The Algorithm Design Manual

Second Edition

Steven S. Skiena

The Algorithm Design Manual

Second Edition

Springer

Стивен С. Скиена

Алгоритмы
Руководство по разработке

2-е издание

Санкт-Петербург
«БХВ-Петербург»
2011

УДК 681.3.06
ББК 32.973.26-018.2
С42

Скиена С.

С42 Алгоритмы. Руководство по разработке. — 2-е изд.: Пер. с англ. — СПб.: БХВ-Петербург, 2011. — 720 с.: ил.

ISBN 978-5-9775-0560-4

Книга является наиболее полным руководством по разработке эффективных алгоритмов. Первая часть книги содержит практические рекомендации по разработке алгоритмов: приводятся основные понятия, дается анализ алгоритмов, рассматриваются типы структур данных, основные алгоритмы сортировки, операции обхода графов и алгоритмы для работы со взвешенными графами, примеры использования комбинаторного поиска, эвристических методов и динамического программирования. Вторая часть книги содержит обширный список литературы и каталог из 75 наиболее распространенных алгоритмических задач, для которых перечислены существующие программные реализации. Приведены многочисленные примеры задач.

Книгу можно использовать в качестве справочника по алгоритмам для программистов, исследователей и в качестве учебного пособия для студентов соответствующих специальностей.

Для программистов, исследователей и студентов

УДК 681.3.06
ББК 32.973.26-018.2

Translation from the English language edition: "The Algorithm Design Manual" by Steven S. Skiena; ISBN 978-1-84800-069-8. Copyright © 2008 Springer, The Netherlands as a part of Springer Science+Business Media. All rights reserved. Russian edition copyright © 2011 year by BHV – St.Petersburg. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Перевод английской редакции книги The Algorithm Design Manual, Steven S. Skiena; ISBN 978-1-84800-069-8. Copyright © 2008 Springer, The Netherlands as a part of Springer Science+Business Media. Все права защищены. Русская редакция издания выпущена издательством "БХВ-Петербург" в 2011 году. Все права защищены. Никакая часть настоящей книги не может быть воспроизведена или передана в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на то нет письменного разрешения издательства.

ISBN 978-1-84800-069-8 (англ.)
ISBN 978-5-9775-0560-4 (рус.)

© 2008 Springer, The Netherlands as a part of Springer Science+Business Media
© Перевод на русский язык "БХВ-Петербург", 2011

Оглавление

Предисловие ... 13
Читателю ... 13
Преподавателю ... 15
Благодарности ... 16
Ограничение ответственности ... 17

ЧАСТЬ I. ПРАКТИЧЕСКАЯ РАЗРАБОТКА АЛГОРИТМОВ ... 19

Глава 1. Введение в разработку алгоритмов ... 21
1.1. Оптимизация маршрута робота ... 22
1.2. Задача календарного планирования ... 26
1.3. Обоснование правильности алгоритмов ... 29
    1.3.1. Представление алгоритмов ... 30
    1.3.2. Задачи и свойства ... 31
    1.3.3. Демонстрация неправильности алгоритма ... 32
    1.3.4. Индукция и рекурсия ... 33
    1.3.5. Суммирование ... 35
1.4. Моделирование задачи ... 37
    1.4.1. Комбинаторные объекты ... 37
    1.4.2. Рекурсивные объекты ... 39
1.5. Истории из жизни ... 40
1.6. История из жизни. Моделирование проблемы ясновидения ... 41
1.7. Упражнения ... 45

Глава 2. Анализ алгоритмов ... 49
2.1. Модель вычислений RAM ... 49
    2.1.1. Анализ сложности наилучшего, наихудшего и среднего случая ... 50
2.2. Асимптотические обозначения ... 52
2.3. Скорость роста и отношения доминирования ... 55
    2.3.1. Отношения доминирования ... 56
2.4. Работа с асимптотическими обозначениями ... 58
    2.4.1. Сложение функций ... 58
    2.4.2. Умножение функций ... 58
2.5. Оценка эффективности ... 59
    2.5.1. Сортировка методом выбора ... 59
    2.5.2. Сортировка вставками ... 60
    2.5.3. Сравнение строк ... 61
    2.5.4. Умножение матриц ... 63

2.6. Логарифмы и их применение....64
2.6.1. Логарифмы и двоичный поиск....64
2.6.2. Логарифмы и деревья....64
2.6.3. Логарифмы и биты....65
2.6.4. Логарифмы и умножение....65
2.6.5. Быстрое возведение в степень....66
2.6.6. Логарифмы и сложение....66
2.6.7. Логарифмы и система уголовного судопроизводства....67
2.7. Свойства логарифмов....68
2.8. История из жизни. Загадка пирамид....69
2.9. Анализ высшего уровня (*)....72
2.9.1. Малораспространенные функции....73
2.9.2. Пределы и отношения доминирования....74
2.10. Упражнения....75

Глава 3. Структуры данных....84
3.1. Смежные и связные структуры данных....84
3.1.1. Массивы....85
3.1.2. Указатели и связные структуры данных....86
3.1.3. Сравнение....89
3.2. Стеки и очереди....90
3.3. Словари....91
3.4. Двоичные деревья поиска....95
3.4.1. Реализация двоичных деревьев....96
3.4.2. Эффективность двоичных деревьев поиска....100
3.4.3. Сбалансированные деревья поиска....101
3.5. Очереди с приоритетами....102
3.6. История из жизни. Триангуляция....104
3.7. Хэширование и строки....107
3.7.1. Коллизии....108
3.7.2. Эффективный метод поиска строк посредством хэширования....110
3.7.3. Выявление дубликатов с помощью хэширования....112
3.8. Специализированные структуры данных....113
3.9. История из жизни. Геном человека....114
3.10. Упражнения....118

Глава 4. Сортировка и поиск....123
4.1. Применение сортировки....123
4.2. Практические аспекты сортировки....126
4.3. Пирамидальная сортировка....128
4.3.1. Пирамиды....129
4.3.2. Создание пирамиды....132
4.3.3. Наименьший элемент пирамиды....133
4.3.4. Быстрый способ создания пирамиды (*)....135
4.3.5. Сортировка вставками....137
4.4. История из жизни. Билет на самолет....138
4.5. Сортировка слиянием. Метод "разделяй и властвуй"....141
4.6. Быстрая сортировка. Рандомизированная версия....143
4.6.1. Ожидаемое время исполнения алгоритма быстрой сортировки....146

4.6.2. Рандомизированные алгоритмы 147
4.6.3. Действительно ли алгоритм быстрой сортировки работает быстро? 150
4.7. Сортировка распределением. Метод блочной сортировки 150
4.7.1. Нижние пределы для сортировки 151
4.8. История из жизни. Адвокат Скиена 152
4.9. Двоичный поиск и связанные с ним алгоритмы 154
4.9.1. Частота вхождения элемента 155
4.9.2. Односторонний двоичный поиск 155
4.9.3. Корни числа 156
4.10. Метод "разделяй и властвуй" 156
4.10.1. Рекуррентные соотношения 157
4.10.2. Рекуррентные соотношения метода "разделяй и властвуй" 158
4.10.3. Решение рекуррентных соотношений типа "разделяй и властвуй" (*) 159
4.11. Упражнения 161

Глава 5. Обход графов 168
5.1. Разновидности графов 169
5.1.1. Граф дружеских отношений 172
5.2. Структуры данных для графов 174
5.3. История из жизни. Жертва закона Мура 178
5.4. История из жизни. Создание графа 181
5.5. Обход графа 184
5.6. Обход в ширину 185
5.6.1. Применение обхода 187
5.6.2. Поиск путей 188
5.7. Применение обхода в ширину 189
5.7.1. Компоненты связности 189
5.7.2. Раскраска графов двумя цветами 191
5.8. Обход в глубину 192
5.9. Применение обхода в глубину 195
5.9.1. Поиск циклов 196
5.9.2. Шарниры графа 196
5.10. Обход в глубину ориентированных графов 200
5.10.1. Топологическая сортировка 202
5.10.2. Сильно связные компоненты 203
5.11. Упражнения 207

