пути i и длиной внешнего пути e. Докажите, что $e = i + 2n$.

⋆ Б.5-6. Назначим каждому листу x с глубиной d бинарного дерева T "вес" $w(x) = 2^{-d}$. Докажите, что $\sum_x w(x) \leqslant 1$, где суммирование выполняется по всем листьям дерева T (***неравенство Крафта*** (Kraft inequality)).

⋆ Б.5-7. Покажите, что если $L \geqslant 2$, то каждое бинарное дерево с L листьями содержит поддерево с количеством листьев от $L/3$ до $2L/3$ включительно.

Задачи

Б-1. **Раскраска графа**

Назовем ***k-раскраской*** (k-coloring) неориентированного графа $G = (V, E)$ функцию $c : V \to \{0, 1, \ldots, k-1\}$, такую что для всех ребер $(u, v) \in E$ выполняется $c(u) \neq c(v)$. Другими словами, числа $0, 1, \ldots, k-1$ представляют k цветов, и смежные вершины графа должны иметь разные цвета.

а) Покажите, что любое дерево можно раскрасить двумя цветами.

б) Покажите, что следующие утверждения эквивалентны:

1) граф G — двудольный;
2) граф G можно раскрасить двумя цветами;
3) граф G не имеет циклов нечетной длины.

в) Пусть d — максимальная степень вершины в графе G. Докажите, что G может быть раскрашен при помощи $d + 1$ цветов.

г) Покажите, что если граф имеет $O(|V|)$ ребер, то его можно раскрасить при помощи $O\left(\sqrt{|V|}\right)$ цветов.

Б-2. **Граф дружбы**

Преобразуйте следующие утверждения в теоремы о неориентированных графах и докажите их. Отношение дружбы считаем симметричным, но не рефлексивным.

а) В любой группе из $n \geqslant 2$ человек имеется два человека с одним и тем же количеством друзей из этой группы.

б) Каждая группа из шести человек содержит либо три человека, которые являются друзьями друг друга, либо три человека, никакие два из которых не являются друзьями.

в) Любую группу людей можно разделить на две подгруппы так, что как минимум половина друзей каждого человека из одной подгруппы будет находиться в другой подгруппе.

г) Если каждый человек в группе является другом по меньшей мере для половины группы, то можно рассадить эту группу людей за столом так, что каждый будет сидеть между двумя друзьями.

Б-3. **Разбиение деревьев**

Многие алгоритмы типа "разделяй и властвуй", работающие с графами, требуют разбиения графа на два близких по размеру подграфа. Вопрос заключается в том, как сделать это с наименьшим количеством удаляемых ребер.

а) Покажите, что удалением единственного ребра можно разбить вершины любого бинарного дерева с n вершинами на два множества, в каждом из которых оказывается не больше $3n/4$ вершин.

б) Покажите, что константу $3/4$ из пункта *а*) нельзя улучшить. Для этого приведите пример простого бинарного дерева, для которого при удалении любого ребра в одной из частей оказывается ровно $3n/4$ вершин.

в) Покажите, что, удаляя не более $O(\lg n)$ вершин, мы можем разбить бинарное дерево с n вершинами на такие два множества A и B, что $|A| = \lfloor n/2 \rfloor$ и $|B| = \lceil n/2 \rceil$.

Заключительные замечания

Основатель символьной логики Дж. Буль (G. Boole) ввел многие обозначения, связанные с множествами, в своей книге, изданной в 1854 году. Современная теория множеств (в первую очередь теория мощности бесконечных множеств) была создана Кантором (G. Cantor) в 1874–1895 гг. Термин "функция" введен Лейбницем (G.W. Leibniz) для некоторых типов математических формул. Его весьма ограниченное определение функции позже неоднократно обобщалось и расширялось. Создание теории графов относится к 1736 году, когда Л. Эйлер (L. Euler) доказал невозможность такого обхода семи мостов в Кенигсберге, при котором выполняется по одному проходу по каждому из мостов, и обход завершается в исходной точке.

Полезным справочником, содержащим множество определений и свойств графов, является книга Харари (Harary) [138].

ПРИЛОЖЕНИЕ B

Комбинаторика и теория вероятности

В этом приложении вы познакомитесь с азами комбинаторики и теории вероятности. Если вы уже знакомы с этими разделами математики, то можете просто бегло ознакомиться с началом приложения и обратить большее внимание на его окончание. Большинство глав в этой книге не используют теорию вероятностей, но некоторые целиком построены на ее применении.

В разделе B.1 приведен обзор основ комбинаторики, включая формулы для количества перестановок и сочетаний. В разделе B.2 вас ожидает встреча с аксиомами теории вероятности и основами распределения вероятностей. Случайные величины, а также математическое ожидание и дисперсия рассматриваются в разделе B.3. Раздел B.4 посвящен геометрическому и биномиальному распределениям, изучение которых продолжается в разделе B.5, где обсуждается проблема "хвостов" распределений.

B.1 Основы комбинаторики

Комбинаторика пытается ответить на вопрос "Сколько?", не выполняя перечисления. Например, вы можете спросить "Сколько всего имеется различных n-битовых чисел?" или "Сколькими способами можно упорядочить n различных чисел?" Здесь мы познакомимся с азами комбинаторики, которые предполагают знание основ теории множеств, так что, надеемся, вы основательно проработали предыдущее приложение.

Правила суммы и произведения

Множество, количество элементов которого мы хотим подсчитать, иногда можно выразить как объединение непересекающихся множеств или как декартово произведение множеств.

Правило суммы гласит, что количество способов, которыми можно выбрать элемент из одного из двух *непересекающихся* множеств, равно сумме мощностей этих множеств. То есть, если A и B — два конечных множества без общих членов, то $|A \cup B| = |A| + |B|$, что следует из уравнения (Б.3). Например, если каждый символ в номере машины должен быть либо латинской буквой, либо цифрой, то всего имеется $26 + 10 = 36$ различных вариантов выбора этого символа, т.к. всего имеется 26 вариантов выбора буквы и 10 — цифры.

Правило произведения гласит, что количество способов, которыми можно выбрать упорядоченную пару, равно количеству вариантов выбора первого элемента, умноженному на количество вариантов выбора второго элемента. То есть, если A и B — конечные множества, то $|A \times B| = |A| \cdot |B|$ (см. уравнение (Б.4)). Например, имея 28 сортов мороженого и 4 разных сиропа, можно приготовить $28 \cdot 4 = 112$ различных вариантов мороженого с сиропом.

Строки

Строкой (string) на конечном множестве S называют последовательность элементов S. Например, вот восемь двоичных (составленных из 0 и 1) строк длины 3: 000, 001, 010, 011, 100, 101, 110, 111. Иногда строки длиной k называются ***k-строками***. ***Подстрокой*** (substring) s' строки s называется упорядоченная последовательность элементов s. Подстрока длиной k называется ***k-подстрокой***. Например, 010 — 3-подстрока строки 01101001 (начинающаяся с 4 позиции строки); 111 же подстрокой указанной строки не является.

k-строка на множестве S может рассматриваться как элемент декартова произведения S^k, так что всего имеется $|S|^k$ строк длины k, в частности, число двоичных k-строк равно 2^k. Интуитивно это очевидно — при построении k-строки из множества с n элементами у нас имеется n вариантов выбора первого элемента строки; для каждого из первых элементов у нас имеется n вариантов выбора второго элемента строки, и так k раз. Это дает нам общее количество k-строк, равное $n \cdot n \cdot \ldots \cdot n = n^k$.

Перестановки

Перестановкой (permutation) конечного множества S называется упорядоченная последовательность всех элементов S, в которой каждый элемент встречается ровно один раз. Например, если $S = \{a, b, c\}$, то имеется шесть перестановок S:

$$abc, acb, bac, bca, cab, cba.$$

Всего имеется $n!$ перестановок множества из n элементов, поскольку первый элемент может быть выбран n способами, второй — $n-1$ способом, третий — $n-2$ и т.д.

k-перестановкой[1] S называется упорядоченная последовательность k элементов из S, в которой ни один элемент не встречается дважды (таким образом, обычная перестановка представляет собой n-перестановку множества из n элементов. Для множества $S = \{a, b, c, d\}$ имеется двенадцать 2-перестановок:

$$ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc.$$

Количество k-перестановок множества из n элементов равно

$$n\,(n-1)\,(n-2)\cdots(n-k+1) = \frac{n!}{(n-k)!}, \tag{B.1}$$

поскольку имеется n способов выбора первого элемента, $n-1$ — второго и т.д., до последнего, k-го элемента, который можно выбрать из оставшихся $n-k+1$ элементов множества.

Сочетания

Сочетаниями (k-combination) из n элементов по k называются k-элементные подмножества n-элементного множества. Например, имеется шесть сочетаний по 2 элемента из множества $S = \{a, b, c, d\}$:

$$ab, ac, ad, bc, bd, cd.$$

(Здесь для простоты для записи подмножества $\{a, b\}$ мы использовали краткую запись ab). Для построения сочетания из множества просто выбирается k различных элементов.

Количество сочетаний можно выразить через количество размещений. Для каждого сочетания имеется $k!$ перестановок его элементов, каждая из которых представляет собой одно из размещений из n элементов по k. Таким образом, количество сочетаний из n элементов по k равно количеству размещений, деленному на $k!$, т.е. с учетом (B.1) количество сочетаний из n элементов по k равно

$$\frac{n!}{k!\,(n-k)!}. \tag{B.2}$$

При $k = 0$ эта формула дает 1, т.е. выбрать пустое подмножество можно единственным способом (напомним, что $0! = 1$).

[1] В отечественной литературе k-перестановка называется **размещением**. — *Прим. ред.*

Биномиальные коэффициенты

Для числа сочетаний из n элементов по k используется обозначение $\binom{n}{k}$ (в отечественной литературе для этой величины принято обозначение C_n^k). Из (В.2) следует, что

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}.$$

Эта формула симметрична относительно k и $n-k$:

$$\binom{n}{k} = \binom{n}{n-k}. \tag{В.3}$$

Эти числа известны также как ***биномиальные коэффициенты***, поскольку они участвуют в ***биноме Ньютона***:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}. \tag{В.4}$$

В частном случае $x = y = 1$ мы получаем

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n.$$

Комбинаторный смысл этой формулы заключается в подсчете количества двоичных строк длины n (число которых равно 2^n) как суммы количества строк с разным числом единиц (имеется $\binom{n}{k}$ двоичных строк длины n с k единицами, т.к. $\binom{n}{k}$ — количество способов выбрать k позиций для единиц в строке длины n).

Имеется масса различных тождеств, в которых принимают участие биномиальные коэффициенты (с некоторыми из них вы познакомитесь в упражнениях к данному разделу).

Оценки биномиальных коэффициентов

В некоторых случаях нам может потребоваться оценить величину биномиальных коэффициентов и указать их границы. Нижняя граница для $1 \leqslant k \leqslant n$ может быть оценена следующим образом:

$$\binom{n}{k} = \frac{n\,(n-1)\cdots(n-k+1)}{k\,(k-1)\cdots 1} = \left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-k+1}{1}\right) \geqslant \left(\frac{n}{k}\right)^k.$$

Используя неравенство $k! \geqslant (k/e)^k$, являющееся следствием из формулы Стирлинга (3.17), мы можем получить оценку верхней границы биномиальных коэффициентов:

$$\binom{n}{k} = \frac{n\,(n-1)\cdots(n-k+1)}{k\,(k-1)\cdots 1} \leqslant \frac{n^k}{k!} \leqslant \left(\frac{en}{k}\right)^k. \tag{В.5}$$

Для всех $0 \leqslant k \leqslant n$ по индукции можно доказать (см. упражнение B.1-12), что

$$\binom{n}{k} \leqslant \frac{n^n}{k^k (n-k)^{n-k}}, \tag{B.6}$$

где для удобства принято, что $0^0 = 1$. Для $k = \lambda n$, где $0 \leqslant \lambda \leqslant 1$, это неравенство можно переписать как

$$\binom{n}{\lambda n} \leqslant \frac{n^n}{(\lambda n)^{\lambda n} ((1-\lambda) n)^{(1-\lambda)n}} = \left(\left(\frac{1}{\lambda}\right)^{\lambda} \left(\frac{1}{1-\lambda}\right)^{1-\lambda} \right)^n = 2^{nH(\lambda)},$$

где

$$H(\lambda) = -\lambda \lg \lambda - (1-\lambda) \lg (1-\lambda) \tag{B.7}$$

называется (***двоичной***) ***энтропийной функцией*** (binary entropy function). Для удобства принято, что $0 \lg 0 = 0$, так что $H(0) = H(1) = 0$.

Упражнения

B.1-1. Сколько имеется k-подстрок у n-строки? (Одинаковые подстроки, начинающиеся в разных позициях строки, считаются разными.) Сколько всего подстрок имеется у строки длиной n?

B.1-2. ***Булева функция*** (boolean function) с n входами и m выходами — это функция с областью определения $\{\text{TRUE}, \text{FALSE}\}^n$ и областью значений $\{\text{TRUE}, \text{FALSE}\}^m$. Сколько всего имеется различных функций с n входами и 1 выходом? С n входами и m выходами?

B.1-3. Сколькими способами n профессоров могут разместиться на конференции за круглым столом? Варианты, отличающиеся поворотом, считаются одинаковыми.

B.1-4. Сколькими способами можно выбрать из множества $\{1, 2, \ldots, 100\}$ три различных числа так, чтобы их сумма была четной?

B.1-5. Докажите тождество

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \quad \text{для } 0 < k \leqslant n. \tag{B.8}$$

B.1-6. Докажите тождество

$$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k} \quad \text{для } 0 \leqslant k < n.$$

В.1-7. При выборе k объектов из n один из объектов можно пометить специальным образом и следить, выбран он или нет. Используя этот подход, докажите, что

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

В.1-8. Используя результат упражнения В.1-7, составьте таблицу биномиальных коэффициентов $\binom{n}{k}$ для $n = 0, 1, \ldots, 6$ и $0 \leqslant k \leqslant n$ в виде равнобедренного треугольника (в первой строке — $\binom{0}{0}$, во второй — $\binom{1}{0}$ и $\binom{1}{1}$ и т.д.). Такая таблица биномиальных коэффициентов называется ***треугольником Паскаля***.

В.1-9. Докажите, что

$$\sum_{i=1}^{n} i = \binom{n+1}{2}.$$

В.1-10. Покажите, что для любого $n \geqslant 0$ и $0 \leqslant k \leqslant n$ максимальное значение $\binom{n}{k}$ достигается при $k = \lfloor n/2 \rfloor$ или $k = \lceil n/2 \rceil$.

⋆ В.1-11. Докажите, что для любых $n, k, j \geqslant 0$ и $j+k \leqslant n$ выполняется неравенство

$$\binom{n}{j+k} \leqslant \binom{n}{j}\binom{n-j}{k}. \tag{В.9}$$

Приведите как алгебраическое доказательство данного неравенства, так и доказательство, основанное на рассуждениях о выборе $j+k$ элементов из n. Когда данное неравенство превращается в равенство?

⋆ В.1-12. Докажите неравенство (В.6) по индукции для $k \leqslant n/2$, а затем воспользуйтесь уравнением (В.3) для распространения результата на все $k \leqslant n$.

⋆ В.1-13. Воспользуйтесь приближением Стирлинга для доказательства того, что

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}\left(1 + O\left(1/n\right)\right). \tag{В.10}$$

⋆ В.1-14. Дифференцируя энтропийную функцию $H(\lambda)$, покажите, что ее максимум достигается при $\lambda = 1/2$. Чему равно значение $H(1/2)$?

⋆ В.1-15. Покажите, что для любого натурального n

$$\sum_{k=0}^{n} \binom{n}{k} k = n2^{n-1}. \tag{В.11}$$

B.2 Вероятность

Вероятность является очень важным инструментом при разработке и анализе вероятностных и рандомизированных алгоритмов. В этом разделе вы познакомитесь с основами теории вероятности.

Мы определим вероятность с помощью ***пространства событий*** (sample space) S, которое представляет собой множество ***элементарных событий*** (elementary events). Каждое элементарное событие может рассматриваться как возможный исход некоторого эксперимента. Например, в случае эксперимента, состоящего в подбрасывании двух различимых монеток пространство событий состоит из всех возможных 2-строк над множеством $\{\mathrm{O}, \mathrm{P}\}$ (где O обозначает выпадение орла, а P-решки:

$$S = \{\mathrm{OO}, \mathrm{OP}, \mathrm{PO}, \mathrm{PP}\}.$$

Событие (event) представляет собой подмножество пространства событий S. Например, в эксперименте с бросанием двух монет событием может быть выпадение одного орла и одной решки: $\{\mathrm{OP}, \mathrm{PO}\}$. Событие S называется ***достоверным событием*** (certain event), а событие $\emptyset$ — ***невозможным*** (null event). Мы говорим, что два события A и B являются взаимоисключающими (mutually exclusive), если $A \cap B = \emptyset$. Каждое элементарное событие $s \in S$ также будет рассматриваться нами как событие $\{s\}$. Все элементарные события по определению являются взаимоисключающими.

Аксиомы вероятности

Распределение вероятностей (probability distribution) $\Pr\{\}$ на пространстве событий S отображает события на действительные числа, удовлетворяя при этом ***аксиомам вероятности***:

1. Для любого события A $\Pr\{A\} \geqslant 0$.
2. $\Pr\{S\} = 1$.
3. Для любых двух взаимоисключающих событий A и B $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}$. В общем случае для любой (конечной или бесконечной счетной) последовательности попарно взаимоисключающих событий $A_1, A_2, \ldots$

$$\Pr\left\{\bigcup_i A_i\right\} = \sum_i \Pr\{A_i\}.$$

Мы называем $\Pr\{A\}$ ***вероятностью*** (probability) события A. Заметим, что аксиома 2 выполняет нормализующее действие: нет никаких фундаментальных оснований в выборе в качестве вероятности достоверного события именно 1; просто такое значение наиболее естественное и удобное.

Некоторые результаты следуют непосредственно из приведенных аксиом и основ теории множеств (см. раздел Б.1). Невозможное событие имеет вероятность $\Pr\{\emptyset\} = 0$. Если $A \subseteq B$, то $\Pr\{A\} \leqslant \Pr\{B\}$. Используя запись $\bar{A}$ для обозначения события $S - A$ (***дополнения*** (complement) A), получим $\Pr\{\bar{A}\} = 1 - \Pr\{A\}$. Для любых двух событий A и B

$$\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\} - \Pr\{A \cap B\} \leqslant \tag{В.12}$$

$$\leqslant \Pr\{A\} + \Pr\{B\} \tag{В.13}$$

Предположим, что в нашем примере с бросанием монет вероятность каждого из четырех элементарных событий равна $1/4$. Тогда вероятность получить как минимум одного орла равна

$$\Pr\{\text{OO}, \text{OP}, \text{PO}\} = \Pr\{\text{OO}\} + \Pr\{\text{OP}\} + \Pr\{\text{PO}\} = 3/4.$$

Другой способ получить эту вероятность — это заметить, что единственный способ получить при броске меньше одного орла — это выпадение двух решек, вероятность чего равна $\Pr\{\text{PP}\} = 1/4$, так что вероятность получить по крайней мере одного орла равна $1 - 1/4 = 3/4$.