Глава 6. Алгоритмы для работы со взвешенными графами 213
6.1. Минимальные остовные деревья 214
6.1.1. Алгоритм Прима 215
6.1.2. Алгоритм Крускала 218
6.1.3. Поиск-объединение 220
6.1.4. Разновидности остовных деревьев 223
6.2. История из жизни. И все на свете только сети 224
6.3. Поиск кратчайшего пути 227
6.3.1. Алгоритм Дейкстры 228
6.3.2. Кратчайшие пути между всеми парами вершин 231
6.3.3. Транзитивное замыкание 233

6.4. История из жизни. Печатаем с помощью номеронабирателя 234
6.5. Потоки в сетях и паросочетание в двудольных графах 239
6.5.1. Паросочетание в двудольном графе 239
6.5.2. Вычисление потоков в сети 240
6.6. Разрабатывайте не алгоритмы, а графы 244
6.7. Упражнения 246

Глава 7. Комбинаторный поиск и эвристические методы 251
7.1. Перебор с возвратом 251
7.1.1. Генерирование всех подмножеств 254
7.1.2. Генерирование всех перестановок 255
7.1.3. Генерирование всех путей в графе 256
7.2. Отсечение вариантов поиска 258
7.3. Судоку 259
7.4. История из жизни. Покрытие шахматной доски 264
7.5. Эвристические методы перебора 267
7.5.1. Произвольная выборка 268
7.5.2. Локальный поиск 271
7.5.3. Имитация отжига 274
7.5.4. Применение метода имитации отжига 278
7.6. История из жизни. Только это не радио 280
7.7. История из жизни. Отжиг массивов 283
7.8. Другие эвристические методы поиска 286
7.9. Параллельные алгоритмы 287
7.10. История из жизни. "Торопиться в никуда" 289
7.11. Упражнения 290

Глава 8. Динамическое программирование 293
8.1. Кэширование и вычисления 294
8.1.1. Генерирование чисел Фибоначчи методом рекурсии 294
8.1.2. Генерирование чисел Фибоначчи посредством кэширования 295
8.1.3. Генерирование чисел Фибоначчи посредством динамического программирования 297
8.1.4. Биномиальные коэффициенты 298
8.2. Поиск приблизительно совпадающих строк 300
8.2.1. Применение рекурсии для вычисления расстояния редактирования 301
8.2.2. Применение динамического программирования для вычисления расстояния редактирования 302
8.2.3. Восстановление пути 304
8.2.4. Разновидности расстояния редактирования 306
8.3. Самая длинная возрастающая последовательность 310
8.4. История из жизни. Эволюция омара 312
8.5. Задача разбиения 315
8.6. Синтаксический разбор 318
8.6.1. Триангуляция с минимальным весом 320
8.7. Ограничения динамического программирования. Задача коммивояжера 322
8.7.1. Вопрос правильности алгоритмов динамического программирования 323
8.7.2. Эффективность алгоритмов динамического программирования 324
8.8. История из жизни. Динамическое программирование и язык Prolog 325

8.9. История из жизни. Сжатие текста для штрих-кодов.....328
8.10. Упражнения.....332

Глава 9. Труднорешаемые задачи и аппроксимирующие алгоритмы.....338
9.1. Сведение задач.....338
9.1.1. Ключевая идея.....339
9.1.2. Задачи разрешимости.....340
9.2. Сведение для создания новых алгоритмов.....341
9.2.1. Поиск ближайшей пары.....341
9.2.2. Максимальная возрастающая подпоследовательность.....342
9.2.3. Наименьшее общее кратное.....343
9.2.4. Выпуклая оболочка (*).....344
9.3. Простые примеры сведения сложных задач.....345
9.3.1. Гамильтонов цикл.....345
9.3.2. Независимое множество и вершинное покрытие.....347
9.3.3. Задача о клике.....350
9.4. Задача выполнимости булевых формул.....351
9.4.1. Задача выполнимости в 3-конъюнктивной нормальной форме.....352
9.5. Нестандартные сведения.....353
9.5.1. Целочисленное программирование.....354
9.5.2. Вершинное покрытие.....356
9.6. Искусство доказательства сложности.....358
9.7. История из жизни. Наперегонки со временем.....360
9.8. История из жизни. Полный провал.....362
9.9. Сравнение классов сложности P и NP.....364
9.9.1. Верификация решения и поиск решения.....365
9.9.2. Классы сложности P и NP.....365
9.9.3. Почему задача выполнимости является самой сложной из всех сложных задач?.....366
9.9.4. NP-сложность по сравнению с NP-полнотой.....367
9.10. Решение NP-полных задач.....367
9.10.1. Аппроксимация вершинного покрытия.....368
9.10.2. Задача коммивояжера в евклидовом пространстве.....370
9.10.3. Максимальный бесконтурный подграф.....371
9.10.4. Задача о покрытии множества.....372
9.11. Упражнения.....375

Глава 10. Как разрабатывать алгоритмы.....380

ЧАСТЬ II. КАТАЛОГ АЛГОРИТМИЧЕСКИХ ЗАДАЧ.....385

Глава 11. Описание каталога.....387

Глава 12. Структуры данных.....389
12.1. Словарь.....389
12.2. Очереди с приоритетами.....395
12.3. Суффиксные деревья и массивы.....398
12.4. Графы.....402
12.5. Множества.....406
12.6. Kd-деревья.....410

Глава 13. Численные задачи ... 415
13.1. Решение системы линейных уравнений ... 416
13.2. Уменьшение ширины ленты матрицы ... 419
13.3. Умножение матриц ... 422
13.4. Определители и перманенты ... 425
13.5. Условная и безусловная оптимизация ... 427
13.6. Линейное программирование ... 431
13.7. Генерирование случайных чисел ... 435
13.8. Разложение на множители и проверка чисел на простоту ... 440
13.9. Арифметика произвольной точности ... 443
13.10. Задача о рюкзаке ... 448
13.11. Дискретное преобразование Фурье ... 451

Глава 14. Комбинаторные задачи ... 455
14.1. Сортировка ... 456
14.2. Поиск ... 461
14.3. Поиск медианы и выбор элементов ... 465
14.4. Генерирование перестановок ... 468
14.5. Генерирование подмножеств ... 472
14.6. Генерирование разбиений ... 475
14.7. Генерирование графов ... 479
14.8. Календарные вычисления ... 484
14.9. Календарное планирование ... 486
14.10. Выполнимость ... 489

Глава 15. Задачи на графах с полиномиальным временем исполнения ... 493
15.1. Компоненты связности ... 494
15.2. Топологическая сортировка ... 497
15.3. Минимальные остовные деревья ... 500
15.4. Поиск кратчайшего пути ... 505
15.5. Транзитивное замыкание и транзитивная редукция ... 511
15.6. Паросочетание ... 514
15.7. Задача поиска эйлерова цикла и задача китайского почтальона ... 517
15.8. Реберная и вершинная связность ... 521
15.9. Потоки в сети ... 524
15.10. Рисование графов ... 528
15.11. Рисование деревьев ... 531
15.12. Планарность ... 534

Глава 16. Сложные задачи на графах ... 538
16.1. Задача о клике ... 539
16.2. Независимое множество ... 542
16.3. Вершинное покрытие ... 544
16.4. Задача коммивояжера ... 547
16.5. Гамильтонов цикл ... 551
16.6. Разбиение графов ... 554
16.7. Вершинная раскраска ... 557
16.8. Реберная раскраска ... 561
16.9. Изоморфизм графов ... 563

16.10. Дерево Штейнера....568
16.11. Разрывающее множество ребер или вершин....572