Дискретные распределения вероятностей

Распределение вероятностей называется ***дискретным*** (discrete), если оно определено на конечном или бесконечном счетном пространстве событий. Пусть S — пространство событий. Тогда для любого события A

$$\Pr\{A\} = \sum_{s \in A} \Pr\{s\},$$

поскольку элементарные события, составляющие A, являются взаимоисключающими. Если S конечно и каждое элементарное событие $s \in S$ имеет вероятность

$$\Pr\{s\} = 1/|S|,$$

то мы имеем дело с ***равномерным распределением вероятностей*** (uniform probability distribution) на S. В таком случае эксперимент часто описывается словами "выберем случайным образом элемент S".

В качестве примера рассмотрим бросание ***симметричной монеты*** (fair coin), для которой вероятность выпадения орла равна вероятности выпадения решки и составляет $1/2$. Если мы бросаем монету n раз, то получим равномерное распределение вероятностей на пространстве событий $S = \{\text{O,P}\}^n$ (которое представляет собой множество размером 2^n). Каждое элементарное событие S может

быть представлено строкой длиной n на множестве {O,P}, и вероятность каждого элементарного события равна $1/2^n$. Событие

$$A = \{\text{Выпало ровно } k \text{ орлов и } n - k \text{ решек}\}$$

представляет собой подмножество S размером $|A| = \binom{n}{k}$, поскольку имеется ровно $\binom{n}{k}$ строк длиной n на множестве {O,P}, содержащих k O. Вероятность события A, таким образом, равна $\binom{n}{k}/2^n$.

Непрерывное равномерное распределение вероятности

Непрерывное равномерное распределение вероятности представляет собой пример распределения вероятности, в котором не все подмножества пространства событий рассматриваются как события. Непрерывное равномерное распределение вероятности определено на закрытом отрезке $[a, b]$ действительных чисел ($a < b$). Интуитивно все точки отрезка $[a, b]$ рассматриваются как "равновероятные". Имеется несчетное множество точек, так что мы не можем назначить каждой точке свою конечную положительную вероятность, так как мы не сможем одновременно удовлетворить аксиомы 2 и 3. По этой причине мы должны связывать вероятность только с *некоторыми* из подмножеств S, чтобы удовлетворить аксиомы вероятности для этих событий.

Для любого закрытого отрезка $[c, d]$, где $a \leqslant c \leqslant d \leqslant b$, ***непрерывное равномерное распределение вероятности*** (continuous uniform probability distribution) определяет вероятность события $[c, d]$ как

$$\Pr\{[c, d]\} = \frac{d - c}{b - a}.$$

Обратите внимание, что для произвольной отдельной точки $x = [x, x]$ вероятность x равна 0. Если мы удалим конечные точки отрезка $[c, d]$, то получим открытый интервал (c, d). Поскольку $[c, d] = [c, c] \cup (c, d) \cup [d, d]$, в соответствии с аксиомой 3 $\Pr\{[c, d]\} = \Pr\{(c, d)\}$. Вообще говоря, множество событий для непрерывного равномерного распределения вероятности представляет собой любое подмножество пространства событий $[a, b]$, которое может быть получено конечным (или бесконечным счетным) объединением открытых и закрытых интервалов.

Условная вероятность и независимость

Иногда мы располагаем частичной информацией о результате эксперимента. Например, пусть нам известно, что в результате бросания двух симметричных монет по крайней мере на одной из них выпал орел. Чему в таком случае равна вероятность того, что обе монеты выпали орлом? Имеющаяся информация позволяет исключить выпадение двух решек, а три оставшихся элементарных события

имеют равную вероятность $1/3$, так что именно такой и будет интересующая нас вероятность выпадения двух орлов.

Изложенная идея формализуется в определении ***условной вероятности*** (conditional probability) события A при условии осуществления события B:

$$\Pr\{A \mid B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}} \tag{В.14}$$

(при этом $\Pr\{B\} \neq 0$). Интуитивно формула легко объяснима. Вероятность того, что произойдет как событие A, так и событие B (т.е. $\Pr\{A \cap B\}$), равна вероятности того, что произойдет событие B ($\Pr\{B\}$), умноженной на вероятность того, что при условии осуществления события B произойдет еще и событие A ($\Pr\{A \mid B\}$), т.е. $\Pr\{A \cap B\} = \Pr\{B\} \Pr\{A \mid B\}$.

В приведенном выше примере эксперимента событие A заключается в том, что обе монеты выпали орлами, а B — что как минимум одна монета выпала орлом. Таким образом, $\Pr\{A \mid B\} = (1/4)/(3/4) = 1/3$.

Два события называются ***независимыми*** (independent), если

$$\Pr\{A \cap B\} = \Pr\{A\} \Pr\{B\}, \tag{В.15}$$

что при $\Pr\{B\} \neq 0$ эквивалентно условию $\Pr\{A \mid B\} = \Pr\{A\}$

В нашем примере с бросанием двух монет результаты отбельных бросков независимы, так что вероятность выпадения двух орлов равна $(1/2)(1/2) = 1/4$. Предположим теперь, что одно событие состоит в том, что первая монета выпала орлом, а второе — что монеты выпали по-разному. Каждое из этих событий имеет вероятность $1/2$, а вероятность осуществления обоих — $1/4$. В соответствии с определением, эти события независимы, несмотря на то, что на первый взгляд это не очевидно. И наконец, представим, что монеты спаяны вместе, так что они либо обе выпадают орлами, либо обе выпадают решками (вероятности этих выпадений равны). Итак, вероятность выпадения каждой монеты орлом — $1/2$, но и вероятность того, что обе монеты выпадут орлом, — тоже $1/2$, а так как $1/2 \neq (1/2)(1/2)$, следовательно, события "первая монета выпала орлом" и "вторая монета выпала орлом" в данном случае не являются независимыми.

События $A_1, A_2, \ldots, A_n$ называются ***попарно независимыми*** (pairwise independent), если для всех $1 \leqslant i < j \leqslant n$ выполняется равенство

$$\Pr\{A_i \cap A_j\} = \Pr\{A_i\} \Pr\{A_j\}.$$

Мы говорим, что эти события ***независимы в совокупности*** (mutually independent), если для любого k-подмножества $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$ исходного множества, где $2 \leqslant k \leqslant n$ и $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$, выполняется равенство

$$\Pr\{A_{i_1} \cap A_{i_2} \cap \ldots \cap A_{i_k}\} = \Pr\{A_{i_1}\} \Pr\{A_{i_2}\} \cdots \Pr\{A_{i_k}\}.$$

Например, предположим, что мы бросили две симметричные монеты. Пусть A_1 — событие, заключающееся в выпадении первой монеты орлом, A_2 — событие, заключающееся в выпадении орлом второй монеты, а A_3 — что монеты выпали по-разному. Мы имеем

$$\begin{aligned}
\Pr\{A_1\} &= 1/2,\\
\Pr\{A_2\} &= 1/2,\\
\Pr\{A_3\} &= 1/2,\\
\Pr\{A_1 \cap A_2\} &= 1/4,\\
\Pr\{A_1 \cap A_3\} &= 1/4,\\
\Pr\{A_2 \cap A_3\} &= 1/4,\\
\Pr\{A_1 \cap A_2 \cap A_3\} &= 0.
\end{aligned}$$

Поскольку для $1 \leqslant i < j \leqslant 3$ $\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\} = 1/4$, события A_1, A_2 и A_3 попарно независимы, однако не являются независимыми в совокупности, так как $\Pr\{A_1 \cap A_2 \cap A_3\} = 0$, а $\Pr\{A_1\}\Pr\{A_2\}\Pr\{A_3\} = 1/8 \neq 0$.

Теорема Байеса

Из определения условной вероятности (B.14) и коммутативности $A \cap B = B \cap$ $\cap A$ следует, что для двух событий A и B с ненулевыми вероятностями

$$\Pr\{A \cap B\} = \Pr\{B\}\Pr\{A \mid B\} = \Pr\{A\}\Pr\{B \mid A\}. \tag{B.16}$$

Разрешая уравнение относительно $\Pr\{A \mid B\}$, получим формулу

$$\Pr\{A \mid B\} = \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{B\}}, \tag{B.17}$$

известную как ***теорема Байеса*** (Bayes's theorem). Знаменатель $\Pr\{B\}$ представляет собой нормализующий член, который можно выразить иначе. Поскольку $B = (B \cap A) \cup \left(B \cap \bar{A}\right)$, а $B \cap A$ и $B \cap \bar{A}$ — взаимоисключающие события,

$$\Pr\{B\} = \Pr\{B \cap A\} + \Pr\left\{B \cap \bar{A}\right\} = \Pr\{A\}\Pr\{B \mid A\} + \Pr\left\{\bar{A}\right\}\Pr\left\{B \mid \bar{A}\right\}.$$

Подставляя полученное выражение в (B.17), получим эквивалентный вид формулы Байеса

$$\Pr\{A \mid B\} = \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{A\}\Pr\{B \mid A\} + \Pr\left\{\bar{A}\right\}\Pr\left\{B \mid \bar{A}\right\}}.$$

Теорема Байеса может упростить вычисление условной вероятности. Предположим, например, что у нас есть симметричная монета и монета, которая всегда

выпадает орлом. Мы случайным образом выбираем монету и два раза ее подбрасываем. Предположим, что оба раза выпал орел. Какова вероятность того, что мы выбрали несимметричную монету?

Эта задача легко решается при помощи формулы Байеса. Пусть A — событие, состоящее в том, что выбрана несимметричная монета, а B — выпадение двух орлов. Нам надо найти вероятность $\Pr\{A \mid B\}$ Так как $\Pr\{A\} = 1/2$, $\Pr\{B \mid A\} = 1$, $\Pr\left\{\bar{A}\right\} = 1/2$ и $\Pr\left\{B \mid \bar{A}\right\} = 1/4$, получаем

$$\Pr\{A \mid B\} = \frac{(1/2) \cdot 1}{(1/2) \cdot 1 + (1/2) \cdot (1/4)} = \frac{4}{5}.$$

Упражнения

В.2-1. Докажите ***неравенство Буля***: для любой конечной или бесконечной счетной последовательности событий $A_1, A_2, \ldots$ справедливо следующее неравенство:

$$\Pr\{A_1 \cup A_2 \cup \ldots\} \leqslant \Pr\{A_1\} + \Pr\{A_2\} + \ldots. \tag{В.18}$$

В.2-2. Профессор Ванстоун бросает симметричную монету один раз, а профессор Унопетри — два раза. Чему равна вероятность того, что профессор Ванстоун получит больше выпавших орлов, чем профессор Унопетри?

В.2-3. Имеется тщательно перетасованная колода из десяти карт, пронумерованных числами от 1 до 10. Из колоды последовательно вынимаются 3 карты. Какова вероятность того, что эти карты будут находиться в порядке возрастания их достоинства?

⋆ В.2-4. Разработайте процедуру, которая получает в качестве входных параметров два целых числа a и b $(0 < a < b)$ и, используя симметричную монету, имитирует бросание несимметричной монеты с появлением орла с вероятностью a/b, и решки — с вероятностью $(b-a)/b$. Оцените математическое ожидание количества бросков монеты (которое должно быть $O(1)$. (*Указание*: используйте бинарное представление числа a/b.)

В.2-5. Докажите, что $\Pr\{A \mid B\} + \Pr\left\{\bar{A} \mid B\right\} = 1$.

В.2-6. Докажите, что для любого множества событий $A_1, A_2, \ldots, A_n$

$$\Pr\{A_1 \cap A_2 \cap \ldots \cap A_n\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_n \mid A_1 \cap A_2 \cap \ldots \cap A_{n-1}\}.$$

⋆ В.2-7. Покажите, как построить множество из n попарно независимых событий так, чтобы никакое его подмножество из более чем двух элементов не являлось независимым в совокупности.

⋆ В.2-8. Два события A и B являются ***условно независимыми*** относительно события C, если $\Pr\{A \cap B \mid C\} = \Pr\{A \mid C\} \cdot \Pr\{B \mid C\}$. Приведите нетривиальный пример двух событий, которые не являются независимыми, но при этом условно независимы относительно некоторого третьего события.

⋆ В.2-9. Вы участвуете в игровом шоу, где приз спрятан за одним из трех занавесов. Вы выигрываете приз, если угадываете, где он находится. После того как вы выбираете занавес, ведущий открывает один из оставшихся. Если за ним приза нет, вы можете изменить свой выбор, указав на третий занавес. Как изменятся ваши шансы на выигрыш, если вы сделаете это?

⋆ В.2-10. Один из трех заключенных X, Y и Z будет освобожден, а оставшиеся будут отбывать наказание. Кто именно — знает охранник, но ему запрещено сообщать заключенным об их судьбе. Заключенный X просит охранника сообщить, кто из Y и Z останется в тюрьме, мотивируя это тем, что он и так знает, что один из них будет отбывать наказание, так что из ответа охранника он ничего не узнает о собственной судьбе. Охранник сообщает, что Y освобожден не будет. После этого X полагает, что его шансы на свободу, которые составляли $1/3$, возрастают до $1/2$, так как он знает, что освобожден будет только либо он, либо Z. Прав ли он, или его шансы на свободу остаются равными $1/3$? Обоснуйте свой ответ.

В.3 Дискретные случайные величины

Дискретная случайная величина (discrete random variable) X — это функция, отображающая конечное или бесконечное счетное пространство событий S на множество действительных чисел. Она связывает с каждым возможным исходом эксперимента действительное число, что позволяет нам работать с распределением вероятностей на множестве значений данной функции. Случайные величины могут быть определены и для несчетных бесконечных пространств событий, но при этом возникают технические проблемы, которых лучше избежать. Нам не придется работать с несчетными бесконечными пространствами событий, так что далее везде предполагается, что случайные величины дискретны.

Для случайной величины X и действительного числа x определим событие $X = x$ как $\{s \in S : X(s) = x\}$; таким образом,

$$\Pr\{X = x\} = \sum_{\{s \in S : X(s) = x\}} \Pr\{s\}.$$

Функция $f(x) = \Pr\{X = x\}$ называется ***функцией плотности вероятности*** (probability density function) случайной величины X. Из аксиом вероятности следует, что $\Pr\{X = x\} \geqslant 0$ и $\sum_x \Pr\{X = x\} = 1$.

В качестве примера рассмотрим эксперимент, состоящий в бросании пары игральных костей. В пространстве событий имеется 36 элементарных событий. Будем считать, что все они равновероятны, т.е. для каждого элементарного события $s \in S$ его вероятность $\Pr\{s\} = 1/36$. Определим случайную величину X как *максимальное* количество очков, выпавшее при броске на одной из костей. Тогда, например, $\Pr\{X = 3\} = 3/5$, так как X равно 3 в 5 из 36 возможных элементарных событий, а именно при выпадении $(1,3)$, $(2,3)$, $(3,3)$, $(3,2)$ или $(3,1)$.

Как правило, на одном пространстве событий определяются несколько случайных величин. Если X и Y — случайные величины, то функция

$$f(x,y) = \Pr\{X = x \text{ и } Y = y\}$$

называется ***совместной функцией плотности вероятности*** (joint probability density function) X и Y. Для фиксированного значения y

$$\Pr\{Y = y\} = \sum_x \Pr\{X = x \text{ и } Y = y\},$$

и, аналогично, для фиксированного значения x

$$\Pr\{X = x\} = \sum_y \Pr\{X = x \text{ и } Y = y\}.$$

Используя определение условной вероятности (В.14), мы имеем

$$\Pr\{X = x \mid Y = y\} = \frac{\Pr\{X = x \text{ и } Y = y\}}{\Pr\{Y = y\}}.$$

Две случайные величины X и Y являются ***независимыми***, если для всех x и y события $X = x$ и $Y = y$ независимы, т.е. для всех x и y выполняется $\Pr\{X = x \text{ и } Y = y\} = \Pr\{X = x\}\Pr\{Y = y\}$.

Для данного множества случайных величин, определенных на одном и том же пространстве событий, можно определить новую случайную величину как сумму, произведение или другую функцию имеющихся случайных величин.

Математическое ожидание случайной величины

Простейшая и наиболее часто используемая характеристика случайной величины — ее ***среднее*** значение (mean), или ***математическое ожидание*** (expected value, expectation), которое определяется для дискретной случайной величины следующим образом:

$$\mathrm{E}[X] = \sum_x x \Pr\{X = x\}.^2 \qquad \text{(В.19)}$$

[2] В отечественной математической литературе для математического ожидания принято обозначение $\mathrm{M}[X]$. — *Прим. ред.*

Это значение вполне определено в случае конечного или бесконечного абсолютно сходящегося ряда. Иногда математическое ожидание X обозначают как μ_X или, если случайная величина очевидна из контекста, просто μ.

Рассмотрим игру, в которой бросается пара симметричных монет. Вы выигрываете 3 доллара при выпадении орла, и проигрываете 2 доллара при выпадении решки. Математическое ожидание случайной величины X, равной вашему выигрышу, равно

$$\begin{aligned}\mathrm{E}[X] &= 6 \cdot \Pr\{\text{2 орла}\} + 1 \cdot \Pr\{\text{1 орел, одна решка}\} - 4 \cdot \Pr\{\text{2 решки}\} = \\ &= 6 \cdot (1/4) + 1 \cdot (1/2) - 4 \cdot (1/4) = 1.\end{aligned}$$

Математическое ожидание суммы двух случайных величин равно сумме их математических ожиданий:

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y], \tag{B.20}$$

если значения $\mathrm{E}[X]$ и $\mathrm{E}[Y]$ определены. Это свойство называется ***линейностью математического ожидания*** (linearity of expectation), причем оно справедливо даже тогда, когда случайные величины X и Y не являются независимыми. Это правило распространяется и на конечные, и на абсолютно сходящиеся бесконечные суммы математических ожиданий. Линейность математического ожидания является ключевым свойством, обеспечивающим возможность проведения вероятностного анализа с использованием индикаторной случайной величины (см. раздел 5.2).