Глава 17. Вычислительная геометрия....576
17.1. Элементарные задачи вычислительной геометрии....577
17.2. Выпуклая оболочка....581
17.3. Триангуляция....585
17.4. Диаграммы Вороного....589
17.5. Поиск ближайшей точки....592
17.6. Поиск в области....596
17.7. Местоположение точки....599
17.8. Выявление пересечений....603
17.9. Разложение по контейнерам....607
17.10. Преобразование к срединной оси....610
17.11. Разбиение многоугольника на части....613
17.12. Упрощение многоугольников....615
17.13. Выявление сходства фигур....619
17.14. Планирование перемещений....621
17.15. Конфигурации прямых....625
17.16. Сумма Минковского....628

Глава 18. Множества и строки....631
18.1. Поиск покрытия множества....631
18.2. Задача укладки множества....635
18.3. Сравнение строк....638
18.4. Нечеткое сравнение строк....641
18.5. Сжатие текста....647
18.6. Криптография....651
18.7. Минимизация конечного автомата....656
18.8. Максимальная общая подстрока....659
18.9. Поиск минимальной общей надстроки....663

Глава 19. Ресурсы....666
19.1. Программные системы....666
  19.1.1. Библиотека LEDA....666
  19.1.2. Библиотека CGAL....667
  19.1.3. Библиотека Boost....668
  19.1.4. Библиотека GOBLIN....668
  19.1.5. Библиотека Netlib....668
  19.1.6. Коллекция алгоритмов ассоциации ACM....669
  19.1.7. Сайты SourceForge и CPAN....669
  19.1.8. Система Stanford GraphBase....669
  19.1.9. Пакет Combinatorica....670
  19.1.10. Программы из книг....670
19.2. Источники данных....672
19.3. Библиографические ресурсы....673
19.4. Профессиональные консалтинговые услуги....673

Список литературы....675

Предметный указатель....713

Предисловие

Многие профессиональные программисты, с которыми я встречался, не очень хорошо подготовлены к решению проблем разработки алгоритмов. Это прискорбно, так как методика разработки алгоритмов является одной из важнейших *технологий* вычислительной техники. Создание правильных, эффективных и реализуемых алгоритмов для решения реальных задач требует от разработчика знаний в двух областях:

- *Методика.* Хорошие разработчики алгоритмов знают несколько фундаментальных приемов, в число которых входят работа со структурами данных, динамическое программирование, поиск в глубину, перебор с возвратами и эвристика. Пожалуй, самым важным техническим приемом является моделирование — искусство абстрагирования от запутанного реального приложения к четко сформулированной задаче, поддающейся алгоритмическому решению.
- *Ресурсы.* Хорошие разработчики алгоритмов пользуются коллективным опытом предыдущих поколений разработчиков. Вместо того, чтобы создавать "с нуля" алгоритм для стоящей перед ними задачи, они сначала узнают, что уже известно о ней и ищут существующие реализации решения, чтобы использовать их в качестве отправной точки. Они знают много классических алгоритмических задач, которые служат исходным материалом для моделирования практически любого приложения.

Эта книга была задумана как руководство по разработке алгоритмов, в котором я планировал изложить технологию разработки комбинаторных алгоритмов. Она рассчитана как на студентов, так и на профессионалов. Книга состоит из двух частей. Часть I является собой общее руководство по техническим приемам разработки и анализа компьютерных алгоритмов. Часть II предназначена для использования в качестве справочного и познавательного материала и состоит из каталога алгоритмических ресурсов, реализаций и обширного списка литературы.

Читателю

Меня очень обрадовал теплый прием, с которым было встречено первое издание книги "Руководство по разработке алгоритмов", опубликованное в 1997 г. Она была признана уникальным руководством по применению алгоритмических приемов для решения задач, которые часто возникают в реальной жизни. Но с тех пор многое изменилось в этом мире. Если считать, что начало современной разработке и анализу алгоритмов было положено приблизительно в 1970 г., то получается, что около 30% современной истории алгоритмов приходится на время, прошедшее после первой публикации руководства.

Читатели одобрили три аспекта руководства: каталог алгоритмических задач, истории из жизни и электронную версию книги. Эти элементы были сохранены в настоящем издании.

- *Каталог алгоритмических задач.* Не так-то просто узнать, что уже известно о стоящей перед вами задаче. Именно поэтому в книге имеется каталог 75 наиболее важных задач, часто возникающих в реальной жизни. Просматривая его, студент или специалист-практик может быстро выяснить название своей задачи и понять, что о ней известно и как приступить к ее решению. Чтобы облегчить идентификацию, каждая задача в каталоге сопровождается рисунками состояния "до и после", иллюстрирующими требуемые характеристики входа и выхода. За этот каталог задач один остроумный рецензент предложил назвать мою книгу "Автостопом по алгоритмам".

  Каталог задач является самой важной частью этой книги. Обновляя каталог для этого издания, я собрал отзывы и рекомендации ведущих мировых экспертов по каждой содержащейся в нем задаче. Особое внимание было уделено обновлению программных реализаций решений задач.

- *Истории из жизни.* На практике алгоритмические задачи редко возникают в начале работы над большим проектом. Наоборот, они обычно появляются в виде подзадач, когда становится ясно, что программист не знает, что делать дальше, или что принятое решение ошибочно.

  Чтобы продемонстрировать, как алгоритмические задачи возникают в реальной жизни, в материал книги были включены правдивые истории, описывающие наш опыт по решению практических задач. При этом преследовалась цель показать, что разработка и анализ алгоритмов представляют собой не отвлеченную теорию, а важный инструмент, требующий умелого обращения.

  В этом издании сохранены все первоначальные истории из жизни (обновленные по мере необходимости), а также были добавлены новые истории, имеющие отношение к внешней сортировке, работе с графами, методу имитации отжига и другим алгоритмическим областям.

- *Электронная версия.* Поскольку человек практичный обычно ищет готовую программу, а не алгоритм, в книге даются ссылки на все имеющиеся рабочие реализации алгоритмов. Для удобства эти реализации собраны на одном веб-сайте **http://www.cs.sunysb.edu/~algorith**. После публикации книги этот веб-сайт долгое время был одним из первых в результатах поиска в Google по слову "algorithm".

  Кроме этого, в книге даны рекомендации, как найти подходящий код для решения той или иной задачи. Наличие данных реализаций сводит проблему разработки алгоритма к правильному моделированию приложения и избавляет разработчика от необходимости разбираться в подробностях самого алгоритма. Этот подход применяется на всем протяжении книги.

Важно обозначить то, чего нет в этой книге. Мы не уделяем большого внимания математическому обоснованию алгоритмов и в большинстве случаев ограничиваемся неформальными рассуждениями. В этой книге вы не найдете ни одной теоремы. Более подробную информацию вы можете получить, изучив приведенные программы или справочный материал. Цель данного руководства в том, чтобы как можно быстрее указать читателю верное направление движения.

Преподавателю

Эта книга содержит достаточно материала для курса "Введение в алгоритмы". Предполагается, что читатель обладает знаниями, полученными при изучении таких курсов, как "Структуры данных" или "Теория вычислительных машин и систем".

На сайте **http://www.algorist.com** можно загрузить полный набор лекционных слайдов для преподавания материала этой книги. Кроме того, я выложил аудио- и видеолекции с использованием этих слайдов для преподавания полного курса продолжительностью в один семестр. Таким образом, вы можете через Интернет воспользоваться моей помощью в преподавании вашего курса.

Главное внимание в книге уделяется разработке алгоритмов, а их анализ стоит на втором плане. Книгу можно использовать как для обычных лекционных курсов, так и для активного обучения, при котором преподаватель не читает лекцию, а руководит решением реальных задач в группе студентов. Истории из жизни предоставляют введение в активный метод обучения.