Если X — произвольная случайная величина, то любая функция $g(x)$ определяет новую случайную величину $g(X)$. Если математическое ожидание этой величины определено, то

$$\mathrm{E}[g(X)] = \sum_x g(x) \Pr\{X = x\}.$$

Пусть $g(x) = ax$. Тогда для произвольной константы a

$$\mathrm{E}[aX] = a\mathrm{E}[X]. \tag{B.21}$$

Таким образом, математическое ожидание представляет собой линейную функцию: для двух произвольных случайных величин X и Y и произвольной константы a

$$\mathrm{E}[aX + Y] = a\mathrm{E}[X] + \mathrm{E}[Y]. \tag{B.22}$$

Если случайные величины X и Y независимы и каждая имеет определенное математическое ожидание, то

$$\begin{aligned}\mathrm{E}[XY] &= \sum_x \sum_y xy \Pr\{X = x \text{ и } Y = y\} = \\ &= \sum_x \sum_y xy \Pr\{X = x\} \Pr\{Y = y\} = \\ &= \left(\sum_x x \Pr\{X = x\}\right)\left(\sum_y y \Pr\{Y = y\}\right) = \\ &= \mathrm{E}[X]\,\mathrm{E}[Y].\end{aligned}$$

В общем случае, если у нас имеется n независимых в совокупности случайных величин $X_1, X_2, \ldots, X_n$, то

$$\mathrm{E}[X_1 X_2 \cdots X_n] = \mathrm{E}[X_1]\,\mathrm{E}[X_2] \cdots \mathrm{E}[X_n]. \tag{B.23}$$

Если случайная величина X принимает значения из множества неотрицательных целых чисел, то для ее математического ожидания имеется красивая формула:

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=0}^{\infty} i \Pr\{X = i\} = \\ &= \sum_{i=0}^{\infty} i \left(\Pr\{X \geqslant i\} - \Pr\{X \geqslant i+1\}\right) = \\ &= \sum_{i=1}^{\infty} \Pr\{X \geqslant i\},\end{aligned} \tag{B.24}$$

поскольку каждый член $\Pr\{X \geqslant i\}$ присутствует в сумме со знаком плюс i раз, и $i-1$ раз — со знаком минус (кроме члена $\Pr\{X \geqslant 0\}$, который в сумме отсутствует).

При применении выпуклой вниз функции $f(x)$ к случайной величине X ***неравенство Йенсена*** (Jensen's inequality) гласит, что

$$\mathrm{E}[f(X)] \geqslant f(\mathrm{E}[X]) \tag{B.25}$$

(в случае, когда математическое ожидание существует и конечно). Функция $f(x)$ называется ***выпуклой вниз*** (convex), если для всех x и y и всех $0 \leqslant \lambda \leqslant 1$ выполняется неравенство $f(\lambda x + (1-\lambda)y) \leqslant \lambda f(x) + (1-\lambda) f(y)$.

Дисперсия и стандартное отклонение

Математическое ожидание случайной величины не говорит нам о том, насколько сильно "разбросаны" ее значения. Например, если у нас есть случайные величины X и Y, такие что $\Pr\{X = 1/4\} = \Pr\{X = 3/4\} = 1/2$ и $\Pr\{Y = 0\} = \Pr\{Y = 1\} = 1/2$, то их математические ожидания равны $1/2$, однако реальные значения Y находятся дальше от математического ожидания этой случайной величины, чем в случае X.

Дисперсия (variance) случайной величины X с математическим ожиданием $E[X]$ определяется как[3]

$$\begin{aligned}\operatorname{Var}[X] &= \mathrm{E}\left[(X - \mathrm{E}[X])^2\right] = \\ &= \mathrm{E}\left[X^2 - 2X\mathrm{E}[X] + \mathrm{E}^2[X]\right] \\ &= \mathrm{E}\left[X^2\right] - \mathrm{E}^2[X].\end{aligned} \tag{B.26}$$

В процессе вывода использованы равенства $\mathrm{E}\left[\mathrm{E}^2[X]\right] = \mathrm{E}^2[X]$ и $\mathrm{E}[X\mathrm{E}[X]] = \mathrm{E}^2[X]$, которые вполне корректны, поскольку математическое ожидание $\mathrm{E}[X]$ является просто действительным числом, а не случайной величиной, так что мы можем воспользоваться формулой (B.21). Уравнение (B.26) можно использовать для поиска математического ожидания квадрата случайной величины:

$$\mathrm{E}\left[X^2\right] = \operatorname{Var}[X] + \mathrm{E}^2[X]. \tag{B.27}$$

Дисперсия случайной величины X связана с дисперсией случайной величины aX следующим соотношением (см. упражнение B.3-10):

$$\operatorname{Var}[aX] = a^2 \operatorname{Var}[X].$$

Если X и Y — независимые случайные величины, то

$$\operatorname{Var}[X + Y] = \operatorname{Var}[X] + \operatorname{Var}[Y].$$

В общем случае, если n случайных величин $X_1, X_2, \ldots, X_n$ попарно независимы, то

$$\operatorname{Var}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \operatorname{Var}[X_i]. \tag{B.28}$$

Стандартным отклонением (standard deviation) случайной величины X называется неотрицательный квадратный корень дисперсии X. Иногда стандартное отклонение X обозначают как σ_X или, если случайная величина очевидна из контекста, просто σ. При использовании этого обозначения дисперсия X записывается как σ^2.

[3] В отечественной математической литературе для дисперсии принято обозначение $\mathrm{D}[X]$. — *Прим. ред.*

Упражнения

В.3-1. Бросаются две обычные шестигранные игральные кости. Чему равно математическое ожидание суммы выпавших очков? Чему равно математическое ожидание максимального из двух выпадающих чисел?

В.3-2. Массив $A[1..n]$ содержит n различных чисел в произвольном порядке; все перестановки чисел равновероятны. Чему равно математическое ожидание индекса максимального элемента массива? Чему равно математическое ожидание индекса минимального элемента массива?

В.3-3. Игрок ставит доллар на одно из чисел от 1 до 6 и выбрасывает одновременно три игральные кости. Если указанное число не выпало ни на одной из костей, игрок теряет свой доллар; если же число выпало на k костях, игрок сохраняет свой доллар и получает дополнительно k долларов. Чему равно математическое ожидание выигрыша игрока в одной партии?

В.3-4. Покажите, что если X и Y — неотрицательные случайные величины, то $\mathrm{E}[\max(X, Y)] \leqslant \mathrm{E}[X] + \mathrm{E}[Y]$.

⋆ В.3-5. Пусть X и Y — независимые случайные величины. Докажите, что $f(X)$ и $g(Y)$ независимы при любом выборе функций f и g.

⋆ В.3-6. Пусть X — неотрицательная случайная величина, и математическое ожидание $\mathrm{E}[X]$ вполне определено. Докажите ***неравенство Маркова***

$$\Pr\{X \geqslant t\} \leqslant \mathrm{E}[X]/t \quad \text{для всех } t > 0. \tag{В.29}$$

⋆ В.3-7. Пусть S — пространство событий, а X и X' — случайные величины, такие что $X(s) \geqslant X'(s)$ для всех $s \in S$. Докажите, что для произвольной действительной константы t $\Pr\{X \geqslant t\} \geqslant \Pr\{X' \geqslant t\}$.

В.3-8. Что больше — математическое ожидание квадрата случайной величины или квадрат ее математического ожидания?

В.3-9. Покажите, что для любой случайной величины X, которая принимает только значения 0 и 1, справедливо соотношение

$$\mathrm{Var}[X] = \mathrm{E}[X]\,\mathrm{E}[1 - X]\ .$$

В.3-10. Докажите, используя определение дисперсии (В.26), что $\mathrm{Var}[aX] = a^2 \mathrm{Var}[X]$.

В.4 Геометрическое и биномиальное распределения

Бросание симметричной монеты — пример ***испытания Бернулли*** (Bernoulli trial), которое определяется как эксперимент с двумя возможными исходами —

успехом с вероятностью p и ***неудачей*** с вероятностью $q = 1 - p$. Когда мы говорим об испытаниях Бернулли, то подразумевается, что испытания независимы в совокупности и (если явно не оговорено иное) что вероятность успеха в каждом испытании равна p. С испытаниями Бернулли связаны два важных распределения вероятностей: геометрическое и биномиальное.

Геометрическое распределение

Предположим, что у нас есть последовательность испытаний Бернулли, вероятность успеха в каждом из которых равна p, а неудачи — $q = 1 - p$. Сколько испытаний будет проведено до того, как будет достигнут успех? Пусть случайная величина X равна количеству испытаний, необходимых для достижения успеха. Тогда X принимает значения $\{1, 2, \ldots\}$ и для $k \geqslant 1$

$$\Pr\{X = k\} = q^{k-1}p, \tag{B.30}$$

поскольку перед наступлением одного успешного испытания было выполнено $k - 1$ неуспешных. Распределение вероятности, удовлетворяющее уравнению (B.30), называется ***геометрическим распределением*** (geometric distribution). На рис. B.1 показан пример такого распределения.

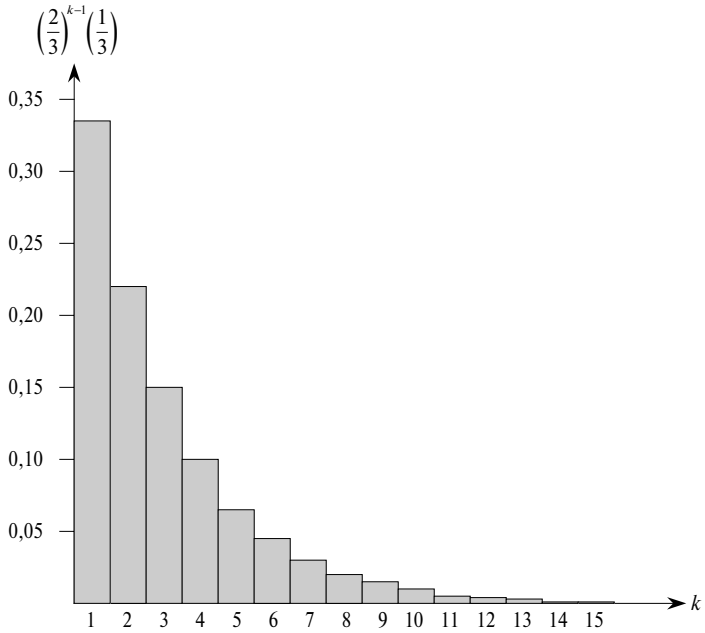


Рис. B.1. Геометрическое распределение с $p = 1/3$

Полагая, что $q < 1$, можно найти математическое ожидание геометрического распределения, воспользовавшись тождеством (А.8):

$$\mathrm{E}[X] = \sum_{k=1}^{\infty} kq^{k-1}p = \frac{p}{q}\sum_{k=0}^{\infty} kq^k = \frac{p}{q}\cdot\frac{q}{(1-q)^2} = \frac{1}{p}. \tag{В.31}$$

Таким образом, в среднем нужно выполнить $1/p$ испытаний до получения успеха (что интуитивно представляется вполне естественным). Дисперсия вычисляется аналогично, с использованием результата упражнения А.1-3, и равна

$$\mathrm{Var}[X] = q/p^2. \tag{В.32}$$

В качестве примера рассмотрим бросание двух кубиков до тех пор, пока мы не получим 7 или 11 очков. Из 36 возможных исходов бросания 6 дают нам семь очков, и 2 — одиннадцать. Таким образом, вероятность успеха равна $p = 8/36 = 2/9$, так что в среднем нам надо выбросить кости $1/p = 9/2 = 4.5$ раза для того, чтобы выпало 7 или 11 очков.

Биномиальное распределение

Какое количество из n испытаний Бернулли завершится успешно, если вероятность успеха равна p, а неудачи — $q = 1-p$? Определим случайную величину X как количество успехов в n испытаниях. Тогда X принимает значения $\{0, 1, \ldots, n\}$ и для $0 \leqslant k \leqslant n$

$$\Pr\{X = k\} = \binom{n}{k}p^k q^{n-k}, \tag{В.33}$$

поскольку существует $\binom{n}{k}$ способов выбрать k успешных испытаний из n, и вероятность каждого составляет $p^k q^{n-k}$. Распределение вероятностей (В.33) называется ***биномиальным распределением*** (binomial distribution). Для удобства для биномиального распределения используется запись

$$b(k; n, p) = \binom{n}{k}p^k (1-p)^{n-k}. \tag{В.34}$$

Пример биномиального распределения показан на рис. В.2. Название “биномиальное” связано с тем, что формула (В.33) описывает k-й член в разложении $(p+q)^n$. Следовательно, поскольку $p + q = 1$,

$$\sum_{k=0}^{n} b(k; n, p) = 1, \tag{В.35}$$

как того требует вторая аксиома вероятностей.

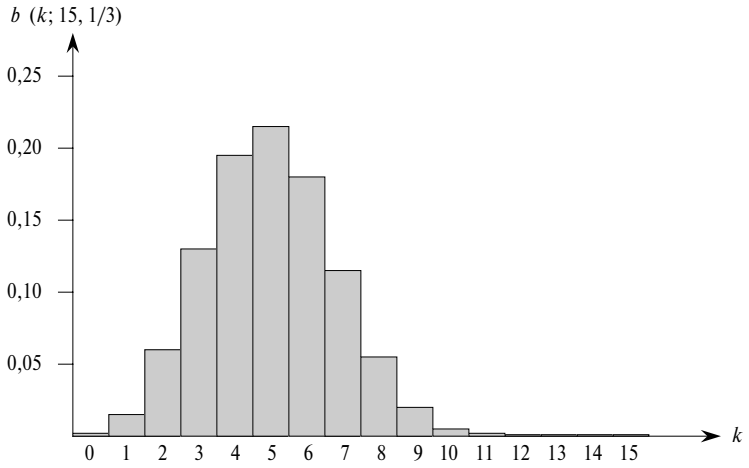


Рис. В.2. Биномиальное распределение $b(k; 15, 1/3)$

Вычислить математическое ожидание случайной величины, имеющей биномиальное распределение, можно с использованием формул (В.8) и (В.35). Пусть X — случайная величина с биномиальным распределением $b(k; n, p)$, и пусть $q = 1 - p$. По определению математического ожидания,

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{k=0}^{n} k \Pr\{X = k\} = \\
&= \sum_{k=0}^{n} k\, b(k; n, p) = \\
&= \sum_{k=1}^{n} k \binom{n}{k} p^k q^{n-k} = \\
&= np \sum_{k=1}^{n} \binom{n-1}{k-1} p^{k-1} q^{n-k} = \\
&= np \sum_{k=0}^{n-1} \binom{n-1}{k} p^k q^{(n-1)-k} = \\
&= np \sum_{k=0}^{n-1} b(k; n-1, p) = np
\end{aligned} \tag{В.36}$$

Используя линейность математического ожидания, мы можем получить тот же результат с существенно меньшим количеством выкладок. Пусть X_i — случайная величина, описывающая количество успешных случаев в i-м испытании. Тогда

$\mathrm{E}[X_i] = p \cdot 1 + q \cdot 0 = p$ и, согласно линейности математического ожидания (уравнение (В.20)), математическое ожидание числа успешных испытаний из общего количества n равно

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} p = np. \tag{В.37}$$

Такой же подход можно применить и для вычисления дисперсии распределения. Используя уравнение (В.26), получаем $\mathrm{Var}[X_i] = \mathrm{E}\left[X_i^2\right] - \mathrm{E}^2[X_i]$. Поскольку X_i может принимать только значения 0 или 1, имеем $\mathrm{E}\left[X_i^2\right] = \mathrm{E}[X_i] = p$, следовательно

$$\mathrm{Var}[X_i] = p - p^2 = pq. \tag{В.38}$$

Для вычисления дисперсии X воспользуемся независимостью всех n попыток. Таким образом, в соответствии с (В.28),

$$\mathrm{Var}[X] = \mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{Var}[X_i] = \sum_{i=1}^{n} pq = npq. \tag{В.39}$$

Как видно из рис. В.2, функция биномиального распределения $b(k; n, p)$ растет с ростом k до тех пор, пока k не достигает значения np, после чего начинает уменьшаться. Мы можем доказать, что биномиальное распределение всегда ведет себя подобным образом, рассматривая отношение двух последовательных членов:

$$\begin{aligned}
\frac{b(k; n, p)}{b(k-1; n, p)} &= \frac{\binom{n}{k} p^k q^{n-k}}{\binom{n}{k-1} p^{k-1} q^{n-k+1}} = \\
&= \frac{n!\,(k-1)!\,(n-k+1)!p}{k!\,(n-k)!n!q} = \\
&= \frac{(n-k+1)\,p}{kq} = \\
&= 1 + \frac{(n+1)\,p - k}{kq}.
\end{aligned} \tag{В.40}$$

Это отношение больше 1 тогда и только тогда, когда $(n+1)\,p - k$ положительно. Следовательно, $b(k; n, p) > b(k-1; n, p)$ при $k < (n+1)\,p$ (функция распределения возрастает) и $b(k; n, p) < b(k-1; n, p)$ при $k > (n+1)\,p$ (функция распределения убывает). Если $k = (n+1)\,p$ — целое число, то $b(k; n, p) = b(k-1; n, p)$ и функция распределения имеет два максимальных значения — при $k = (n+1)\,p$ и при $k - 1 = (n+1)\,p - 1 = np - q$. В противном случае она приминает максимальное значение при единственном целом значении k в диапазоне $np - q < k < (n+1)\,p$.

Следующая лемма дает верхнюю оценку биномиального распределения.

Лемма B.1. Пусть $n \geqslant 0$, $0 < p < 1$, $q = 1 - p$ и $0 \leqslant k \leqslant n$. Тогда

$$b(k; n, p) \leqslant \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}.$$

Доказательство. Используя уравнение (B.6), получаем:

$$\begin{aligned} b(k; n, p) &= \binom{n}{k} p^k q^{n-k} \leqslant \\ &\leqslant \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k} = \\ &= \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}. \end{aligned}$$

■

Упражнения

B.4-1. Убедитесь в выполнении аксиомы 2 для геометрического распределения.

B.4-2. Сколько раз в среднем надо бросить 6 симметричных монет для получения трех орлов и трех решек?

B.4-3. Покажите, что $b(k; n, p) = b(n - k; n, q)$, где $q = 1 - p$.

B.4-4. Покажите, что максимальное значение функции биномиального распределения $b(k; n, p)$ приближенно равно $1/\sqrt{2\pi npq}$, где $q = 1 - p$.

⋆ B.4-5. Покажите, что вероятность не получить ни одного успешного исхода в серии из n испытаний Бернулли с вероятностью успеха $p = 1/n$ составляет приблизительно $1/e$. Покажите также, что вероятность получения ровно одного успешного исхода также составляет примерно $1/e$.

⋆ B.4-6. Профессора Ванстоун и Унопетри бросают симметричную монету n раз. Покажите, что вероятность того, что они получат одинаковое количество орлов, равна $\binom{2n}{n}/4^n$. (*Указание*: считайте успехом выпадение орла у профессора Ванстоуна, а у профессора Унопетри — выпадение решки.) Воспользуйтесь вашей аргументацией для проверки тождества

$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n}.$$

⋆ B.4-7. Покажите, что при $0 \leqslant k \leqslant n$

$$b(k; n, 1/2) \leqslant 2^{nH(k/n)-n},$$

где $H(x)$ — энтропийная функция (B.7).

⋆ В.4-8. Рассмотрим n испытаний Бернулли, где p_i — вероятность успеха в i-м испытании, а X — случайная величина, равная общему количеству успехов. Пусть $p \geqslant p_i$ для всех $i = 1, 2, \ldots, n$. Докажите, что для $1 \leqslant k \leqslant n$

$$\Pr\{X < k\} \geqslant \sum_{i=0}^{k-1} b(i; n, p).$$

⋆ В.4-9. Пусть X — случайная величина, равная общему количеству успехов в множестве A из n испытаний Бернулли, а p_i — вероятность успеха в i-м испытании. Пусть X' — аналогичная случайная величина, равная общему количеству успехов в множестве A' из n испытаний Бернулли, где $p'_i \geqslant p_i$ — вероятность успеха в i-м испытании. Докажите, что для $0 \leqslant k \leqslant n$

$$\Pr\{X' \geqslant k\} \geqslant \Pr\{X \geqslant k\}.$$

(*Указание*: покажите, как получить результаты испытаний Бернулли A' из эксперимента, включающего испытания A, а затем воспользуйтесь результатом упражнения В.3-7.)

⋆ В.5 Хвосты биномиального распределения

Во многих задачах требуется определить не вероятность того, что в n испытаниях Бернулли будет получено ровно k успешных исходов, а вероятность того, что будет получено не более (или не менее) k успешных исходов. В этом разделе мы рассмотрим ***хвосты*** (tails) биномиального распределения, т.е. области распределения $b(k; n, p)$, далекие от среднего значения np, и найдем некоторые важные оценки для них.

Здесь мы будем рассматривать правый хвост распределения $b(k; n, p)$; левый хвост получается при простой взаимной замене успешных исходов на неудачи.

Теорема В.2. Рассмотрим последовательность из n испытаний Бернулли с вероятностью успешного исхода каждого испытания, равной p. Пусть X — случайная величина, равная общему количеству успешных исходов. Тогда для $0 \leqslant k \leqslant n$ вероятность того, что будет получено как минимум k успешных исходов, равна

$$\Pr\{X \geqslant k\} = \sum_{i=k}^{n} b(i; n, p) \leqslant \binom{n}{k} p^k.$$

Доказательство. Для множества $S \subseteq \{1, 2, \ldots, n\}$ обозначим через A_S событие, которое заключается в том, что i-я попытка успешна для всех $i \in S$. Понятно, что

если $|S| = k$, то $\Pr\{A_S\} = p^k$. Таким образом,

$$\begin{aligned}\Pr\{X \geqslant k\} &= \Pr\{\text{Существует } S \subseteq \{1,2,\ldots,n\} : |S| = k \text{ и } A_S\} = \\ &= \Pr\left\{\bigcup_{S\subseteq\{1,2,\ldots,n\}:|S|=k} A_S\right\} \leqslant \\ &\leqslant \sum_{S\subseteq\{1,2,\ldots,n\}:|S|=k} \Pr\{A_S\} = \binom{n}{k} p^k,\end{aligned}$$

где использованное неравенство вытекает из неравенства Буля (B.18). ■

Приведенное далее следствие просто переформулирует эту теорему для левого хвоста биномиального распределения. Все доказательства переформулированных таким образом (для противоположного хвоста) теорем далее в этом приложении предлагаются читателю в качестве самостоятельного упражнения.

Следствие B.3. Рассмотрим последовательность из n испытаний Бернулли с вероятностью успешного исхода каждого испытания, равной p. Пусть X — случайная величина, равная общему количеству успешных исходов. Тогда для $0 \leqslant k \leqslant n$ вероятность того, что будет получено не более k успешных исходов, равна

$$\Pr\{X \leqslant k\} = \sum_{i=0}^{k} b(i;n,p) \leqslant \binom{n}{n-k}(1-p)^{n-k} = \binom{n}{k}(1-p)^{n-k}.$$

■

Следующая рассматриваемая оценка относится к левому хвосту биномиального распределения.

Теорема B.4. Рассмотрим последовательность из n испытаний Бернулли с вероятностью успешного исхода каждого испытания, равной p, и вероятностью неудачи, равной $q = 1 - p$. Пусть X — случайная величина, равная общему количеству успешных исходов. Тогда для $0 < k < np$ вероятность того, что будет получено менее k успешных исходов, равна

$$\Pr\{X < k\} = \sum_{i=0}^{k-1} b(i;n,p) < \frac{kq}{np-k} b(k;n,p).$$

Доказательство. Ограничим ряд $\sum_{i=0}^{k-1} b(i;n,p)$ геометрическим рядом с использованием методики из раздела A.2. Для $i = 1, 2, \ldots, k$ из (B.40) получим:

$$\frac{b(i-1;n,p)}{b(i;n,p)} = \frac{iq}{(n-i+1)p} < \frac{iq}{(n-i)p} \leqslant \frac{kq}{(n-k)p}.$$

Вводя обозначение

$$x = \frac{kq}{(n-k)\,p} < \frac{kq}{(n-np)\,p} = \frac{kq}{nqp} = \frac{k}{np} < 1$$

получим, что

$$b\,(i-1;n,p) < xb\,(i;n,p)$$

для $0 < i \leqslant k$. Применяя это неравенство итеративно $k-i$ раз, мы получим

$$b\,(i;n,p) < x^{k-i}b\,(k;n,p)$$

для $0 \leqslant i < k$ и, следовательно,

$$\begin{aligned}\sum_{i=0}^{k-1} b\,(i;n,p) < \sum_{i=0}^{k-1} x^{k-i}b\,(k;n,p) < \\ < b\,(k;n,p)\sum_{i=1}^{\infty} x^i = \\ = \frac{x}{1-x}b\,(k;n,p) = \\ = \frac{kq}{np-k}b\,(k;n,p)\,.\end{aligned}$$

■

Следствие В.5. Рассмотрим последовательность из n испытаний Бернулли с вероятностью успешного исхода каждого испытания, равной p, и вероятностью неудачи, равной $q = 1-p$. Тогда для $0 < k \leqslant np/2$ вероятность того, что будет получено менее k успешных исходов, составляет менее половины от вероятности получения менее $k+1$ успешного исхода.

Доказательство. Поскольку $k \leqslant np/2$, мы имеем

$$\frac{kq}{np-k} \leqslant \frac{(np/2)\,q}{np-(np/2)} = \frac{(np/2)\,q}{np/2} \leqslant 1,$$

поскольку $q \leqslant 1$. Пусть X — случайная величина, равная общему количеству успешных исходов. Из теоремы В.4 следует, что вероятность получить менее k успешных исходов равна

$$\Pr\{X < k\} = \sum_{i=0}^{k-1} b\,(i;n,p) < b\,(k;n,p)\,.$$

Таким образом,

$$\frac{\Pr\{X < k\}}{\Pr\{X < k+1\}} = \frac{\sum_{i=0}^{k-1} b\,(i;n,p)}{\sum_{i=0}^{k} b\,(i;n,p)} = \frac{\sum_{i=0}^{k-1} b\,(i;n,p)}{\sum_{i=0}^{k-1} b\,(i;n,p) + b\,(k;n,p)} < \frac{1}{2},$$

поскольку $\sum_{i=0}^{k-1} b\,(i;n,p) < b\,(k;n,p)$. ■

Оценка для правого хвоста выполняется аналогично. Ее доказательство оставлено читателю в качестве упражнения B.5-2.

Следствие B.6. Рассмотрим последовательность из n испытаний Бернулли с вероятностью успешного исхода каждого испытания, равной p. Пусть X — случайная величина, равная общему количеству успешных исходов. Тогда для $np < k < n$ вероятность более чем k успешных исходов равна

$$\Pr\{X > k\} = \sum_{i=k+1}^{n} b(i; n, p) < \frac{(n-k)\,p}{k-np} b(k; n, p).$$

■

Следствие B.7. Рассмотрим последовательность из n испытаний Бернулли с вероятностью успешного исхода каждого испытания, равной p, и вероятностью неудачи, равной $q = 1 - p$. Тогда для $(np + n)/2 < k < n$ вероятность более чем k успешных исходов не превышает половины вероятности более чем $k - 1$ успешных исходов. ■

В следующей теореме рассматриваются n испытаний Бернулли; вероятность успеха в i-м испытании составляет p_i. Как видно из следствия из данной теоремы, ее можно использовать для оценки правого хвоста биномиального распределения, положив $p_i = p$ для всех испытаний.

Теорема B.8. Рассмотрим последовательность из n испытаний Бернулли, в которой в i-м испытании ($i = 1, 2, \ldots, n$) вероятность успеха равна p_i, а неудачи — $q_i = 1 - p_i$. Пусть X — случайная величина, равная общему количеству успешных исходов, а $\mu = \mathrm{E}[X]$. Тогда для $r > \mu$

$$\Pr\{X - \mu \geqslant r\} \leqslant \left(\frac{\mu e}{r}\right)^r.$$

Доказательство. Поскольку при любом $\alpha > 0$ функция $e^{\alpha x}$ — строго возрастающая,

$$\Pr\{X - \mu \geqslant r\} = \Pr\left\{e^{\alpha(X-\mu)} \geqslant e^{\alpha r}\right\}, \tag{B.41}$$

где значение α будет определено позже. Используя неравенство Маркова (B.29), мы получим

$$\Pr\left\{e^{\alpha(X-\mu)} \geqslant e^{\alpha r}\right\} \leqslant \mathrm{E}\left[e^{\alpha(X-\mu)}\right] e^{-\alpha r}. \tag{B.42}$$

Теперь нам надо оценить величину $\mathrm{E}\left[e^{\alpha(X-\mu)}\right]$ и найти подходящее значение α для уравнения (B.42). Начнем с вычисления $\mathrm{E}\left[e^{\alpha(X-\mu)}\right]$. Используя обозначения, принятые в разделе 5.2, мы можем записать

$$X_i = \mathrm{I}\{\text{Исход } i\text{-го испытания Бернулли успешен}\}$$

для $i = 1, 2, \ldots, n$; т.е. X_i — случайная величина, принимающая значение 1 в случае успеха и 0 — в случае неудачи. Таким образом,

$$X = \sum_{i=1}^{n} X_i$$

и, вследствие линейности математического ожидания,

$$\mu = \mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} p_i,$$

откуда следует, что

$$X - \mu = \sum_{i=1}^{n} (X_i - p_i).$$

Для вычисления выражения $\mathrm{E}\left[e^{\alpha(X-\mu)}\right]$ подставим в него полученную формулу для $X - \mu$ и получим:

$$\mathrm{E}\left[e^{\alpha(X-\mu)}\right] = \mathrm{E}\left[e^{\alpha\sum_{i=1}^{n}(X_i-p_i)}\right] = \mathrm{E}\left[\prod_{i=1}^{n} e^{\alpha(X_i-p_i)}\right] = \prod_{i=1}^{n} \mathrm{E}\left[e^{\alpha(X_i-p_i)}\right],$$

что следует из (В.23), поскольку случайные величины X_i являются независимыми в совокупности, что влечет независимость в совокупности случайных величин $e^{\alpha(X_i-p_i)}$ (см. упражнение В.3-5). Из определения математического ожидания

$$\begin{aligned} \mathrm{E}\left[e^{\alpha(X_i-p_i)}\right] &= e^{\alpha(1-p_i)}p_i + e^{(0-p_i)}q_i = \\ &= p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leqslant \\ &\leqslant p_i e^{\alpha} + 1 \leqslant \exp\left(p_i e^{\alpha}\right), \end{aligned} \tag{В.43}$$

где $\exp(x)$ обозначает экспоненциальную функцию: $\exp(x) = e^x$. (Неравенство (В.43) следует из того, что $\alpha > 0$, $q_i \leqslant 1$, $e^{\alpha q_i} \leqslant e^{\alpha}$ и $e^{-\alpha p_i} \leqslant 1$, а последний переход в (В.43) выполнен в соответствии с неравенством (3.11).) Следовательно,

$$\begin{aligned} \mathrm{E}\left[e^{\alpha(X-\mu)}\right] &= \prod_{i=1}^{n} \mathrm{E}\left[e^{\alpha(X_i-p_i)}\right] \leqslant \prod_{i=1}^{n} \exp\left(p_i e^{\alpha}\right) = \\ &= \exp\left(\sum_{i=1}^{n} p_i e^{\alpha}\right) = \exp\left(\mu e^{\alpha}\right), \end{aligned} \tag{В.44}$$

поскольку $\mu = \sum_{i=1}^{n} p_i$. Таким образом, из (В.41), (В.42) и (В.44) мы получаем, что

$$\Pr\{X - \mu \geqslant r\} \leqslant \exp\left(\mu e^{\alpha} - \alpha r\right). \tag{В.45}$$

Выбирая $\alpha = \ln(r/\mu)$ (см. упражнение B.5-7), мы получим:

$$\begin{aligned}\Pr\{X - \mu \geqslant r\} &\leqslant \exp\left(\mu e^{\ln(r/\mu)} - r\ln(r/\mu)\right) = \\ &= \exp(r - r\ln(r/\mu)) = \\ &= \frac{e^r}{(r/\mu)^r} = \left(\frac{\mu e}{r}\right)^r.\end{aligned}$$

■

Применяя эту теорему к последовательности n испытаний Бернулли с равной вероятностью успеха, мы получим оценку для правого хвоста биномиального распределения.

Следствие B.9. Рассмотрим последовательность n испытаний Бернулли с равной вероятностью успеха p и неудачи $q = 1 - p$ в каждом испытании. Тогда для $r > np$

$$\Pr\{X - np \geqslant r\} = \sum_{k=\lceil np+r\rceil}^{n} b(k; n, p) \leqslant \left(\frac{npe}{r}\right)^r.$$

Доказательство. В соответствии с (B.36), $\mu = \mathrm{E}[X] = np$. ■

Упражнения

⋆ B.5-1. Что менее вероятно: при бросании симметричной монеты n раз не получить ни одного орла или получить менее n орлов при бросании монеты $4n$ раз?

⋆ B.5-2. Докажите следствия B.6 и B.7.

⋆ B.5-3. Покажите, что

$$\sum_{i=0}^{k-1}\binom{n}{i}a^i < (a+1)^n \frac{k}{na - k(a+1)} b(k; n, a/(a+1))$$

для всех $a > 0$ и всех $0 < k < n$.

⋆ B.5-4. Докажите, что если $0 < k < np$, где $0 < p < 1$ и $q = 1 - p$, то

$$\sum_{i=0}^{k-1} p^i q^{n-i} < \frac{kq}{np - k}\left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}.$$

⋆ B.5-5. Покажите, что из условий теоремы B.8 следует, что

$$\Pr\{\mu - X \geqslant r\} \leqslant \left(\frac{(n-\mu)e}{r}\right)^r,$$

а из условий следствия B.9 —

$$\Pr\{np - X \geqslant r\} \leqslant \left(\frac{nqe}{r}\right)^r.$$

⋆ В.5-6. Рассмотрим последовательность n испытаний Бернулли, в которой в i-м испытании ($i = 1, 2, \ldots, n$) вероятность успеха равна p_i, а неудачи — $q_i = 1 - p_i$. Пусть X — случайная величина, равная общему количеству успешных исходов, а $\mu = \mathrm{E}[X]$. Покажите, что для $r \geqslant 0$

$$\Pr\{X - \mu \geqslant r\} \leqslant e^{-r^2/2n}.$$

(*Указание*: докажите, что $p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leqslant e^{\alpha^2/2}$, а затем следуйте схеме доказательства теоремы В.8, используя это неравенство вместо (В.43).)

⋆ В.5-7. Покажите, что правая сторона неравенства (В.45) при выборе $\alpha = \ln(r/\mu)$ принимает минимальное значение.

Задачи

В-1. **Шары и корзины**

В этой задаче мы рассмотрим размещение n шаров по b различным корзинам.

а) Предположим, что все n шаров различны, а их порядок в корзине не имеет значения. Докажите, что имеется b^n способов разместить шары в корзинах.

б) Предположим, что все n шаров различны, а их порядок в корзине существенен. Докажите, что имеется ровно $(b + n - 1)!/(b - 1)!$ способов размещения шаров в корзинах. (*Указание*: подсчитайте количество способов разместить в ряд n различных шаров и $b - 1$ неразличимых разделителей.)

в) Предположим, что все шары идентичны, а следовательно, их порядок в корзине не имеет значения. Покажите, что количество способов, которыми можно разместить шары в корзинах, равно $\binom{b+n-1}{n}$. (*Указание*: воспользуйтесь тем же способом, что и в части *б*, только теперь шары также неразличимы.)

г) Покажите, что если шары идентичны, и ни в одной корзине не может находиться больше одного шара, то разместить шары в корзинах можно $\binom{b}{n}$ способами.

д) Покажите, что если шары идентичны и ни одна корзина не должна остаться пустой, то разместить шары в корзинах можно $\binom{n-1}{b-1}$ способами.

Заключительные замечания

Общие методы решения вероятностных задач впервые обсуждались в знаменитой переписке Паскаля (B. Pascal) и Ферма (P. de Fermat), начавшейся в 1654 году, и в книге Гюйгенса (C. Huygens, 1657 год). Строгая теория вероятности началась с работ Бернулли (J. Bernoulli, 1713 год) и Муавра (A. de Moivre, 1730 год). Дальнейшее развитие теории вероятности связано с именами Лапласа (P.S. de Laplace), Пуассона (S.-D. Poisson) и Гаусса (C.F. Gauss).

Суммы случайных величин исследовались П.Л. Чебышевым и А.А. Марковым. Аксиоматизация теории вероятности была выполнена А.Н. Колмогоровым в 1933 году. Оценки хвостов распределений приведены в работах Чернова (Chernoff) [59] и Хеффдинга (Hoeffding) [150]. Важные результаты о случайных комбинаторных структурах принадлежат Эрдешу (P. Erdös).

Материал по элементарной комбинаторике можно найти в книгах Кнута (Knuth) [182] и Лю (Liu) [205]; по теории вероятности — в книгах Биллингсли (Billingsley) [42], Чанга (Chung) [60], Дрейка (Drake) [80], Феллера (Feller) [88] и Розанова (Rozanov) [263].

Библиография

[1] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions*. Dover, 1965.

[2] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An Algorithm for the Organization of Information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[3] Leonard M. Adleman, Carl Pomerance and Robert S. Rumely. On Distinguishing Prime Numbers from Composite Numbers. *Annals of Mathematics*, 117:173–206, 1983.

[4] Alok Aggarwal and Jeffrey Scott Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[6] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

А. Ахо, Д. Хопкрофт, Д. Ульман. *Структуры данных и алгоритмы*. — М.: "Вильямс", 2000.

[7] Ravindra K. Ahuja, Thomas L. Magnanti and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[8] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin and Robert E. Tarjan. Faster Algorithms for the Shortest Path Problem. *Journal of the ACM*, 37:213–223, 1990.

[9] Ravindra K. Ahuja and James B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5):748–759, 1989.