Книга была полностью переработана с целью облегчить ее использование в качестве учебника. Для настоящего издания характерны:

- *подробное изложение материала.* По сравнению с предыдущим изданием, объем первой части книги был увеличен вдвое. Однако дополнительный материал не увеличивает количество обсуждаемых тем, а служит для более полного и тщательного изложения основ;
- *обсуждение основ.* Учебники по разработке алгоритмов обычно представляют общеизвестные алгоритмы как нечто само собой разумеющееся и не обсуждают идеи, лежащие в их основе, или слабые места других подходов. Истории из жизни, приводимые в этой книге, иллюстрируют процесс выбора алгоритма на примерах решения некоторых прикладных задач, но я применил аналогичный подход и к изложению материала, касающегося классических алгоритмов;
- *остановки для размышлений.* В этих разделах я изложил ход собственных рассуждений (включая тупиковые решения) в процессе выполнения конкретного домашнего задания. Разделы "Остановка для размышлений" разбросаны по всему тексту, чтобы повысить активность читателей по решению задач. Ответы к задачам даются тут же;
- *переработаннные и новые домашние задания.* Настоящее издание книги содержит вдвое больше упражнений для домашней работы, чем предыдущее. Нечетко сформулированные задания были исправлены или заменены новыми. Каждой задаче присвоен уровень сложности от 1 до 10;
- *экзамены на основе материала книги.* Студентам моих курсов по изучению алгоритмов я обещаю, что все вопросы текущих контрольных работ и экзаменов в конце семестра будут взяты из домашних заданий в этой книге. Таким образом, студенты точно знают, что нужно изучать, чтобы успешно сдать экзамен. Для действенности этого подхода я тщательно подобрал количество, тип и сложность домашних заданий, следя за тем, чтобы количество задач было оптимальным;
- *разделы с подведением итогов.* В этих разделах делается акцент на основных понятиях, которые нужно усвоить в данной главе;

♦ *ссылки на задачи по программированию.* В конце упражнений для каждой главы даются ссылки на 3–5 задач по программированию, взятых с веб-сайта **http://www.programming-challenges.com**. Эти задания можно использовать, чтобы добавить практический компонент реализации алгоритмов к теоретическому курсу;

♦ *большой объем работающего программного кода вместо псевдокода.* В этой книге увеличено количество алгоритмов, написанных на реальных языках программирования (в частности, на языке С), за счет уменьшения объема псевдокода. Я считаю, что корректность и надежность проверенной работающей реализации дают ей преимущество над неформальным представлением простых алгоритмов. Полностью реализованные алгоритмы доступны на сайте **http://www.algorist.com**;

♦ *замечания к главам.* Каждая глава завершается кратким разделом с замечаниями, содержащими ссылки на основные источники информации и дополнительный справочный материал.

Благодарности

Обновление посвящения через десять лет после выхода первого издания книги заставляет задуматься о скоротечности времени. С тех пор Рени стала моей женой, а потом матерью наших двух детей, Бонни и Эбби. Мой отец ушел в мир иной, но моя мама и мои братья Лен и Роб продолжают играть важную роль в моей жизни. Я посвящаю эту книгу членам моей семьи, новым и старым, тем, кто с нами и тем, кто покинул нас.

Я бы хотел поблагодарить нескольких людей за их непосредственный вклад в новое издание. Эндрю Гон (Andrew Gaun) и Бетсон Томас (Betson Thomas) оказали мне помощь, в частности, при создании инфраструктуры нового сайта **http://www.cs.sunysb.edu/~algorith** и при решении некоторых вопросов по подготовке рукописи. Дэвид Грайз (David Gries) дал ценные рекомендации в объеме, превышающем мои ожидания. Химаншу Гупта (Himanshu Gupta) и Бин Танг (Bin Tang) отважно использовали рукописную версию этого издания в своих академических курсах. Я также выражаю благодарность редакторам издательства Springer-Verlag Уэйну Уиллеру (Wayne Wheeler) и Алану Уайлду (Allan Wylde).

Группа экспертов по разработке алгоритмов изучила материал книги, делясь со мной своими знаниями и извещая меня о новостях в этой области. Я благодарен всей группе, в состав которой входили:

Ами Амир (Ami Amir), Хёрв Бронниманн (Herve Bronnimann), Бернар Шазель (Bernard Chazelle), Крис Чу (Chris Chu), Скотт Коттон (Scott Cotton), Ефим Диниц (Yefim Dinitz), Комеи Фукуда (Komei Fukuda), Майкл Гудрич (Michael Goodrich), Ленни Хит (Lenny Heath), Сихат Имамоглу (Cihat Imamoglu), Тао Жянг (Tao Jiang), Дэвид Каргер (David Karger), Джузеппе Лиотта (Giuseppe Liotta), Альберт Мао (Albert Mao), Сильвано Мартелло (Silvano Martello), Кэтрин Мак-Геох (Catherine McGeoch), Курт Мельхорн (Kurt Mehlhorn), Скотт А. Митчелл (Scott A. Mitchell), Насер Мескини (Naceur Meskini), Джин Майерс (Gene Myers), Гонзало Наварро (Gonzalo Navarro), Стивен Норт (Stephen North), Джо О'Рурк (Joe O'Rourke), Майк Патерсон (Mike Paterson), Тео Павлидис (Theo Pavlidis), Сет Петти (Seth Pettie), Мишель Почиола (Michel Pocchiola), Барт Пренил (Bart Preneel), Томаш Радзик

(Tomasz Radzik), Эдвард Рейнголд (Edward Reingold), Фрэнк Раски (Frank Ruskey), Питер Сэндерс (Peter Sanders), Жоао Сетубал (Joao Setubal), Джонатан Шевчук (Jonathan Shewchuk), Роберт Скил (Robert Skeel), Дженз Стои (Jens Stoye), Торстен Суэл (Torsten Suel), Брюс Уотсон (Bruce Watson) и Ури Цвик (Uri Zwick).

Многие упражнения были созданы по подсказке коллег или под влиянием других работ. Восстановление первоначальных источников годы спустя является задачей не из легких, но на веб-сайте книги дается ссылка на первоисточник каждой задачи (насколько мне удавалось его вспомнить).

Было бы невежливо не поблагодарить людей, внесших важный вклад в первое издание книги. Рики Брэдли (Ricky Bradley) и Дарио Влах (Dario Vlah) создали мощную инфраструктуру для веб-сайта, логически стройную и легко расширяемую. Жонг Ли (Zhong Li) сделал большинство рисунков каталога задач с помощью графического редактора xfig. Ричард Крэндол (Richard Crandall), Рон Дэниэльсон (Ron Danielson), Такис Метаксас (Takis Metaxas), Дэйв Миллер (Dave Miller), Гири Нарасимхан (Giri Narasimhan) и Джо Закари (Joe Zachary) проверили черновые версии первого издания. Их содержательные отзывы и рекомендации помогли мне сформировать содержимое данного издания.

Большую часть моих знаний об алгоритмах я получил, изучая их совместно с моими аспирантами. Многие из них — Йо-Линг Лин (Yaw-Ling Lin), Сундарам Гопалакришнан (Sundaram Gopalakrishnan), Тинг Чен (Ting Chen), Фрэнсин Иванс (Francine Evans), Харальд Рау (Harald Rau), Рики Брэдли (Ricky Bradley) и Димитрис Маргаритис (Dimitris Margaritis) — являются персонажами историй, изложенных в книге. Мне всегда было приятно работать и общаться с моими друзьями и коллегами из Университета Стоуни Брук — Эсти Аркином (Estie Arkin), Майклом Бэндером (Michael Bender), Джи Гао (Jie Gao) и Джо Митчеллом (Joe Mitchell). И, наконец, хочу сказать спасибо Майклу Брокстайну (Michael Brochstein) и остальным жителям города за то, что познакомили меня с настоящей жизнью далеко за пределами Стоуни Брук.

Ограничение ответственности

Традиционно вину за любые недостатки в книге великодушно принимает на себя ее автор. Я же делать этого не намерен. Любые ошибки, недостатки или проблемы в этой книге являются виной кого-то другого; но я буду признателен, если вы поставите меня в известность о них, с тем, чтобы я знал, кто виноват.