[10] Ravindra K. Ahuja, James B. Orlin and Robert E. Tarjan. Improved Time Bounds for the Maximum Flow Problem. *SIAM Journal on Computing*, 18(5):939–954, 1989.

[11] Miklos Ajtai, János Komlós and Endre Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[12] Miklos Ajtai, Nimrod Megiddo and Orli Waarts. Improved Algorithms and Analysis for ecretary Problems and Generalizations. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 473–482, 1995.

[13] Mohamad Akra and Louay Bazzi. On the Solution of Linear Recurrence Equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.

[14] Noga Alon. Generating pseudo-random permutations and maximum flow algorithms. *Information Processing Letters*, 35:201–204, 1990.

[15] Arne Andersson. Balanced Search Trees Made Simple. In *Proceedings of the Third Workshop on Algorithms and Data Structures*, number 709 in Lecture Notes in Computer Science, pages 60–71. Springer-Verlag, 1993

[16] Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 135–141, 1996

[17] Arne Andersson, Torben Hagerup, Stefan Nilsson and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57:74–93, 1998.

[18] Tom M. Apostol. *Calculus*, volume 1. Blaisdell Publishing Company, second edition, 1967.

[19] Sanjeev Arora. *Probabilistic checking of proofs and the hardness of approximation problems*. PhD thesis, University of California, Berkeley, 1994.

[20] Sanjeev Arora. The Approximability of NP-hard Problems. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 337–348,1998.

[21] Sanjeev Arora. Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and Other Geometric Problems. *Journal of the ACM*, 45(5):753–782, 1998.

[22] Sanjeev Arora and Carsten Lund. Hardness of Approximations. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 399–446. PWS Publishing Company, 1997.

[23] Javed A. Aslam. A Simple Bound on the Expected Height of a Randomly Built Binary Search Tree. Technical Report TR2001-387, Dartmouth College Department of Computer Science, 2001.

[24] Mikhail J. Atallah. *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[25] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, 1999.

[26] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, third edition, 1988.

[27] Eric Bach. Private communication, 1989.

[28] Eric Bach. Number-Theoretic Algorithms. In *Annual Review of Computer Science*, volume 4, pages 119–172. Annual Reviews, Inc., 1990.

[29] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory — Volume I: Efficient Algorithms*. The MIT Press, 1996.

[30] David H. Bailey, King Lee and Horst D. Simon. Using Strassen's Algorithm to Accelerate the Solution of Linear Systems. *The Journal of Supercomputing*, 4(4):357–371, 1990.

[31] R. Bayer. Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1:290-306, 1972.

[32] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972.

[33] Pierre Beauchemin, Gilles Brassard, Claude Crepeau, Claude Goutier and Carl Pomerance. The Generation of Random Numbers That Are Probably Prime. *Journal of Cryptology*, 1:53–64, 1988.

[34] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[35] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[36] Michael Ben-Or. Lower Bounds for Algebraic Computation Trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 80–86, 1983.

[37] Michael A. Bender, Erik D. Demaine and Martin Farach-Colton. Cache-Oblivious B-Trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, 2000.

[38] Samuel W. Bent and John W. John. Finding the Median Requires $2n$ Comparisons. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 213–216, 1985.

[39] Jon L. Bentley. *Writing Efficient Programs*. Prentice-Hall, 1982.

[40] Jon L. Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[41] Jon L. Bentley, Dorothea Haken and James B. Saxe. A General Method For Solving Divide-and-conquer Recurrences. *SIGACT News*, 12(3):36–44, 1980.

[42] Patrick Billingsley. *Probability and Measure*. John Wiley & Sons, second edition, 1986.

[43] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest and Robert E. Tarjan. Time Bounds for Selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[44] Bela Bollobas. *Random Graphs*. Academic Press, 1985.

[45] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. American Elsevier, 1976.

[46] Gilles Brassard and Paul Bratley. *Algorithmics: Theory and Practice*. Prentice-Hall, 1988.

[47] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996.

[48] Richard P. Brent. An Improved Monte Carlo Factorization Algorithm. *BIT*, 20(2):176–184, 1980.

[49] Mark R. Brown. *The Analysis of a Practical and Nearly Optimal Priority Queue*. PhD thesis, Computer Science Department, Stanford University, 1977. Technical Report STAN-CS-77-600.

[50] Mark R. Brown. Implementation and Analysis of Binomial Queue Algorithms. *SIAM Journal on Computing*, 7(3):298–319, 1978.

[51] J.P. Buhler, H.W. Lenstra, Jr., and Carl Pomerance. Factoring integers with the number field sieve. In A.K. Lenstra and H.W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 50–94. Springer-Verlag, 1993.

[52] J. Lawrence Carter and Mark N. Wegman. Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[53] Bernard Chazelle. A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[54] Joseph Cheriyan and Torben Hagerup. A Randomized Maximum-Flow Algorithm. *SIAM Journal on Computing*, 24(2):203–226, 1995.

[55] Joseph Cheriyan and S. N. Maheshwari. Analysis of Preflow Push Algorithms For Maximum Network Flow. *SIAM Journal on Computing*, 18(6):1057–1086, 1989.

[56] Boris V. Cherkasky and Andrew V. Goldberg. On Implementing the Push-relabel Method for the Maximum Flow Problem. *Algorithmica*, 19(4):390–410, 1997.

[57] Boris V. Cherkassky, Andrew V. Goldberg and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996.

[58] Boris V. Cherkassky, Andrew V. Goldberg and Craig Silverstein. Buckets, Heaps, Lists And Monotone Priority Queues. *SIAM Journal on Computing*, 28(4):1326–1346, 1999.

[59] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–507, 1952.

[60] Kai Lai Chung. *Elementary Probability Theory with Stochastic Processes*. Springer-Verlag, 1974.

[61] V. Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[62] V. Chvatal. *Linear Programming*. W. H. Freeman and Company, 1983.

[63] V. Chvatal, D. A. Klarner and D. E. Knuth. Selected Combinatorial Research Problems. Technical Report STAN-CS-72-292, Computer Science Department, Stanford University, 1972.

[64] Alan Cobham. The Intrinsic Computational Difficulty of Functions. In *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30. North-Holland, 1964.

[65] H. Cohen and H. W. Lenstra, Jr. Primality Testing and Jacobi Sums. *Mathematics of Computation*, 42(165):297–330, 1984.

[66] D. Comer. The Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[67] Stephen Cook. The Complexity of Theorem Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[68] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[69] Don Coppersmith. Modifications to the Number Field Sieve. *Journal of Cryptology*, 6:169–180, 1993.

[70] Don Coppersmith and Shmuel Winograd. Matrix Multiplication Via Arithmetic Progression. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[71] Thomas H. Cormen. *Virtual Memory for Data-Parallel Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1992.

[72] Eric V. Denardo and Bennett L. Fox. Shortest-Route Methods: 1. Reaching, Pruning, and Buckets. *Operations Research*, 27(1):161–186, 1979.

[73] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert and Robert E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[74] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[75] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[76] E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation. *Soviet Mathematics Doklady*, 11(5):1277–1280, 1970.

[77] Brandon Dixon, Monika Rauch and Robert E. Tarjan. Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[78] John D. Dixon. Factorization and Primality Tests. *The American Mathematical Monthly*, 91(6):333–352, 1984.

[79] Dorit Dor and Uri Zwick. Selecting the Median. In *Preceedings of th 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 28–37, 1995.

[80] Alvin W. Drake. *Fundamentals of Applied Probability Theory*. McGraw-Hill, 1967.

[81] James R. Driscoll, Harold N. Gabow, Ruth Shrairman and Robert E. Tarjan. Relaxed Heaps: An Alternative to Fibonacci Heaps With Applications to Parallel Computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[82] James R. Driscoll, Neil Sarnak, Daniel D. Sleator and Robert E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[83] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*. Volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1987.

[84] Jack Edmonds. Paths, Trees, and Flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[85] Jack Edmonds. Matroids and the Greedy Algorithm. *Mathematical Programming*, 1:126–136, 1971.

[86] Jack Edmonds and Richard M. Karp. Theoretical Improvements in the Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19:248–264, 1972.

[87] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

[88] William Feller. *An Introduction to Probability Theory and Its Applications*.*Third edition*. John Wiley & Sons, 1968.

[89] Robert W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.

[90] Robert W. Floyd. Algorithm 245 (TREESORT). *Communications of the ACM*, 7:701, 1964.

[91] Robert W. Floyd. Permuting Information in Idealized Two-Level Storage. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 105–109, Plenum Press, 1972.

[92] Robert W. Floyd and Ronald L. Rivest. Expected Time Bounds for Selection. *Communications of the ACM*, 18(3):165–172, 1975.

[93] Lestor R. Ford, Jr., and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[94] Lestor R. Ford, Jr., and Selmer M. Johnson. A Tournament Problem. *The American Mathematical Monthly*, 66:387–389, 1959.

[95] Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976.

[96] Michael L. Fredman, Janos Komlos and Endre Szemeredi. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *Journal of the ACM*, 31(3):538–544, 1984.

[97] Michael L. Fredman and Michael E. Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, 1989.

[98] Michael L. Fredman and Robert E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[99] Michael L. Fredman and Dan E. Willard. Surpassing the Information Theoretic Bound with Fusion Trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[100] Michael L. Fredman and Dan E. Willard. Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.

[101] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74:107–114, 2000.

[102] Harold N. Gabow, Z. Galil, T. Spencer and Robert E. Tarjan. Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs. *Combinatorica*, 6(2):109–122, 1986.

[103] Harold N. Gabow and Robert E. Tarjan. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[104] Harold N. Gabow and Robert E. Tarjan. Faster Scaling Algorithms for Network Problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.

[105] Zvi Galil and Oded Margalit. All Pairs Shortest Distances for Graphs with Small Integer Length Edges. *Information and Computation*, 134(2):103–139, 1997.

[106] Zvi Galil and Oded Margalit. All Pairs Shortest Paths for Graphs with Small Integer Length edges. *Journal of Computer and System Sciences*, 54(2):243–254, 1997.

[107] Zvi Galil and Joel Seiferas. Time-Space-Optimal String Matching. *Journal of Computer and System Sciences*, 26(3):280–294, 1983.

[108] Igal Galperin and Ronald L. Rivest. Scapegoat Trees. In *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174, 1993.

[109] Michael R. Garey, R. L. Graham and J. D. Ullman. Worst-case analyis of memory allocation algorithms. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, pages 143–150, 1972.

[110] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[111] Saul Gass. *Linear Programming: Methods and Applications.Fourth edition*. International Thomson Publishing, 1975.

[112] Fanica Gavril. Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

[113] Alan George and Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.

[114] E. N. Gilbert and E. F. Moore. Variable-Length Binary Encodings. *Bell System Technical Journal*, 38(4):933–967, 1959.

[115] Michel X. Goemans and David P. Williamson. Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming. *Journal of the ACM*, 42(6):1115–1145, 1995.

[116] Michel X. Goemans and David P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. In Dorit Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 144–191. PWS Publishing Company, 1997.

[117] Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.

[118] Andrew V. Goldberg. Scaling Algorithms for the Shortest Paths Problem. *SIAM Journal on Computing*, 24(3):494–504, 1995.

[119] Andrew V. Goldberg, Eva Tardos and Robert E. Tarjan. Network Flow Algorithms. In Bernhard Korte, Laszlo Lovász, Hans Jürgen Prömel and Alexander Schrijver, editors. *Paths, Flows, and VLSI-Layout*, pages 101–164. Springer-Verlag, 1990.

[120] Andrew V. Goldberg and Satish Rao. Beyond the Flow Decomposition Barrier. *Journal of the ACM*, 45:783–797, 1998.

[121] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[122] D. Goldfarb and M.J. Todd. Linear Programming. In G.L. Nemhauser, A.H.G. Rinnooy-Kan and M.J. Todd, editors. *Handbook in Operations Research and Management Science, Vol. 1, Optimization*, pages 73–170. Elsevier Science Publishers,1989.

[123] Shafi Goldwasser and Silvio Micali. Probabilistic Encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[124] Shafi Goldwasser, Silvio Micali and Charles Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[125] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1983.

[126] G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1984.

[127] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992.

[128] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.

[129] Ronald L. Graham. Bounds for certain multiprocessor anomalies. *Bell Systems Technical Journal*, 45:1563–1581, 1966.

[130] Ronald L. Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters*, 1:132–133, 1972.

[131] Ronald L. Graham and Pavol Hell. On the History of the Minimum Spanning Tree Problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[132] Ronald L. Graham, Donald E. Knuth and Oren Patashnik. *Concrete Mathematics. Second edition*. Addison-Wesley, 1994.

Р. Грэхем, Д. Кнут, О. Паташник. *Конкретная математика. Основание информатики*. — М.: Мир, 1998.

[133] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[134] M. Grötschel, László Lovász and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, 1988.

[135] Leo J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.

[136] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[137] Yijie Han. Improved Fast Integer Sorting in Linear Space. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 793–796, 2001.

[138] Frank Harary. *Graph Theory*. Addison-Wesley, 1969.

[139] Gregory C. Harfst and Edward M. Reingold. A Potential-Based Amortized Analysis of the Union-Find Data Structure. *SIGACT News*, 31(3):86–95, 2000.

[140] J. Hartmanis and R. E. Stearns. On the Computational Complexity of Algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[141] Michael T. Heideman, Don H. Johnson and C. Sidney Burrus. Gauss and the History of the Fast Fourier Transform. *IEEE ASSP Magazine*, pages 14–21, 1984.

[142] Monika R. Henzinger and Valerie King. Fully Dynamic Biconnectivity and Transitive Closure. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages664–672, 1995.

[143] Monika R. Henzinger and Valerie King. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time Per Operation. *Journal of the ACM*, 46(4):502–516, 1999.

[144] Monika R. Henzinger, Satish Rao and Harold N. Gabow. Computing Vertex Connectivity: New Bounds from Old Techniques. *Journal of Algorithms*, 34(2):222–250, 2000.

[145] Nicholas J. Higham. Exploiting Fast Matrix Multiplication Within the Level 3 BLAS. *ACM Transactions on Mathematical Software*, 16(4):352–368, 1990.

[146] C. A. R. Hoare. Algorithm 63 (PARTITION) and Algorithm 65 (FIND). *Communications of the ACM*, 4(7):321–322, 1961.

[147] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[148] Dorit S. Hochbaum. Efficient Bounds for the Stable Set, Vertex Cover and Set Packing Problems. *Discrete Applied Math*, 6:243–254, 1983.

[149] Dorit S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997.

[150] W. Hoeffding. On the Distribution of the Number of Successes in Independent Trials. *Annals of Mathematical Statistics*, 27:713–721, 1956.

[151] Micha Hofri. *Probabilistic Analysis of Algorithms*. Springer-Verlag, 1987.

[152] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[153] John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.

[154] John E. Hopcroft and Robert E. Tarjan. Efficient Algorithms for Graph Manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[155] John E. Hopcroft and Jeffrey D. Ullman. Set Merging Algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973.

[156] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[157] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.

[158] Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran. *Computer Algorithms*. Computer Science Press, 1998.

[159] T. C. Hu and M. T. Shing. Computation of Matrix Chain Products. Part I. *SIAM Journal on Computing*, 11(2):362–373, 1982.

[160] T. C. Hu and M. T. Shing. Computation of Matrix Chain Products. Part II. *SIAM Journal on Computing*, 13(2):228–251, 1984.

[161] T. C. Hu and A. C. Tucker. Optimal Computer Search Trees and Variable-Length Alphabetic Codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

[162] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[163] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao and Thomas Turnbull. Implementation of Strassen's Algorithm for Matrix Multiplication. In *SC96 Technical Papers*, 1996.

[164] Oscar H. Ibarra and Chul E. Kim. Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. *Journal of the ACM*, 22(4):463–468, 1975.

[165] R. A. Jarvis. On the Identification of the Convex Hull of a Finite Set of Points in the Plane. *Information Processing Letters*, 2:18–21, 1973.

[166] David S. Johnson. Approximation Algorithms for Combinatorial Problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[167] David S. Johnson. The NP-Completeness Column: An Ongoing Guide — The Tale of the Second Prover. *Journal of Algorithms*, 13(3):502–524, 1992.

[168] Donald B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM*, 24(1):1–13, 1977.

[169] David R. Karger, Philip N. Klein and Robert E. Tarjan. A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees. *Journal of the ACM*, 42(2):321–328, 1995.

[170] David R. Karger, Daphne Koller and Steven J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6):1199–1217, 1993.

[171] Howard Karloff. *Linear Programming*. Birkhäuser, 1991.

[172] N. Karmarkar. A New Polynomial-Time Algorithm for Linear Programming. *Combinatorica*, 4(4):373–395, 1984.

[173] Richard M. Karp. Reducibility Among Combinatorial Problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[174] Richard M. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34:165–201, 1991.

[175] Richard M. Karp and Michael O. Rabin. Efficient Randomized Pattern-Matching Algorithms. Technical Report TR-31-81, Aiken Computation Laboratory, Harvard University,1981.

[176] A. V. Karzanov. Determining the Maximal Flow in a Network by the Method of Preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.

[177] Valerie King. A Simpler Minimum Spanning Tree Verification Algorithm. *Algorithmica*, 18(2):263–270, 1997.

[178] Valerie King, Satish Rao and Robert E. Tarjan. A Faster Deterministic Maximum Flow Algorithm. *Journal of Algorithms*, 17:447–474, 1994.

[179] Jeffrey H. Kingston. *Algorithms and Data Structures: Design, Correctness,Analysis*. Addison-Wesley, 1990.

[180] D. G. Kirkpatrick and R. Seidel. The Ultimate Planar Convex Hull Algorithm? *SIAM Journal on Computing*, 15(2):287–299, 1986.

[181] Philip N. Klein and Neal E. Young. Approximation Algorithms for NP-Hard Optimization Problems. In *CRC Handbook on Algorithms*, pages 34-1–34-19. CRC Press, 1999.

[182] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1968. Second edition, 1973.

Д. Кнут. *Искусство программирования, т.1. Основные алгоритмы, 3-е изд.* — М.: “Вильямс”, 2000.

[183] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1969. Second edition, 1981.

Д. Кнут. *Искусство программирования, т.2. Получисленные алгоритмы, 3-е изд.* — М.: “Вильямс”, 2000.

[184] Donald E. Knuth. Optimum Binary Search Trees. *Acta Informatica*, 1:14–25, 1971.

[185] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

Д. Кнут. *Искусство программирования, т.3. Сортировка и поиск, 2-е изд.* — М.: “Вильямс”, 2000.

[186] Donald E. Knuth. Big Omicron and Big Omega and Big Theta. *ACM SIGACT News*, 8(2):18–23, 1976.

[187] Donald E. Knuth, Morris, Jr., James H. and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[188] J. Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.

[189] Bernhard Korte and László Lovász. Mathematical Structures Underlying Greedy Algorithms. In F. Gecseg, editor, *Fundamentals of Computation Theory*, number 117 in Lecture Notes in Computer Science, pages 205–209. Springer-Verlag, 1981.