Стивен С. Скиена
Кафедра вычислительной техники
Университет Стоуни Брук
Стоуни Брук, Нью-Йорк 11794-4400
http://www.cs.sunysb.edu/~skiena
Апрель 2008 г.

ЧАСТЬ I

Практическая разработка алгоритмов

ГЛАВА 1

Введение в разработку алгоритмов

Что такое алгоритм? Это процедура выполнения определенной задачи. Алгоритм является основополагающей идеей любой компьютерной программы.

Чтобы представлять интерес, алгоритм должен решать общую, корректно поставленную *задачу*. Определение задачи, решаемой с помощью алгоритма, дается описанием всего множества *экземпляров*, которые должен обработать алгоритм, и выхода, т. е. результата, получаемого после обработки одного из этих экземпляров. Описание одного из экземпляров задачи может заметно отличаться от формулировки общей задачи. Например, постановка *задача сортировки* делается таким образом:

ЗАДАЧА. Сортировка.

Вход. Последовательность из n элементов: $a_1, ..., a_n$.

Выход. Перестановка элементов входной последовательности таким образом, что для ее членов справедливо соотношение $a'_1 \leq a'_2 \leq ... \leq a'_{n-1} \leq a'_n$.

Экземпляром задачи сортировки может быть массив имен, например {Mike, Bob, Sally, Jill, Jan}, или набор чисел, например {154, 245, 568, 324, 654, 324}. Первым шагом к решению — определить общую задачу.

Алгоритм — это процедура, которая принимает любой из возможных входных экземпляров и преобразует его в соответствии с требованиями, указанными в условии задачи. Для решения задачи сортировки существует много разных алгоритмов. В качестве примера одного из таких алгоритмов можно привести метод *сортировки вставками*. Сортировка этим методом заключается во вставке в требуемом порядке элементов из неотсортированной части списка в отсортированную часть, первоначально содержащую один элемент. Реализация этого алгоритма на языке С представлена в листинге 1.1.

Листинг 1.1. Реализация алгоритма сортировки вставками

```
insertion_sort(item s[], int n)
{
   int i,j;                     /* Счетчики */
      for (i=1; i<n; i++) {
         j=i;
         while ((j>0) && (s[j] < s[j-1])) {
            swap(&s[j],&s[j-1]);
            j = j-1;
         }
      }
   }
```

А на рис. 1.1 показано применение этого алгоритма для сортировки определенного экземпляра — строки INSERTIONSORT.

```
I N S E R T I O N S O R T
I N S E R T I O N S O R T
I N S E R T I O N S O R T
E I N S R T I O N S O R T
E I N R S T I O N S O R T
E I N R S T I O N S O R T
E I I N R S T O N S O R T
E I I N O R S T N S O R T
E I I N N O R S T S O R T
E I I N N O R S S T O R T
E I I N N O O R S S T R T
E I I N N O O R R S S T T
E I I N N O O R R S S T T
```

Рис. 1.1. Пример использования алгоритма сортировки вставками (шкала времени направлена вниз)

Обратите внимание на универсальность этого алгоритма. Его можно применять как для сортировки слов, так и для сортировки чисел, используя соответствующую операцию сравнения для определения, какое из двух значений поставить первым. Можно с легкостью убедиться, что этот алгоритм правильно сортирует любой возможный набор входных величин в соответствии с нашим определением задачи сортировки.

Хороший алгоритм должен обладать тремя свойствами: быть корректным, эффективным и легкореализуемым. Получение комбинации всех трех свойств сразу может оказаться недостижимой задачей. В производственной обстановке любая программа, которая предоставляет достаточно хорошие результаты и не замедляет работу системы, в большинстве случаев является приемлемой, независимо от того, возможны ли улучшения этих показателей. В этой области вопрос получения самых лучших возможных результатов или достижения максимальной эффективности обычно возникает только в случае серьезных проблем с производительностью или с законом.

В этой главе основное внимание уделяется вопросу корректности алгоритмов, а их эффективность рассматривается в *главе 2*. Способность определенного алгоритма правильно решить поставленную задачу, т. е. его корректность, редко поддается очевидной оценке. Алгоритмы обычно сопровождаются доказательством их правильности в виде объяснения, почему для каждого экземпляра задачи будет выдан требуемый результат. Но прежде чем продолжить обсуждение темы, мы продемонстрируем, что аргумент "это очевидно" никогда не является достаточным доказательством правильности алгоритма.

1.1. Оптимизация маршрута робота

Рассмотрим задачу, которая часто возникает на производстве и транспорте. Допустим, что нам нужно запрограммировать роботизированный манипулятор, который применяется для припаивания контактов интегральной схемы к контактным площадкам на печатной плате. Чтобы запрограммировать манипулятор для выполнения этой задачи, сначала нужно установить порядок, в котором манипулятор припаивает первый кон-

такт, потом второй, третий и т. д., пока не будут припаяны все. После обработки последнего контакта манипулятор возвращается к исходной позиции первого контакта для обработки следующей платы. Таким образом, маршрут манипулятора является замкнутым маршрутом, или циклом.

Так как роботы являются дорогостоящими устройствами, мы хотим минимизировать время, затрачиваемое манипулятором на обработку платы. Будет логичным предположить, что манипулятор перемещается с постоянной скоростью; соответственно, время перемещения от одной точки к другой прямо пропорционально расстоянию между точками. То есть, нам нужно найти алгоритмическое решение такой задачи:

ЗАДАЧА. Оптимизация маршрута робота.

Вход. Множество S из n точек, лежащих на плоскости.

Выход. Самый короткий маршрут посещения всех точек множества S.

Прежде чем приступать к программированию маршрута манипулятора, нам нужно разработать алгоритм решения этой задачи. На ум может прийти несколько подходящих алгоритмов. Но самым подходящим будет *эвристический алгоритм ближайшего соседа* (nearest-neighbor heuristic). Начиная с какой-либо точки p_0, мы идем к ближайшей к ней точке (соседу) p_1. От точки p_1 мы идем к ее ближайшему еще не посещенному соседу, таким образом исключая точку p_0 из числа кандидатов на посещение. Процесс повторяется до тех пор, пока не останется ни одной не посещенной точки, после чего мы возвращаемся в точку p_0, завершая маршрут. Псевдокод эвристического алгоритма ближайшего соседа представлен в листинге 1.2.

Листинг 1.2. Эвристический алгоритм ближайшего соседа

```
NearestNeighbor(P)
  Из множества P выбираем и посещаем произвольную начальную точку p0
  p = p0
  i = 0
  Пока остаются непосещенные точки
    i = i + 1
    Выбираем и посещаем непосещенную точку pi, ближайшую к точке pi-1
    Посещаем точку pi
  Возвращаемся в точку p0 от точки pn-1
```

Этот алгоритм можно рекомендовать к применению по многим причинам. Он прост в понимании и легко реализуется. Вполне логично сначала посетить близлежащие точки, чтобы сократить общее время прохождения маршрута. Алгоритм дает отличные результаты для входного экземпляра, показанного на рис. 1.2.

Алгоритм ближайшего соседа достаточно эффективен, т. к. в нем каждая пара точек (p_i, p_j) рассматривается, самое большее, два раза: первый раз при добавлении в маршрут точки p_i , а второй — при добавлении точки p_j. При всех этих достоинствах алгоритм имеет всего лишь один недостаток — он совершенно неправильный.

Неправильный? Да как он может быть неправильным? Поясню: несмотря на то, что алгоритм всегда создает маршрут, этот маршрут не обязательно будет самым коротким возможным маршрутом, или хотя бы приближающимся к таковому. Рассмотрим множество точек, расположенных в линию, как показано на рис. 1.3.

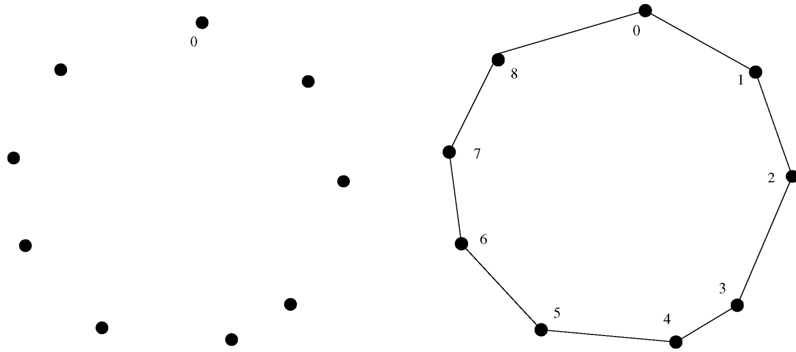

Рис. 1.2. Случай удачного входного экземпляра для эвристического алгоритма ближайшего соседа

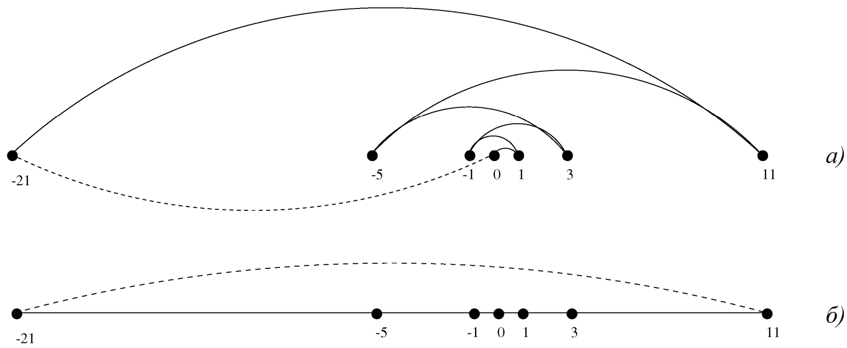

Рис. 1.3. Пример неудачного входного экземпляра для эвристического алгоритма ближайшего соседа *(а)* и оптимальное решение *(б)*

Цифры на рисунке обозначают расстояние от начальной точки до соответствующей точки справа или слева. Начав обход с точки 0 и посещая затем ближайшего непосещенного соседа текущей точки, мы будем метаться вправо-влево через нулевую точку, т. к. алгоритм не содержит указания, что нужно делать в случае одинакового расстояния между точками. Гораздо лучшее (более того, оптимальное) решение данного экземпляра задачи — начать обход с крайней левой точки и двигаться направо, посещая каждую точку, после чего возвратиться в исходное положение.

Представьте себе реакцию вашего начальника при виде манипулятора, мечущегося вправо-влево при выполнении такой простой задачи.

Можно сказать, что в данном случае проблема заключается в неудачном выборе отправной точки маршрута. Почему бы не начать маршрут с самой левой точки в качестве точки p_0? Это даст нам оптимальное решение данного экземпляра задачи.

Верно на все 100%, но лишь до тех пор, пока мы не развернем множество точек на 90 градусов, сделав все точки самыми левыми. А если к тому же немного сдвинуть первоначальную точку 0 влево, то она опять будет выбрана в качестве отправной. Теперь вместо дергания из стороны в сторону манипулятор будет скакать вверх-вниз, но время прохождения маршрута по-прежнему оставляет желать лучшего. Таким образом,

независимо от того, какую точку мы выберем в качестве исходной, алгоритм ближайшего соседа обречен на неудачу с некоторыми экземплярами задачи (т. е. с некоторыми множествами точек), и нам нужно искать другой подход. Условие, заставляющее всегда искать ближайшую точку, является излишне ограничивающим, т. к. оно принуждает нас выполнять нежелательные переходы. Задачу можно попробовать решить другим способом — соединяя пары самых близких точек, если такое соединение не вызывает никаких проблем, например, досрочного завершения цикла. Каждая вершина рассматривается как самостоятельная одновершинная цепочка. Соединив все вместе, мы получим одну цепочку, содержащую все точки. Соединив две конечные точки, мы получим цикл. На любом этапе выполнения этого эвристического алгоритма ближайших пар у нас имеется множество отдельных вершин и не имеющих общих вершин цепочек, которые можно соединить. Псевдокод соответствующего алгоритма показан в листинге 1.3.

Листинг 1.3. Эвристический алгоритм ближайших пар

```
ClosestPair(P)
  Пусть n — количество точек множества P.
  For i = 1 to n — 1 do
    d = ∞
    For каждой пары точек(s, t), не имеющих общих вершин цепей
      if dist(s, t) ≤ d then sm = s, tm = t и d = dist(s, t)
    Соединяем (sm,tm) ребром
  Соединяем две конечные точки ребром
```

Для экземпляра задачи, представленного на рис. 1.3, этот алгоритм работает должным образом. Сначала точка 0 соединяется со своими ближайшими соседями, точками 1 и –1. Потом соединение следующих ближайших пар точек выполняется поочередно влево-вправо, расширяя центральную часть по одному сегменту за проход. Эвристический алгоритм ближайших пар чуть более сложный и менее эффективный, чем предыдущий, но, по крайней мере, для данного экземпляра задачи он дает правильный результат.

Впрочем, это верно не для всех экземпляров. Посмотрите на результаты работы алгоритма на рис. 1.4, *а*. Данный входной экземпляр состоит из двух рядов равномерно расположенных точек. Расстояние между рядами $(1 - e)$ несколько меньше, чем расстояние между смежными точками в рядах $(1 + e)$. Таким образом, пары наиболее близких точек располагаются не по периметру, а напротив друг друга. Сперва противоположные точки соединяются попарно, а затем полученные пары соединяются поочередно по периметру. Общее расстояние маршрута алгоритма ближайших пар в этом случае будет равно $3(1-e)+2(1+e)+\sqrt{(1-e)^2+(2+2e)^2}$. Этот маршрут на 20% длиннее, чем маршрут на рис. 1.4, *б*, когда $e \approx 0$. Более того, есть входные экземпляры, дающие значительно худшие результаты, чем этот.

Таким образом, этот алгоритм тоже не годится. Какой из этих двух алгоритмов более эффективный? На этот вопрос нельзя ответить, просто посмотрев на них. Но очевидно, что оба алгоритма могут выдать очень плохие маршруты на некоторых с виду простых входных экземплярах.

Но каков же правильный алгоритм решения этой задачи? Можно попробовать перечислить все возможные перестановки множества точек, а потом выбрать перестановку,

сводящую к минимуму длину маршрута. Псевдокод этого алгоритма показан в листинге 1.4.

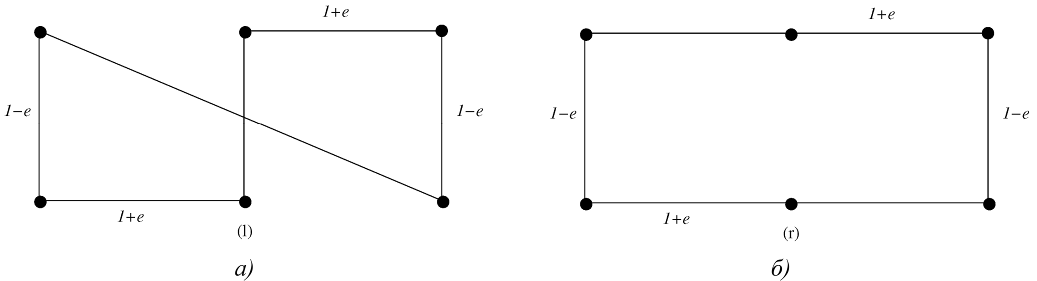

Рис. 1.4. Неудачный входной экземпляр для алгоритма ближайших пар *(а)* и оптимальное решение *(б)*

Листинг 1.4. Оптимальный алгоритм поиска маршрута

```
OptimalTSP(P)
  d = ∞
  For каждой перестановки Pi из общего числа перестановок n! множества
                                                                точек P
    If (cost(Pi) ≤ d) then d = cost(Pi) и Pmin = Pi
  Return Pmin
```

Так как рассматриваются все возможные упорядочения, то получение самого короткого маршрута гарантировано. Поскольку мы выбираем самую лучшую комбинацию, алгоритм правильный. В то же самое время он чрезвычайно медленный. Например, самый быстрый компьютер в мире не сможет в течение дня перечислить все 20! = 2 432 902 008 176 640 000 возможных перестановок 20 точек. А о реальных ситуациях, когда количество точек печатной платы достигает тысячи, можно и не говорить. Все компьютеры в мире, работая круглосуточно, не смогут даже приблизиться к решению этой задачи до конца существования Вселенной, а тогда решение этой задачи, скорее всего, уже не будет актуальным.