[190] Bernhard Korte and László Lovász. Structural Properties of Greedoids. *Combinatorica*, 3:359–374, 1983.

[191] Bernhard Korte and László Lovász. Greedoids — A Structural Framework for the Greedy Algorithm. In W. Pulleybank, editor, *Progress in Combinatorial Optimization*, pages 221–243. Academic Press, 1984.

[192] Bernhard Korte and László Lovász. Greedoids and Linear Objective Functions. *SIAM Journal on Algebraic and Discrete Methods*, 5(2):229–238, 1984.

[193] Dexter C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, 1992.

[194] David W. Krumme, George Cybenko and K. N. Venkataraman. Gossiping in Minimal Time. *SIAM Journal on Computing*, 21(1):111–139, 1992.

[195] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[196] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, 1976.

[197] Eugene L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

[198] C. Y. Lee. An Algorithm for Path Connection and Its Applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961.

[199] Tom Leighton and Satish Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 422–431, 1988.

[200] Debra A. Lelewer and Daniel S. Hirschberg. Data Compression. *ACM Computing Surveys*, 19(3):261–296, 1987.

[201] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse and J. M. Pollard. The number field sieve. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of*

the Number Field Sieve, volume 1554 of *Lecture Notes in Mathematics*, pages 11–42, Springer-Verlag, 1993.

[202] H. W. Lenstra, Jr. Factoring Integers with Elliptic Curves. *Annals of Mathematics*, 126:649–673, 1987.

[203] L. A. Levin. Universal Sorting Problems. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973.

[204] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, second edition, 1998.

[205] C. L. Liu. *Introduction to Combinatorial Mathematics*. McGraw-Hill, 1968.

[206] László Lovász. On the Ratio of Optimal Integral and Fractional Covers. *Discrete Mathematics*, 13:383–390, 1975.

[207] László Lovász and M. D. Plummer. *Matching Theory*. Volume 121 of *Annals of Discrete Mathematics*. North Holland, 1986.

[208] Bruce M. Maggs and Serge A. Plotkin. Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters*, 26(6):291–293, 1988.

[209] Michael Main. *Data Structures and Other Objects Using Java*. Addison-Wesley, 1999.

[210] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[211] Conrado Martínez and Salvador Roura. Randomized Binary Search Trees. *Journal of the ACM*, 45(2):288–323, 1998.

[212] William J. Masek and Michael S. Paterson. A Faster Algorithm Computing String Edit Distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[213] H. A. Maurer, Th. Ottmann and H.-W. Six. Implementing Dictionaries Using Binary Trees of Very Small Height. *Information Processing Letters*, 5(1):11–14, 1976.

[214] Ernst W. Mayr, Hans Jürgen Prömel and Angelika Steger. *Lectures on Proof Verification and Approximation Algorithms. Number 1367 in*. of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[215] C. C. McGeoch. All pairs shortest paths and the essential subgraph. *Algorithmica*, 13(5):426–441, 1995.

[216] M. D. McIlroy. A Killer Adversary for Quicksort. *Software — Practice and Experience*, 29(4):341–344, 1999.

[217] Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1984.

[218] Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 of *Data Structures and Algorithms*. Springer-Verlag, 1984.

[219] Kurt Mehlhorn. *Multidimensional Searching and Computational Geometry*, volume 3 of *Data Structures and Algorithms*. Springer-Verlag, 1984.

[220] Alfred J. Menezes, van Oorschot, Paul C. and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[221] Gary L. Miller. Riemann's Hypothesis and Tests for Primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[222] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[223] Joseph S. B. Mitchell. Guillotine Subdivisions Approximate Polygonal Subdivisions: A Simple Polynomial-Time Approximation Scheme for Geometric TSP, k-MST, and Related Problems. *SIAM Journal on Computing*, 28(4):1298–1309, 1999.

[224] Louis Monier. *Algorithmes de Factorisation D'Entiers*. PhD thesis, L'Universite Paris-Sud, 1980.

[225] Louis Monier. Evaluation and Comparison of Two Efficient Probabilistic Primality Testing Algorithms. *Theoretical Computer Science*, 12(1):97–108, 1980.

[226] Edward F. Moore. The Shortest Path Through a Maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[227] Rajeev Motwani, Joseph (Seffi) Naor and Prabakhar Raghavan. Randomized Approximation Algorithms in Combinatorial Optimization. In *Dorit Hochbaum, editor, Approximation Algorithms for NP-Hard Problems*, pages 447–481. PWS Publishing Company, 1997.

[228] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge Unversity Press, 1995.

[229] J. I. Munro and V. Raman. Fast Stable In-Place Sorting with $O(n)$ Data Moves. *Algorithmica*, 16(2):151–160, 1996.

[230] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[231] Ivan Niven and Herbert S. Zuckerman. *An Introduction to the Theory of Numbers.Fourth edition*. John Wiley & Sons, 1980.

[232] Alan V. Oppenheim and Ronald W. Schafer, with John R. Buck. *Discrete-Time Signal Processing.Second edition*. Prentice-Hall, 1998.

[233] Alan V. Oppenheim and Alan S. Willsky, with S. Hamid Nawab. *Signals and Systems*. Prentice-Hall, second edition, 1997.

[234] James B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78(1):109–129, 1997.

[235] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1993.

[236] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[237] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[238] Michael S. Paterson, 1974. Unpublished lecture, Ile de Berder, France.

[239] Michael S. Paterson. Progress in Selection. In *Proceedings of the Fifth Scandinavian Workshop on Algorithm Theory*, pages368–379, 1996.

[240] Pavel A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, 2000.

[241] Steven Phillips and Jeffery Westbrook. Online Load Balancing and Network Flow. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 402–411, 1993.

[242] J. M. Pollard. A Monte Carlo Method for Factorization. *BIT*, 15:331–334, 1975.

[243] J. M. Pollard. Factoring with cubic integers. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 4–10, Springer-Verlag, 1993.

[244] Carl Pomerance. On the Distribution of Pseudoprimes. *Mathematics of Computation*, 37(156):587–593, 1981.

[245] Carl Pomerance, editor. *Proceedings of the AMS Symposia in Applied Mathematics: Computational Number Theory and Cryptography*. American Mathematical Society, 1990.

[246] William K. Pratt. *Digital Image Processing.Second edition*. John Wiley & Sons, 1991.

[247] Franco P. Preparata. An Optimal Real Time Algorithm for Planar Convex Hulls. *Communications of the ACM*, 22:402–405, 1979.

[248] William H. Press, Brian P. Flannery, Saul A. Teukolsky and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1986.

[249] William H. Press, Brian P. Flannery, Saul A. Teukolsky and William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1988.

[250] R. C. Prim. Shortest Connection Networks and Some Generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[251] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, 1990.

[252] Paul W. Purdom, Jr., and Cynthia A. Brown. *The Analysis of Algorithms*. Holt, Rinehart, and Winston, 1985.

[253] Michael O. Rabin. Probabilistic Algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39. Academic Press, 1976.

[254] Michael O. Rabin. Probabilistic Algorithm for Testing Primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[255] P. Raghavan and C. D. Thompson. Randomized Rounding: A Technique for Provably Good Algorithms and Algorithmic Proofs. *Combinatorica*, 7:365–374, 1987.

[256] Rajeev Raman. Recent Results on the Single-Source Shortest Paths Problem. *SIGACT News*, 28(2):81–87, 1997.

[257] Edward M. Reingold, Jürg Nievergelt and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.

[258] Hans Riesel. *Prime Numbers and Computer Methods for Factorization*. of *Progress in Mathematics*. Birkhäuser, 1985.

[259] Ronald L. Rivest, Adi Shamir and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. See also U.S. Patent 4,405,829$.

[260] Herbert Robbins. A Remark on Stirling's Formula. *American Mathematical Monthly*, 62(1):26–29, 1955.

[261] D. J. Rosenkrantz, R. E. Stearns and P. M. Lewis. An Analysis of Several Heuristics for the Traveling Salesman Problem. *SIAM Journal on Computing*, 6:563–581, 1977.

[262] Salvador Roura. An Improved Master Theorem for Divide-and-Conquer Recurrences. In *Proceedings of Automata, Languages and Programming, 24th International Colloquium, ICALP'97*, pages 449–459, 1997.

[263] Y. A. Rozanov. *Probability Theory: A Concise Course*. Dover, 1969.

[264] S. Sahni and T. Gonzalez. P-Complete Approximation Problems. *Journal of the ACM*, 23:555–565, 1976.

[265] A. Schönhage, M. Paterson and N. Pippenger. Finding the median. *Journal of Computer and System Sciences*, 13(2):184–199, 1976.

[266] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[267] Alexander Schrijver. Paths and Flows — a Historical Survey. *CWI Quarterly*, 6:169–183, 1993.

[268] Robert Sedgewick. Implementing Quicksort Programs. *Communications of the ACM*, 21(10):847–857, 1978.

[269] Robert Sedgewick. *Algorithms*. Addison-Wesley, second edition. 1988.

[270] Raimund Seidel. On the All-Pairs-Shortest-Path Problem in Unweighted Undirected Graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

[271] Raimund Seidel and C. R. Aragon. Randomized Search Trees. *Algorithmica*, 16:464–497, 1996.

[272] João Setubal and João Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[273] Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis.Second edition*. Prentice Hall, 2001.

[274] Jeffrey Shallit. Origins of the Analysis of the Euclidean Algorithm. *Historia Mathematica*, 21(4):401–419, 1994.

[275] Michael I. Shamos and Dan Hoey. Geometric Intersection Problems. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pages 208–215, 1976.

[276] M. Sharir. A Strong-Connectivity Algorithm and Its Applications in Data Flow Analysis. *Computers and Mathematics with Applications*, 7:67–72, 1981.

[277] David B. Shmoys. Computing Near-Optimal Solutions to Combinatorial Optimization Problems. In William Cook, László Lovász and Paul Seymour, editors, Combinatorial Optimization, volume 20 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1995.

[278] Avi Shoshan and Uri Zwick. All Pairs Shortest Paths In Undirected Graphs With Integer Weights. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 605–614, 1999.

[279] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[280] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1998.

[281] Daniel D. Sleator and Robert E. Tarjan. A Data Structure for Dynamic Trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[282] Daniel D. Sleator and Robert E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.

[283] Joel Spencer. *Ten Lectures on the Probabilistic Method*. Regional Conference Series on Applied Mathematics (No. 52). SIAM, 1987.

[284] Daniel A. Spielman and Shang-Hua Teng. The Simplex Algorithm Usually Takes a Polynomial Number of Steps. In *Proceedings of the 33th Annual ACM Symposium on Theory of Computing*, 2001.

[285] Gilbert Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986.

[286] Gilbert Strang. *Linear Algebra and Its Applications*.*Third edition*. Harcourt Brace Jovanovich, 1988.

[287] Volker Strassen. Gaussian Elimination Is Not Optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[288] T. G. Szymanski. A Special Case of the Maximal Common Subsequence Problem. Technical Report TR-170, Computer Science Laboratory, Princeton University, 1975.

[289] Robert E. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[290] Robert E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[291] Robert E. Tarjan. A Class of Algorithms Which Require Nonlinear Time to Maintain Disjoint Sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979.

[292] Robert E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[293] Robert E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[294] Robert E. Tarjan. Class notes: Disjoint set union. COS 423, Princeton University, 1999.

[295] Robert E. Tarjan and Jan van Leeuwen. Worst-Case Analysis of Set Union Algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

[296] George B. Thomas, Jr., and Ross L. Finney. *Calculus and Analytic Geometry*.*Seventh edition*. Addison-Wesley, 1988.

[297] Mikkel Thorup. Faster determinstic sorting and priority queues in linear space. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 550–555, 1998.

[298] Mikkel Thorup. Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. *Journal of the ACM*, 46(3):362–394, 1999.

[299] Mikkel Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000.

[300] Richard Tolimieri, Myoung An and Chao Lu. *Mathematics of Multidimensional Fourier Transform Algorithms*. Springer-Verlag, second edition, 1997.

[301] P. van Emde Boas. Preserving Order in a Forest in Less Than Logarithmic Time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84. IEEE Computer Society, 1975.

[302] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier Science Publishers and The MIT Press, 1990.

[303] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992.

[304] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer Academic Publishers, 1996.

[305] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.

[306] Rakesh M. Verma. General Techniques for Analyzing Recursive Algorithms with Applications. *SIAM Journal on Computing*, 26(2):568–581, 1997.

[307] Jean Vuillemin. A Data Structure for Manipulating Priority Queues. *Communications of the ACM*, 21(4):309–315, 1978.

[308] Stephen Warshall. A Theorem on Boolean Matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[309] Michael S. Waterman. *Introduction to Computational Biology, Maps, Sequences and Genomes*. Chapman & Hall, 1995.

[310] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, 1994.

[311] Mark Allen Weiss. *Algorithms, Data Structures and Problem Solving with C++*. Addison-Wesley, 1996.

[312] Mark Allen Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley, 1998.

[313] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison-Wesley, 1999.

[314] Hassler Whitney. On the Abstract Properties of Linear Dependence. *American Journal of Mathematics*, 57:509–533, 1935.

[315] Herbert S. Wilf. *Algorithms and Complexity*. Prentice-Hall, 1986.

[316] J. W. J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7:347–348, 1964.

[317] S. Winograd. On the algebraic complexity of functions. In *Actes du Congrès International des Mathématiciens*, volume 3, pages 283–288, 1970.

[318] Andrew C.-C. Yao. A Lower Bound to Finding Convex Hulls. *Journal of the ACM*, 28(4):780–787, 1981.

[319] Yinyu Ye. *Interior Point Algorithms: Theory and Analysis*. John Wiley & Sons, 1997.

[320] Daniel Zwillinger, editor. *CRC Standard Mathematical Tables and Formulae*. CRC Press, 30th edition, 1996.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Символы

O-обозначения, *91*
Ω-обозначения, *92*
Θ-обозначения, *88*
ω-обозначения, *95*
ϕ-функция Эйлера, *972*
o-обозначения, *94*
2-3-4-дерево, *520*
2-3-4-пирамида, *555*

A

Abelian group, *968*
Additive group modulo n, *969*
Aggregate analysis, *483*
Amortized analysis, *482*
Approximation scheme, *1152*
Articulation point, *641*
AVL-дерево, *359*

B

B*-дерево, *520*
B+-дерево, *520*
B-дерево, *515*; *520*
  Вставка, *524*; *526*
  Высота, *521*
  Минимальная степень, *520*
  Поиск, *522*
  Разбиение заполненного узла, *524*
  Создание, *523*
  Удаление, *530*
Back substitution, *843*
Bernoulli trial, *1243*
Bijection, *1212*
Binary heap, *179*
Binary relation, *1207*
Binary search, *82*
Binary tree, *1221*
Binomial distribution, *1245*
Binomial heap, *541*
Breadth-first search, *613*
Breadth-first tree, *620*
Bubblesort, *83*
Bucket sort, *230*

C

Cartesian product, *1206*
ceil, *98*
Clique problem, *1128*
Component graph, *636*
Conjunctive normal form, *1123*
Constraint graph, *690*
Convex combination, *1048*
Convex hull, *1063*
Counting sort, *224*

D

Decision tree, *221*
Deque, *264*
Diagonal matrix, *825*
Directed graph, *1213*
Discrete Fourier Transform, *933*; *938*
Discrete random variable, *1238*

Disjoint-set forest, *589*
Disjunctive normal form, *1124*
Divisor, *956*
Dynamic graph, *514*
Dynamic tree, *513*

E

Edit distance, *437*
Equivalence class, *957*
Expected value, *1239*

F

Fast Fourier Transform, *938*
Feasible solution, *873*
Fibonacci heap, *559*
FIFO, *260*; *262*
Final state function, *1030*
Finite automaton, *1029*
Finite group, *968*
floor, *98*
Forest, *1217*
Forward substitution, *842*

G

Geometric distribution, *1244*
Greedy algorithm, *442*
Group, *968*

H

Heap, *178*
Heapsort, *178*
Hypergraph, *1217*

I

Identity matrix, *825*
Incidence matrix, *613*
Indicator random variable, *144*
Injection, *1211*
Inorder tree walk, *317*
Input size, *66*
Insertion sort, *58*
Integer linear-programming problem, *1145*
Internal node, *1221*
Inverse matrix, *828*

L

Leaf, *1221*
Least common multiple, *967*
Legendre symbol, *1014*
LIFO, *260*
Linear-programming problem, *687*
Linearity of expectation, *1240*
Linked list, *264*
Longest-simple-cycle problem, *1145*
Loop invariant, *60*
Lower-triangular matrix, *826*
LUP-разложение, *841*

M

Manhattan distance, *253*
Maximal matching, *1157*
Median, *240*
Merge sort, *72*
Modular exponentiation, *985*
Monge array, *137*
Multigraph, *1217*
Multiple, *956*
Multiplicative group modulo n, *970*

N

Normal equation, *863*
NP-полнота, *1087*; *1108*

O

One-to-one correspondence, *1212*
Order of growth, *70*
Order statistic, *240*
Ordered pair, *1206*
Ordered tree, *1221*

P

Pairwise relatively prime, *960*
Permutation matrix, *826*
Permutation network, *820*
Positive-definite matrix, *831*
Postorder tree walk, *318*
Predecessor matrix, *709*
Predecessor subgraph, *620*; *668*; *709*
Prefix code, *460*
Prefix function, *1038*
Preorder tree walk, *318*
Prime, *956*
Priority queue, *190*
Probability density function, *1238*

Probability distribution, *1232*

Q

Quadratic residue, *1014*
Queue, *260*
Quotient, *957*

R

Radix sort, *226*
Radix tree, *333*
Rate of growth, *70*
Recursion tree, *115*
Relatively prime, *960*
Reminder, *957*
Residue, *957*
RSA, *991*

S

Satellite data, *256*
Sentinel, *266*
Set-covering problem, *1164*
Set-partition problem, *1145*
Shortest path, *617*
Singly linked list, *264*
Singular matrix, *829*
Singular value decomposition, *868*
Sorting network, *803*
Spanning tree, *469*; *644*
Splay tree, *513*
Stack, *260*
Standard deviation, *1242*
String, *1017*; *1227*
String-matching problem, *1017*
Subgraph, *1215*
Subset-sum problem, *1140*
Substring, *1227*
Suffix function, *1030*
Surjection, *1211*
Symmetric matrix, *826*

T

Tail recursion, *217*
Toeplitz matrix, *949*
Topological sort, *632*
Transitive closure, *722*
Transposition network, *819*
Traveling-salesman problem, *1138*
Treap, *360*
Triangle inequality, *1158*
Tridiagonal matrix, *825*

U

Undirected graph, *1213*
Uniform probability distribution, *1233*
Upper-triangular matrix, *826*