Поиском эффективного алгоритма решения этой задачи, называющейся *задачей коммивояжера* (traveling salesman problem, TSP), мы будем заниматься на протяжении большей части этой книги. Если же вам не терпится узнать решение уже сейчас, то вы можете посмотреть его в *разделе 16.4*.

Подведение итогов

Алгоритмы, которые всегда выдают правильное решение, коренным образом отличаются от *эвристических алгоритмов*, которые обычно выдают достаточно хорошие, но не гарантированные результаты.

1.2. Задача календарного планирования

Теперь рассмотрим задачу календарного планирования. Представьте, что вы кинозвезда и вам наперебой предлагают роли в разных кинофильмах, общее количество кото-

рых равно n. Каждое предложение имеет условие, что вы должны посвятить себя ему с первого до последнего дня съемок. Поэтому вы не можете сниматься одновременно в фильмах с полностью или частично накладывающимися периодами съемок.

Ваш критерий для принятия того или иного предложения довольно прост: вы хотите заработать как можно больше денег. Поскольку вам платят одинаково за каждый фильм, то вы стремитесь получить роли в как можно большем количестве фильмов, периоды съемок которых не конфликтуют.

На рис. 1.5 перечислены фильмы, в которых вам предлагают роли.

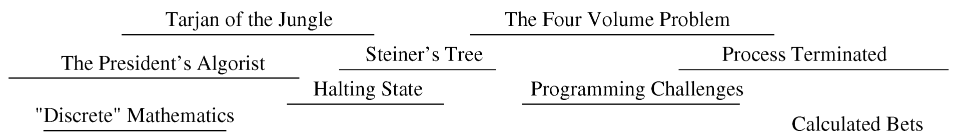

Рис. 1.5. Экземпляр задачи планирования непересекающихся календарных периодов

В данном случае очевидно, что вы можете сниматься, самое большее, в четырех фильмах — "Discrete Mathematics", "Programming Challenges", "Calculated Bets", а потом в "Halting State" или в "Steiner's Tree".

А в менее очевидных случаях вам (или вашему менеджеру) нужно будет решить следующую алгоритмическую задачу календарного планирования:

ЗАДАЧА. Планирование съемок в фильмах.

Вход. Множество I интервалов времени n в линейном порядке.

Выход. Самое большое подмножество непересекающихся интервалов времени, которое возможно во множестве I.

На ум может прийти несколько способов решения этой задачи. Один из них основан на представлении, что надо работать всегда, когда это возможно. Это означает, что вам нужно брать роль в фильме, съемки которого начинаются раньше всех других. Псевдокод этого алгоритма представлен в листинге 1.5.

Листинг 1.5. Алгоритм первой возможной работы

```
EarliestJobFirst(I)
   Из множества фильмов I берем роль в фильме j с самым ранним началом
   съемок, период которых не пересекается с периодом ваших предыдущих
   обязательств. Поступаем таким образом до тех пор, пока больше
   не останется таких фильмов.
```

Этот подход выглядит логично, по крайней мере, до тех пор, пока вы не осознаете, что хватаясь за самую раннюю работу, можете пропустить несколько других, если первый фильм является сериалом. Пример такой ситуации показан на рис. 1.6, *а*, где самым ранним фильмом является киноэпопея "War and Peace", которая закрывает перед вами все другие перспективы.

Этот пример заставляет искать другое решение. Проблема с фильмом "War and Peace" заключается в том, что его съемки длятся слишком долго. В таком случае, может быть,

вам следует брать роли только в фильмах с самыми короткими периодами съемок? Разве не очевидно, что чем быстрее вы закончите сниматься в одном фильме, тем раньше можно начать сниматься в другом, максимизируя таким образом количество фильмов в любой выбранный период времени? Псевдокод этого алгоритма представлен в листинге 1.6.

Рис. 1.6. Неудачные экземпляры задач для применения эвристики: самых ранних периодов *(а)*, самых коротких периодов *(б)*

Листинг 1.6. Алгоритм самого короткого периода

```
ShortestJobFirst(I)
  While (I ≠ ∅) do
    Из всего множества фильмов I берем фильм j с
    самым коротким периодом съемок
    Удаляем фильм j и любой другой фильм, период съемок которого
    пересекается с фильмом j, из множества доступных фильмов I
```

Но и этот подход окажется действенным только до тех пор, пока вы не увидите, что можете упустить возможность заработать больше (рис. 1.6, *б*). Хотя в данном случае потери меньше, чем в предыдущем случае, тем не менее вы получите только половину возможных заработков.

На данном этапе может показаться заслуживающим внимания алгоритм, который перебирает все возможные комбинации. Если отвлечься от проверки подмножеств интервалов (т. е. периодов съемок) на пересечение, этот алгоритм можно выразить псевдокодом, представленным в листинге 1.7.

Листинг 1.7. Алгоритм полного перебора

```
ExhaustiveScheduling(I)
  j = 0
  S_max = ∅
  For каждого из 2^n подмножеств Si множества интервалов I
    If (S_i непересекающееся) и (size(S_i) > j)
       then j = size(S_i) и S_max = S_i
  Return S_max
```

Но насколько эффективным будет такой алгоритм? Здесь ключевым ограничением является необходимость выполнить перечисление 2^n подмножеств n элементов. А положительным обстоятельством является то, что это *намного* лучше, чем перечисление всех $n!$ порядков n элементов, как предлагается в задаче оптимизации маршрута роботизированного манипулятора. В данном случае при $n = 20$ имеется около миллиона подмножеств, которые можно за несколько секунд перебрать на современном компью-

тере. Но когда выбор фильмов возрастет до $n = 100$, то 2^{100} будет намного больше, чем значение 20!, которое положило нашего робота на лопатки в предыдущей задаче.

Разница между задачей составления маршрута и задачей календарного планирования заключается в том, что для последней имеется алгоритм, который решает задачу и правильно, и эффективно. Для этого вам нужно брать роли только в фильмах с самым ранним окончанием съемок, т. е. выбрать такой временной интервал x, у которого правая конечная точка находится левее правых конечных точек всех прочих временных интервалов. Таким фильмом в нашем примере (см. рис. 1.5) является "Discrete Mathematics". Вполне возможно, что съемки других фильмов начались раньше, чем съемки фильма x, но все они должны пересекаться друг с другом (по крайней мере, частично), поэтому мы можем выбрать, самое большее, один фильм изо всей группы. Съемки фильма x закончатся раньше всего, поэтому остальные фильмы с накладывающимися съемочными периодами потенциально блокируют другие возможности, расположенные справа от них. Очевидно, что выбрав фильм x, вы никак не можете проиграть. Псевдокод эффективного алгоритма правильного решения задачи календарного планирования будет выглядеть так, как показано в листинге 1.8.