V

Variable-length code, *459*
Variance, *1242*
Vertex cover, *1131*; *1154*

A

Алгоритм, *46*
- Анализ, *64*
- Асимптотическая эффективность, *87*
- Беллмана-Форда, *672*; *702*
- Верификации, *1102*
- Витерби, *439*
- Время работы, *66*
- Габова, *703*
- Дейкстры, *680*
- Джонсона, *726*
- Евклида, *963*
- Жадный, *442*
- Карпа, *705*
- Кнута-Морриса-Пратта, *1036*
- Корректность, *47*
- Крускала, *651*
- Миллера-Рабина, *999*
- Обход по Джарвису, *1071*
- “Поднять-в-начало”, *780*
- Поиска gcd бинарный, *1013*
- Приближенный, *1151*
- Приведения, *1107*
- Прима, *653*
- Проталкивания предпотока, *762*
- Рабина-Карпа, *1022*
- Рандомизированный, *70*; *143*; *149*
- Симплекс, *875*
- Сканирования по Грэхему, *1065*
- Умножения матриц Штрассена, *833*
- Флойда-Варшалла, *718*
- Форда-Фалкерсона, *742*
- Хаффмана, *462*

Хопкрофта-Карпа, *791*
Штрассена, *833*
Эдмондса-Карпа, *753*
Эффективность, *52*
Асимптотическая, *87*
Алфавит, *1097*
Амортизированная стоимость, *483*; *488*; *491*
Анализ
Алгоритма, *64*
Амортизационный, *482*
Вероятностный, *142*
Групповой, *483*
Метод потенциалов, *491*
Методе бухгалтерского учета, *487*
Арифметическая прогрессия, *1193*
Асимптотическая верхняя граница, *91*
Асимптотическая нижняя граница, *92*
Асимптотически точная оценка, *89*
Асимптотические обозначения, *88*

Б

Базисная функция, *861*
Базисное решение, *894*
Байеса теорема, *1236*
Бернулли испытание, *1243*
Биекция, *1212*
Бинарное дерево поиска, *317*
Вставка, *324*
Оптимальное, *426*
Поиск, *320*
Поиск минимума и максимума, *321*
Предшествующий и последующий узлы, *321*
Свойство, *317*
Случайное, *328*
Удаление, *325*
Бинарное отношение, *1207*
Бинарный алгоритм gcd, *1013*
Бинарный поиск, *82*
Бином Ньютона, *1229*
Биномиальная пирамида, *541*
Вставка, *550*
Поиск минимального ключа, *544*
Свойства, *541*
Слияние, *545*
Создание, *544*
Список корней, *543*
Удаление, *554*
Уменьшение ключа, *552*
Биномиальное дерево, *539*
Неупорядоченное, *562*
Биномиальные коэффициенты, *1229*
Битонический сортировщик, *810*
Бленда правило, *906*
Буля неравенство, *1237*
Быстрая сортировка, *198*
Анализ, *209*
Глубина стека, *217*
Метод тройной медианы, *218*
Опорный элемент, *200*
Производительность, *203*
Разбиение, *199*
Разбиение по Хоару, *214*
Рандомизированная, *208*
Быстрое преобразование Фурье, *938*

В

Вандермонда матрица, *930*
Вектор, *824*; *1048*
Аннулирующий, *830*
Единичный, *824*
Линейная зависимость, *829*
Норма, *828*
Ортонормальность, *868*
Векторное произведение, *1049*
Вероятностный анализ, *70*; *142*
Вероятность, *1232*
Аксиомы, *1232*
Распределение, *1232*
Биномиальное, *1245*
Геометрическое, *1244*
Дискретное, *1233*
Непрерывное равномерное, *1234*
Равномерное, *1233*
Условная, *1235*
Вершинное покрытие, *1131*
Взаимно однозначное соответствие, *1212*
Взаимно простые числа, *960*
Взвешенная медиана, *253*
Возведение в степень по модулю, *985*
Выметание, *1055*
Выпуклая комбинация, *1048*

Выпуклая оболочка, *1063*
Выпуклое множество, *742*
Вычислительная задача, *46*

Г

Гамильтонов цикл, *1101*
Генератор случайных чисел, *143*
Геометрическая прогрессия, *1193*
Гиперграф, *1217*
Глубина стека, *217*
Горнера правило, *84*
Горнера схема, *929*
Граф, *1213*
  ε-плотный, *732*
  Ациклический, *1215*
  Вершина, *1213*
    Степень, *1214*
  Вершинное покрытие, *1154*
  Взвешенный, *611*
  Двудольный, *1216*
  Динамический, *514*
  Изоморфность, *1215*
  Квадрат, *612*
  Компонентов, *636*
  Кратчайший путь, *617*
  Матрица инциденций, *613*
  Матрица смежности, *611*
  Множество вершин, *1213*
  Множество ребер, *1213*
  Мост, *641*
  Независимое множество, *1145*
  Неориентированный, *1213*
  Ограничений, *690*
  Односвязность, *632*
  Ориентированный, *1213*
  Остовное дерево, *469*; *644*
  Паросочетание, *757*
  Петля, *1213*
  Плотный, *610*
  Подпуть, *1214*
  Поиск в глубину, *622*
  Поиск в ширину, *613*
  Полный, *1216*
  Полусвязный, *640*
  Представление, *609*
  Путь, *1214*
    Простой, *1214*
  Разреженный, *610*
  Разрез, *647*
  Раскраска, *1147*; *1224*
  Ребро, *1213*
    Инцидентное, *1214*
  Связные компоненты, *1215*
  Связный, *1215*
  Сжатие, *1217*
  Сильно связный, *1215*
  Смежные вершины, *1214*
  Список смежности, *610*
  Точка сочленения, *641*
  Транзитивное замыкание, *722*; *731*
  Транспонирование, *612*
  Узкое остовное дерево, *660*
  Цикл, *1214*
  Эйлеров цикл, *642*
Группа, *968*
  Абелева, *968*
  Аддитивная по модулю, *969*
  Генератор, *983*
  Конечная, *968*
  Мультипликативная по модулю, *970*
  Первообразный корень, *983*
  Порядок элемента, *974*
  Циклическая, *983*
Групповой анализ, *483*

Д

Дважды связанный список, *264*
Двоичный поиск, *82*
Двойное хеширование, *303*
Дек, *264*
Декартова сумма, *935*
Декартово произведение, *1206*
Делитель, *956*
  Наибольший общий, *958*
  Тривиальный, *956*
Дерамида, *360*
Дерево
  2-3-4-дерево, *520*
  AVL, *359*
  B*-дерево, *520*
  B+-дерево, *520*
  B-дерево, *515*
  Без выделенного корня, *1218*

Бинарное, *274*; *1221*
Бинарное поиска, *317*
Биномиальное, *539*
Биномиальное неупорядоченное, *562*
Внутренний узел, *1221*
Высота, *1221*
Диаметр, *622*
Динамическое, *513*
Красно-черное, *336*
Кратчайших путей, *669*; *697*
Лист, *1221*
Остовное, *644*
Остовное графа, *469*
Отрезков, *375*
Позиционное, *1222*
Поиска в ширину, *620*
Полностью бинарное, *1222*
Порядковой статистики, *366*
Пустое, *1222*
Расширяющееся, *513*
Рекурсии, *115*
Решений, *221*
С корнем, *1220*
Свободное, *1218*
Степень, *1221*
Узел, *1220*
Упорядоченное, *1221*
Цифровое, *333*
Деревьев рекурсии метод, *115*
Диаграмма PERT, *679*
Диаграмма Венна, *1204*
Дизъюнктивная нормальная форма, *1124*
Динамическая таблица, *495*
Динамический граф, *514*
Динамическое дерево, *513*
Динамическое программирование, *386*
Оптимальная подструктура, *390*
Перекрывающиеся вспомогательные задачи, *411*
Дисковый накопитель, *516*
Дискретная случайная величина, *1238*
Дискретное преобразование Фурье, *933*; *938*
Дискретный логарифм, *984*
Дисперсия, *1242*
Дополнение, *1205*
Дополнение Шура, *846*

Е

Евклида алгоритм, *963*

Ж

Жадный алгоритм, *442*
Оптимальная подструктура, *455*
Свойство жадного выбора, *454*

З

Задача
Абстрактная, *1092*
Выбора, *240*
Выполнимости схемы, *1112*
Иосифа, *381*
Класс сложности, *1094*
Конкретная, *1094*
Линейного программирования, *687*; *869*
Максимизации, *872*
Минимизации, *872*
О вершинном покрытии, *1131*; *1154*
О выборе процессов, *443*
О выходе, *786*
О гамильтоновом цикле, *1102*; *1133*
О гардеробщике, *148*
О клике, *1128*
О коммивояжере, *1138*; *1157*
О коммивояжере битоническая, *434*
О кратчайшем пути, *663*
О кратчайшем пути в заданный пункт назначения, *664*
О кратчайшем пути из одной вершины, *664*
О кратчайшем пути между всеми парами вершин, *665*
О кратчайшем пути между заданной парой вершин, *665*
О максимальном потоке, *736*
О минимальном остовном дереве, *470*
О минимальном покрытии путями, *787*
О найме, *140*
О независимом множестве, *1145*
О перемножении последовательности матриц, *397*

О покрытии множества, *1164*
О разделении множества, *1145*
О разрешимости системы линейных неравенств, *922*
О раскрашивании графа, *1147*
О рюкзаке, *456*
О самой длинной общей подпоследовательности, *418*
О самом длинном простом цикле, *1145*
О сумме подмножества, *1140*
Об изоморфизме подграфу, *1144*
Оптимизации, *386*; *1088*
Планирования единичных заданий, *475*
Поиска, *63*
Поиска максимального потока, *887*
Поиска минимального остовного дерева, *556*; *644*
Поиска подстроки, *1017*
Поиска потока с минимальными затратами, *889*
Поиска сильно связных компонентов графа, *636*
Принятия решения, *1088*
Проверки остовного дерева, *662*
Раскрашивания интервального графа, *453*
Расписание конвейера, *387*
Сортировки, *46*; *57*; *174*
Существования решения, *688*
Целочисленного линейного программирования, *877*; *923*; *1145*
Экземпляр, *47*
Законы де Моргана, *1124*; *1204*
Замыкание, *1097*
Клини, *1097*
Запись, *174*
Золотое сечение, *104*

И

Инвариант цикла, *60*
Инверсия, *85*; *149*
Индикаторная случайная величина, *144*; *156*
Интервал, *375*
Интерполяция, *930*
Инъекция, *1211*
Иосифа задача, *381*
Испытание Бернулли, *1243*
Истинное подмножество, *1203*
Исток, *664*

Й

Йенсена неравенство, *1241*

К

Кармайкла числа, *998*
Карманная сортировка, *230*
Каталана числа, *335*; *398*
Квадратичный вычет, *1014*
Квантиль, *251*
Китайская теорема об остатках, *979*
Класс сложности, *1098*
NP, *1103*
Класс эквивалентности, *1208*
Класс эквивалентности по модулю, *957*
Клика, *1128*
Клини замыкание, *1097*
Ключ, *174*
Код
Бинарный, *459*
Переменной длины, *459*
Префиксный, *460*
Фиксированной длины, *459*
Хаффмана, *459*; *462*
Коллизия, *286*
Коллинеарность, *1049*
Компаратор, *800*
Конечная группа, *968*
Конечный автомат, *1029*
Входной алфавит, *1029*
Поиска подстрок, *1030*
Состояние, *1029*
Функция конечного состояния, *1030*
Функция переходов, *1029*
Конкатенация, *1019*
Конъюнктивная нормальная форма, *1123*
Красно-черное дерево, *336*
Вставка, *342*
Объединение, *358*
Ослабленное, *340*

Поворот, *340*
Свойства, *337*
Удаление, *351*
Черная высота узла, *337*
Кратное, *956*
Кратчайший путь, *664*
Неравенство треугольника, *694*
Свойства ослаблений, *695*
Крафта неравенство, *1224*
Криптографическая система RSA, *991*
Криптографические системы с открытым ключом, *988*

Л

Лагранжа теорема, *973*
Лагранжа формула, *931*
Лежандра символ, *1014*
Лексикографическое сравнение, *332*
Лемма Фаркаша, *924*
Лес, *1217*
Лес непересекающихся множеств, *589*
Линейная функция, *872*
Линейное неравенство, *872*
Линейное ограничение, *872*
Линейное программирование, *687*; *872*
Базисные переменные, *883*
Вспомогательная задача, *915*
Вспомогательная переменная, *882*
Двойственность, *908*
Допустимая область, *873*
Допустимое решение, *688*; *873*; *878*
Дуальность, *908*
Каноническая форма задачи, *883*
Оптимальное решение, *879*
Симплекс, *875*
Стандартная форма задачи, *878*
Целевая функция, *873*
Целевое значение, *873*; *878*
Линейное равенство, *872*
Линейность математического ожидания, *1240*
Линейный поиск, *63*
Логический вентиль, *1110*

М

Максимальное паросочетание, *1157*; *1184*
Манхэттенское расстояние, *253*
Маркова неравенство, *1243*
Массив Монжа, *137*
Математическая индукция, *1196*
Математическое ожидание, *70*; *1239*
Линейность, *1240*
Матрица, *824*
LU-разложение, *845*
LUP-разложение, *841*
Алгебраическое дополнение элемента, *830*
Аннулирующий вектор, *830*
Вандермонда, *832*; *930*
Верхне-треугольная, *826*
Вырожденная, *829*
Вычитание, *827*
Детерминант, *830*
Диагональная, *825*
Дополнение Шура, *846*; *860*
Единичная, *825*
Инциденций, *613*
Квадратная, *825*
Минор, *830*
Нижне-треугольная, *826*
Нулевая, *825*
Обратная, *828*
Обращение, *853*
Определитель, *830*
Перестановки, *826*
Положительно определенная, *831*
Предшествования, *709*
Произведение, *396*
Псевдообратная, *863*
Ранг, *829*
Полный, *829*
Симметричная, *826*
Симметричная положительно определенная, *858*
Сингулярное разложение, *868*
Скалярное произведение, *827*
Сложение, *827*
Смежности, *611*
Совместимость, *396*; *827*
Сопряженно-транспонированная, *858*
Теплица, *949*
Транспонированная, *611*; *824*
Трехдиагональная, *825*; *865*

Умножение, *827*
Эрмитова, *858*
Матрица инцидентности, *480*
Матроид, *467*
Графовый, *468*
Матричный, *468*
Оптимальное подмножество, *470*
Медиана, *240*
Взвешенная, *253*
Метод
Деревьев рекурсии, *109*; *115*
Исключения Гаусса, *845*
Выбор ведущего элемента, *847*
Наименьших квадратов, *861*
Основной, *121*
Подстановки, *111*
Подстановок, *109*
Полларда, *1007*
Форда-Фалкерсона, *742*
Миллера-Рабина проверка простоты, *999*
Многоугольник, *1054*
Многочлен, *926*
Множество, *256*; *1202*
Бесконечное, *1205*
Динамическое, *256*
Дополнение, *1205*
Конечное, *1205*
Мощность, *1205*
Непересекающиеся множества, *1205*
Несчетное, *1205*
Объединение, *1203*
Пересечение, *1203*
Перестановка, *1212*
Перманентное, *357*
Полностью упорядоченное, *256*
Пустое, *1203*
Разбиение, *1205*
Разность, *1203*
Симметрическая разность, *791*
Счетное, *1205*
Частично упорядоченное, *1209*
Модульная арифметика, *98*
Модульное возведение в степень, *985*
Монжа массив, *137*
Монотонность функций, *98*
Моргана законы, *1124*; *1204*
Мультиграф, *1217*
Мультимножество, *1202*
Мультипликативное обратное, *978*

Н

Наибольший общий делитель, *958*; *962*
Наименьшее общее кратное, *967*
Наименьший общий предок, *604*
Наихудший случай, *69*
Непересекающиеся множества, *582*
Лес, *589*
Объединение, *586*
Представление, *585*
Неравенство Буля, *1237*
Неравенство Йенсена, *1241*
Неравенство Крафта, *1224*
Неравенство Маркова, *1243*
Неравенство треугольника, *1158*
Неупорядоченное биномиальное дерево, *562*
Нечеткая сортировка, *218*
Нормальное уравнение, *863*
Нуль-единичный принцип, *805*
Ньютона бином, *1229*

О

Обход дерева, *317*
В обратном порядке, *318*
В прямом порядке, *318*
Центрированный, *317*
Объединяемая пирамида, *278*
Ограничитель, *266*
Однократно связанный список, *264*
Оконечная рекурсия, *217*
Округление чисел, *98*
Оптимальная подструктура, *405*
Оптимальное бинарное дерево поиска, *426*
Основной метод, *121*
Остаток, *957*
Остаток от деления, *98*
Остовное дерево, *469*; *644*
Отношение полного порядка, *1209*
Отношение частичного порядка, *1209*
Отношение эквивалентности, *1208*
Отрезков дерево, *375*
Отрезок, *375*
Отрезок прямой, *1048*

Очередь, *260*; *262*
Голова, *262*
С двусторонним доступом, *264*
С приоритетами, *190*
Невозрастающая, *190*
Неубывающая, *190*
Хвост, *262*

П

Парадокс дней рождения, *157*
Паросочетание, *757*
Максимальной мощности, *1184*
Полное, *761*
Паскаля треугольник, *1231*
Перестановка, *1212*; *1227*
Перестановка Иосифа, *382*
Перестановочная сеть, *820*
Пирамида, *179*
d-арная, *195*
2-3-4-пирамида, *555*
Биномиальная, *541*
Высота узла, *180*
Невозрастающая, *180*
Неубывающая, *180*
Объединяемая, *278*
Свойства, *180*
Фибоначчиева, *559*
Пирамидальная сортировка, *178*; *187*
Пирамиды
Сливаемые, *537*
Подграф, *1215*
Подграф предшествования, *620*; *668*; *709*
Подгруппа, *972*
Генератор, *974*
Истинная, *973*
Подмножество, *1203*
Подстановки метод, *111*
Подстрока, *1227*
Поиск в глубину, *622*
Поиск в ширину, *613*
Покрывающее дерево, *644*
Покрытие путями, *787*
Полином, *99*; *926*
Граница степени, *927*
Коэффициенты, *927*
Представление, основанное на значениях в точках, *929*
Представление, основанное на коэффициентах, *929*
Сложение, *927*; *931*
Степень, *927*
Сумма, *927*
Схема Горнера, *929*
Умножение, *927*
Полларда метод разложения, *1007*
Полуинтервал, *375*
Полярный угол, *1053*
Попарно взаимно простые числа, *960*
Порядковая статистика, *240*
Порядковой статистики дерево, *366*
Порядок роста, *70*
Потенциальная функция, *491*
Правило Бленда, *906*
Правило Горнера, *84*
Правило произведения, *1227*
Правило суммы, *1227*
Преобразование
Бабочки, *944*
Фурье
Быстрое, *938*
Быстрое многомерное, *950*
Дискретное, *933*; *938*
Чирп, *943*
Префиксная функция, *1038*
Приведение задач, *1089*
Произведение
Векторное, *1049*
Производящая функция, *135*
Промежуток, *375*
Пропускная способность, *735*
Простое число, *956*
Псевдокод, *58*
ALLOCATE_OBJECT, *272*
ANY_SEGMENTS_INTERSECT, *1058*
APPROX_TSP_TOUR, *1159*
APPROX_VERTEX_COVER, *1154*
BFS, *614*
B_TREE_CREATE, *524*
B_TREE_INSERT_NONFULL, *527*
B_TREE_INSERT, *526*
B_TREE_SEARCH, *523*
B_TREE_SPLIT_CHILD, *525*

BELLMAN_FORD, *673*
BINOMIAL_HEAP_DECREASE_KEY, *552*
BINOMIAL_HEAP_EXTRACT_MIN, *551*
BINOMIAL_HEAP_INSERT, *550*
BINOMIAL_HEAP_MINIMUM, *544*
BINOMIAL_HEAP_UNION, *545*
BINOMIAL_LINK, *545*
BIT_REVERSE_COPY, *946*
BUBBLESORT, *83*
BUCKET_SORT, *231*
BUILD_MAX_HEAP, *184*
BUILD_MIN_HEAP, *187*
COMPUTE_PREFIX_FUNCTION, *1039*
COMPUTE_TRANSITION_FUNCTION, *1035*
CONSOLIDATE, *568*
COUNTING_SORT, *224*
DFS, *624*
DAG_SHORTEST_PATHS, *677*
DEQUEUE, *262*
DIJKSTRA, *680*
ENQUEUE, *262*
EUCLID, *963*
EXACT_SUBSET_SUM, *1177*
EXTENDED_EUCLID, *966*
FASTEST_WAY, *393*
FIB_HEAP_DECREASE_KEY, *571*
FIB_HEAP_DELETE, *575*
FIB_HEAP_EXTRACT_MIN, *565*
FIB_HEAP_INSERT, *563*
FIB_HEAP_UNION, *564*
FINITE_AUTOMATON_MATCHER, *1032*
FLOYD_WARSHALL, *720*
FREE_OBJECT, *272*
GRAHAM_SCAN, *1065*
GREEDY_SET_COVER, *1166*
HASH_INSERT, *300*
HASH_SEARCH, *301*
HEAP_EXTRACT_MAX, *191*
HEAP_INCREASE_KEY, *192*
HEAP_MAXIMUM, *191*
HEAPSORT, *188*
HOPCROFT_KARP, *791*
INITIALIZE_SIMPLEX, *916*
INORDER_TREE_WALK, *318*
INSERTION_SORT, *59*
INTERVAL_SEARCH, *377*
ITERATIVE_FFT, *946*
KMP_MATCHER, *1038*
LUP_DECOMPOSITION, *850*
LUP_SOLVE, *843*
LU_DECOMPOSITION, *847*
LIST_DELETE, *266*
LIST_INSERT, *265*; *267*
LIST_SEARCH, *265*; *267*
MST_PRIM, *654*
MATRIX_CHAIN_ORDER, *401*
MATRIX_MULTIPLY, *396*
MAX_HEAP_INSERT, *192*
MAX_HEAPIFY, *182*
MERGE_SORT, *77*
MERGE, *73*
MILLER_RABIN, *1001*
MINIMUM, *241*
MODULAR_EXPONENTIATION, *986*
MODULAR_LINEAR_EQUATION_SOLVER, *977*
NAIVE_STRING_MATCHER, *1020*
OS_RANK, *368*
OS_SELECT, *367*
PARTITION, *199*
PERMUTE_BY_SORTING, *151*
PIVOT, *897*
POLLARD_RHO, *1007*
POP, *261*
PSEUDOPRIME, *997*
PUSH, *261*
QUICKSORT, *199*
RB_DELETE, *351*
RB_INSERT_FIXUP, *343*; *352*
RB_INSERT, *342*
RABIN_KARP_MATCHER, *1026*
RADIX_SORT, *228*
RANDOMIZED_PARTITION, *208*
RANDOMIZED_QUICKSORT, *208*
RANDOMIZED_SELECT, *243*
RECURSIVE_FFT, *939*
RELABEL_TO_FRONT, *780*
SEGMENTS_INTERSECT, *1051*
SIMPLEX, *899*
STACK_EMPTY, *261*
STRONGLY_CONNECTED_COMPONENTS, *636*

TOPOLOGICAL_SORT, *633*
TREE_DELETE, *325*
TREE_INSERT, *324*
TREE_MAXIMUM, *321*
TREE_MINIMUM, *321*
TREE_SEARCH, *320*
TREE_SUCCESSOR, *322*
WITNESS, *999*
Псевдопростое число, *997*
Пузырьковая сортировка, *83*

P

Разложение на множители, *1006*
Размер входных данных, *66*
Размещение, *1228*
Разрез, *747*
Рандомизированный алгоритм, *143*; *149*
Распределение вероятностей, *1232*
Расстояние редактирования, *435*; *437*
Расширяющееся дерево, *513*
Реверс битов, *946*
Рекуррентное соотношение
Метод деревьев, *115*
Метод подстановки, *111*
Основной метод, *121*
Рекуррентное уравнение, *78*; *109*
Рекурсия, *72*
Оконечная, *217*
Решений дерево, *221*
Ряд, *1192*
Абсолютно сходящийся, *1192*
Гармонический, *1194*
Линейность, *1192*
Приближение интегралами, *1199*
Расходящийся, *1192*
Сходящийся, *1192*

C

Свертка, *929*
Связанный список, *264*
Символ Лежандра, *1014*
Симплекс-алгоритм, *875*
Система линейных уравнений, *839*
Недоопределенная, *841*
Обратная подстановка, *843*
Переопределенная, *841*
Прямая подстановка, *842*
Решение, *840*
Трехдиагональная, *865*
Система разностных ограничений, *688*
Скалярное произведение, *828*
Скобочная структура, *626*
Сливаемые пирамиды, *537*
Словарь, *256*
Случайная перестановка
С использованием обменов, *153*
С использованием приоритетов, *151*
С равномерным распределением, *152*
Сопутствующие данные, *256*
Сортировка, *46*
Быстрая, *198*
Вставкой, *58*; *83*
Выбором, *71*
Карманная, *230*
Нечеткая, *218*
Пирамидальная, *178*; *187*
Подсчетом, *224*
Поразрядная, *226*
Пузырьковая, *83*
Слиянием, *72*
Сравнением, *220*
Топологическая, *632*
Устойчивая, *226*
Сортировки задача, *174*
Сортирующая сеть, *803*
Глубина провода, *802*
Нечетно-четная, *819*
Провод, *800*
Сортирующие сети, *800*
Составное число, *957*
Сочетание, *1228*
Список
Дважды связанный, *264*
Кольцевой, *264*
Однократно связанный, *264*
Связанный, *264*
Список смежности, *610*
Сплайн, *866*
Сравнивающие сети, *799*
Среднее значение, *1239*
Стандартное отклонение, *1242*
Стек, *217*; *260*
Глубина, *217*

Стирлинга формула, *102*
Строка, *1017*; *1227*
  Префикс, *1019*
  Пустая, *1019*
  Суффикс, *1019*
Структура данных, *50*
Суффиксная функция, *1030*
Схема аппроксимации, *1152*
Схема Горнера, *929*
Сюръекция, *1211*

Т

Таблица истинности, *1111*
Таблица Юнга, *195*
Тензорное произведение, *828*
Теорема
  Байеса, *1236*
  Лагранжа, *973*
  Ферма, *983*
  Халла, *761*
  Эйлера, *983*
Теплица матрица, *949*
Топологическая сортировка, *632*
Транзитивное замыкание, *722*
Транспозиционная сеть, *819*
Транспортная сеть, *735*
Треугольник Паскаля, *1231*
Тривиальный делитель, *956*

У

Универсальное хеширование, *294*
Универсум, *1204*
Упорядоченная пара, *1206*

Ф

Факториал, *102*
Фаркаша лемма, *924*
Ферма теорема, *983*
Фибоначчи числа, *103*; *135*; *576*; *964*; *1013*
Фибоначчиева пирамида, *559*
  Вставка, *563*
  Извлечение минимального узла, *565*
  Объединение, *564*
  Поиск минимального узла, *564*
  Потенциал, *561*
  Создание, *563*
  Список корней, *561*
  Структура, *560*
  Удаление, *575*
  Уменьшение ключа, *571*
  Уплотнение, *566*
Формула Лагранжа, *931*
Формула Стирлинга, *102*
Функция
  Аргумент, *1210*
  Вложение, *1211*
  Выпуклая вниз, *1241*
  Значение, *1210*
  Квадратичная, *69*
  Линейная, *68*; *872*
  Множество значений, *1211*
  Монотонная, *98*
  Наложение, *1211*
  Область значений, *1210*
  Область определения, *1210*
  Обратная, *1212*
  Плотности вероятности, *1238*
  Полилогарифмически ограниченная, *101*
  Полиномиально ограниченная, *99*
  Префиксная, *1038*
  Приведения, *1107*
  Производящая, *135*
  Распределения вероятности, *234*
  Распределения простых чисел, *996*
  Суффиксная, *1030*
  Целевая, *687*
Фурье преобразование, *933*
  Быстрое, *938*
  Дискретное, *938*

Х

Халла теорема, *761*
Хаффмана код, *459*
Хеш-функция, *285*; *291*
  Метод деления, *292*
  Метод умножения, *293*
Хеширование, *282*
  Вторичная кластеризация, *303*
  Двойное, *303*
  Идеальное, *308*
  Квадратичное исследование, *303*
  Коллизия, *286*

Коэффициент заполнения таблицы, *288*
Линейное исследование, *302*
Открытая адресация, *300*
Первичная кластеризация, *302*
Последовательность исследований, *300*
Простое равномерное, *288*
Прямая адресация, *283*
Равномерное, *302*
С использованием цепочек, *286*
Универсальное, *294*
Хеш-функция, *291*

Ц

Целевая функция, *687*; *873*
Цифровая подпись, *990*
Цифровое дерево, *333*

Ч

Частное, *957*
Чирп-преобразование, *943*
Числа
Взаимно простые числа, *960*
Гармонические, *1194*
Кармайкла, *998*
Каталана, *335*; *398*
Попарно взаимно простые числа, *960*
Псевдопростые, *997*
Фибоначчи, *103*; *135*; *576*; *964*; *1013*
Численная устойчивость, *823*

Э

Эйлера ϕ-функция, *972*
Эйлера теорема, *983*
Эйлеров цикл, *642*
Экземпляр задачи, *47*

Ю

Юнга таблица, *195*

Я

Язык, *1097*

Научно-популярное издание

Томас Х. Кормен, Чарльз И. Лейзерсон,
Рональд Л. Ривест, Клиффорд Штайн

Алгоритмы: построение и анализ, 2-е издание

Литературный редактор *С.Г. Татаренко*
Верстка *А.Н. Полинчик*
Художественный редактор *С.А. Чернокозинский*
Корректоры *Л.А. Гордиенко, Т.А. Корзун*

ООО "И.Д. Вильямс"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 05.11.2012. Формат 70×100/16
Гарнитура Times. Печать офсетная
Усл. печ. л. 104,5. Уч.-изд. л. 74,1
Тираж 1000 экз. Заказ № 0000

Первая Академическая типография "Наука"
199034, С.-Петербург, 9-я линия, 12/28

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ: СТРУКТУРЫ И СТРАТЕГИИ РЕШЕНИЙ СЛОЖНЫХ ПРОБЛЕМ, 4-Е ИЗДАНИЕ

Джордж Люгер

www.williamspublishing.com

в продаже

Данная книга посвящена одной из наиболее перспективных и привлекательных областей развития научного знания — методологии искусственного интеллекта (ИИ). В ней детально описываются как теоретические основы искусственного интеллекта, так и примеры построения конкретных прикладных систем. Книга дает полное представление о современном состоянии развития этой области науки. Подробно рассматриваются вопросы представления знаний при решении задач ИИ, логика решения этих задач, алгоритмы поиска, продукционные системы и машинное обучение. Эти вопросы остаются центральными в области искусственного интеллекта. В книге также представлены результаты новейших исследований, связанных с вопросами понимания естественного языка, обучения с подкреплением, рассуждения в условиях неопределенности, эмерджентных вычислений, автоматического доказательства теорем и решения задач ИИ на основе моделей. Большое внимание уделяется описанию реальных прикладных систем, построенных на принципах ИИ, и современных областей приложения этой области знаний. Помимо математических основ искусственного интеллекта в книге затронуты его философские аспекты. В последней части книги рассматриваются технологии программирования задач из области искусственного интеллекта на языках LISP и PROLOG. Книга будет полезна как опытным специалистам в области искусственного интеллекта, так и студентам и начинающим ученым.

ИСКУССТВО ПРОГРАММИРОВАНИЯ

ТОМ 1. ОСНОВНЫЕ АЛГОРИТМЫ, *3-Е ИЗДАНИЕ*

ТОМ 2. ПОЛУЧИСЛЕННЫЕ МЕТОДЫ, *3-Е ИЗДАНИЕ*

ТОМ 3. СОРТИРОВКА И ПОИСК, *2-Е ИЗДАНИЕ*

Дональд Э. Кнут

в продаже

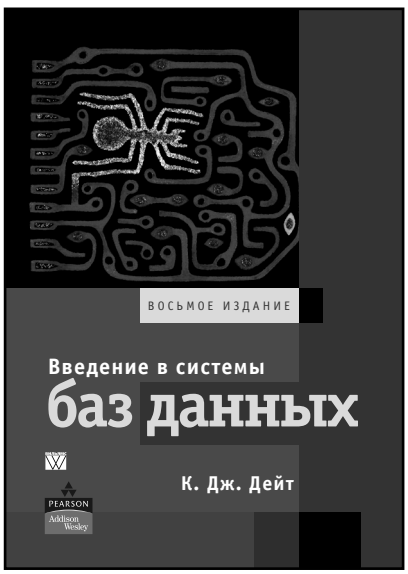
www.williamspublishing.com

На мировом рынке компьютерной литературы существует множество книг, предназначенных для обучения основным алгоритмам и используемых при программировании. Их довольно много, и они в значительной степени конкурируют между собой. Однако среди них есть особая книга.

Это трехтомник *"Искусство программирования"* Д. Э. Кнута, который стоит вне всякой конкуренции, входит в золотой фонд мировой литературы по информатике и является настольной книгой практически для всех, кто связан с программированием. Ценность книги в том, что она предназначена не столько для обучения технике программирования, сколько для обучения, если это возможно, "искусству" программирования, предлагает массу рецептов усовершенствования программ и, что самое главное, учит самостоятельно находить эти рецепты.

ВВЕДЕНИЕ В СИСТЕМЫ БАЗ ДАННЫХ
Восьмое издание

К. Дж. Дейт



www.williamspublishing.com

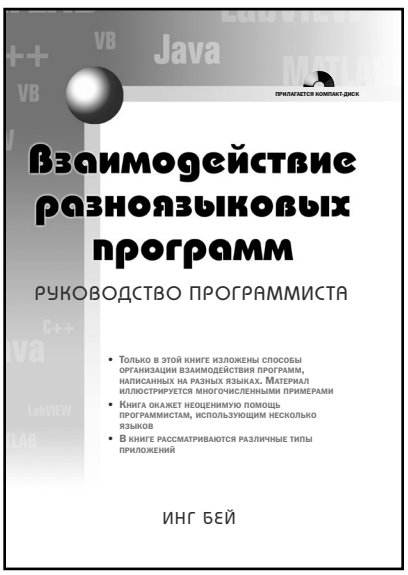
Новое издание фундаментального труда Криса Дейта представляет собой исчерпывающее введение в очень обширную в настоящее время теорию систем баз данных. С помощью этой книги читатель сможет приобрести фундаментальные знания в области технологий баз данных, а также ознакомиться с направлениями, по которым рассматриваемая сфера деятельности, вероятно, будет развиваться в будущем. Книга предназначена для использования в основном в качестве учебника, а не справочника, поэтому, несомненно, вызовет интерес у программистов-профессионалов, научных работников и студентов, изучающих соответствующие курсы в высших учебных заведениях. В ней сделан акцент на усвоении сути и глубоком понимании излагаемого материала, а не просто на его формальном изложении. Книга, безусловно, будет полезна всем, кому приходится работать с базами данных или просто пользоваться ими.

ISBN 978-5-8459-0788-2

в продаже

ВЗАИМОДЕЙСТВИЕ РАЗНОЯЗЫКОВЫХ ПРОГРАММ
Руководство программиста

Инг Бей



www.williamspublishing.com

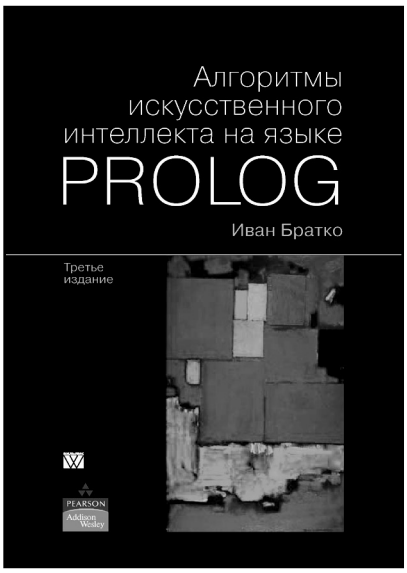
Данная книга посвящена созданию разноязыковых приложений. В частности, в ней рассматриваются вопросы вызова процедуры, написанной на некотором языке, из другой языковой среды. Читатель найдет сведения об использовании ANSI C, Visual C++, Visual Basic, Matlab, Smalltalk, LabView, Java и Perl, а также об организации совместной работы компонентов, написанных на этих языках, с другими частями приложения. Книга в основном ориентирована на разработчиков прикладных и системных программ, научных работников, специалистов по созданию пользовательских интерфейсов и студентов соответствующих специальностей. Она также будет полезна каждому, кто интересуется вопросами создания программного обеспечения и повышения его производительности.

ISBN 5-8459-0861-2

в продаже

АЛГОРИТМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА НА ЯЗЫКЕ PROLOG
3-Е ИЗДАНИЕ

Иван Братко



www.williamspublishing.com

Книга известного специалиста по программированию, в которой приведены основные сведения о языке Prolog, описан процесс разработки программ на Prolog и показано применение языка Prolog во многих областях искусственного интеллекта, включая решение задач и эвристический поиск, программирование в ограничениях, представление знаний и экспертные системы, планирование, машинное обучение, качественные рассуждения, обработка текста на различных языках и ведение игр. Книга содержит описание методов обработки многих важных структур данных, в том числе деревьев и графов. Задачи искусственного интеллекта представлены и разработаны до такой степени детализации, которая позволяет успешно реализовать их на языке Prolog и получить законченные программы.

ISBN 5-8459-0664-4

в продаже