Листинг 1.8. Оптимальный алгоритм календарного планирования

```
OptimalScheduling(I)
  While (I ≠ ∅) do
    Из всего множества фильмов I выбираем фильм j с самым
          ранним окончанием съемок
    Удаляем фильм j и любой другой фильм, съемки которого
          пересекаются с фильмом j, из множества доступных фильмов I
```

Обеспечение оптимального решения для всего диапазона возможных входных экземпляров является трудной, но, как правило, выполнимой задачей. Важной частью процесса разработки такого алгоритма является поиск входных экземпляров задачи, которые опровергают наше допущение о правильности алгоритма-претендента на решение. Эффективные алгоритмы часто скрываются где-то совсем рядом, и в этой книге мы хотим помочь вам развить навыки их обнаружения.

> ***Подведение итогов***
>
> Кажущиеся вполне логичными алгоритмы очень легко могут оказаться неправильными. Правильность алгоритма требует тщательного доказательства.

1.3. Обоснование правильности алгоритмов

Будем надеяться, что предшествующие примеры продемонстрировали вам всю сложность темы правильности алгоритмов. Правильные алгоритмы выделяются из общего числа с помощью специальных инструментов, главный из которых называется *доказательством*.

Адекватное математическое доказательство состоит из нескольких частей. Прежде всего, требуется ясная и четкая формулировка того, что вы пытаетесь доказать. Потом необходим набор предположений, которые всегда считаются верными и поэтому исполь-

зуются как часть доказательства. Далее, цепь умозаключений приводит нас от начальных предположений к конечному утверждению, которое мы пытаемся доказать. Наконец, небольшой черный квадрат ■ в тексте указывает на конец доказательства.

В этой книге формальным доказательствам не уделяется большого внимания, т. к. правильное формальное доказательство привести очень трудно, а неправильное может вас сильно дезориентировать. На самом деле, доказательство является *демонстрацией*. Доказательства полезны только тогда, когда они простые и незамысловатые — ясные и лаконичные аргументы, объясняющие, почему алгоритм удовлетворяет требованию нетривиальной правильности.

Правильные алгоритмы требуют тщательного изложения и определенных усилий для доказательства как их правильности, так и того факта, что они *не* являются неправильными. В последующих разделах мы разработаем инструменты для достижения этих целей.

1.3.1. Представление алгоритмов

Цепь логических умозаключений об алгоритме невозможно построить без тщательного описания последовательности шагов, которые необходимо выполнить. Для этой цели наиболее часто употребляются, по отдельности или в совокупности, три формы представления алгоритма: обычный язык, псевдокод и язык программирования. Самым загадочным из этих средств представления алгоритма является псевдокод; это средство лучше всего можно определить как язык программирования, который никогда не выдает сообщений о синтаксических ошибках. Все три способа являются полезными, т. к. существует естественное стремление к компромиссу между легкостью восприятия и точностью представления алгоритма. Наиболее простым для понимания "языком программирования" является обычный язык, но в то же время он наименее точен. С другой стороны, такие языки, как Java или C/C++, позволяют точно выразить алгоритм, но создавать и понимать алгоритмы на этих языках задача не из легких. В отношении сложности применения и понимания псевдокод представляет золотую середину между этими двумя крайностями.

Выбор самого лучшего способа представления алгоритма зависит от ваших предпочтений. Я, например, сначала описываю свои алгоритмические идеи на обычном языке, а затем перехожу на более формальный псевдокод наподобие языка программирования или даже на настоящий язык программирования для уточнения сложных деталей.

Не допускайте ошибку, которую часто делают мои студенты, — используют псевдокод, чтобы приукрасить плохо определенную идею и придать ей более формальный и солидный вид. При описании алгоритма следует стремиться к ясности. Например, алгоритм ExhaustiveScheduling (см. листинг 1.7) можно было бы выразить на обычном языке так, как показано в листинге 1.9.

Листинг 1.9. Алгоритм полного перебора

```
ExhaustiveScheduling(I)
  Протестировать все 2^n подмножеств множества I и возвратить самое
    большое подмножество непересекающихся интервалов.
```

Подведение итогов

В основе любого алгоритма лежит *идея*. Если в описании алгоритма не просматривается ясно ваша идея, значит, вы используете для ее выражения нотацию слишком низкого уровня.

1.3.2. Задачи и свойства

Чтобы продемонстрировать правильность алгоритма, одного его описания недостаточно. Нам также нужно подробное описание задачи, которая подлежит решению.

Постановка задачи состоит из двух частей: набора допустимых входных экземпляров и требований к выходу алгоритма. Невозможно доказать правильность алгоритма для нечетко поставленной задачи. Иными словами, поставьте неправильно задачу, и вы получите неправильный ответ.

Постановки некоторых задач допускают слишком широкий диапазон входных экземпляров. Допустим, что в нашей задаче календарного планирования съемки фильмов могут прерываться на некоторое время. Например, съемки фильма могут быть запланированы на сентябрь и ноябрь, а октябрь свободен. Тогда календарный план съемок для любого фильма будет состоять из *набора* временных интервалов. В этом случае мы можем браться за роли в двух фильмах с чередующимися, но не пересекающимися периодами съемок. Такую общую задачу календарного планирования нельзя решить с помощью алгоритма самого раннего окончания съемок. Более того, для решения этой общей задачи вообще *не существует* эффективного алгоритма.

Подведение итогов

Важным и заслуживающим внимания приемом разработки алгоритмов является сужение множества допустимых экземпляров задачи до тех пор, пока не будет найден правильный и эффективный алгоритм. Например, задачу общих графов можно свести к задаче деревьев, или двумерную геометрическую задачу свести к одномерной.

При указании требований выхода алгоритма часто допускаются две ошибки. Первая из них — плохая формулировка вопроса. Примером может служить вопрос о *наилучшем* маршруте между двумя точками на карте при отсутствии определения, что значит "наилучший". Лучший в каком смысле? В смысле самого короткого расстояния, самого короткого времени прохождения или, может быть, минимального количества поворотов?

Второй ошибкой является формулирование составных целей. Каждый из трех только что упомянутых критериев наилучшего маршрута является четко определенной целью правильного эффективного алгоритма оптимизации. Но в качестве требования к решению из этих критериев можно выбрать только один. Например, формулировка: "Найти самый короткий маршрут от точки к точке, содержащий количество поворотов не более чем в два раза превышающее необходимое" является вполне четкой постановкой задачи. Но решить такую задачу очень трудно, т. к. решение требует сложного анализа.

Для примеров постановки задач читателю настоятельно рекомендуется ознакомиться с постановкой каждой из 75 задач во второй части этой книги. Правильная постановка задачи является важной частью ее решения. Изучение постановок всех этих классических задач поможет вам распознать задачи, постановкой и решением которых кто-то уже занимался до вас.

1.3.3. Демонстрация неправильности алгоритма

Самый лучший способ доказать *неправильность* алгоритма — найти экземпляр задачи, для которого алгоритм выдает неправильный ответ. Такие экземпляры задачи называются *контрпримерами*. Никто не бросится на защиту алгоритма, для которого был предоставлен контрпример. Вполне разумно выглядящие алгоритмы можно моментально опровергнуть посредством очень простых контрпримеров. Хороший контрпример должен обладать двумя важными свойствами:

- *проверяемостью.* Чтобы продемонстрировать, что некий входной экземпляр задачи является контрпримером для определенного алгоритма, требуется вычислить ответ, который алгоритм выдаст для данного экземпляра, и предоставить лучший ответ, с тем, чтобы доказать, что алгоритм не смог его найти;
- *простотой.* Хороший контрпример не содержит ничего лишнего и ясно демонстрирует, *почему* именно предлагаемый алгоритм неправильный. Поэтому обнаруженный контрпример следует упростить. Контрпример на рис. 1.6, *а* можно упростить и улучшить, сократив количество пересекающихся периодов с четырех до двух.

Развитие навыков поиска контрпримеров будет стоить затраченного времени. Этот процесс в чем-то подобен разработке наборов тестов для проверки компьютерных программ, но в нем главную роль играет удачная догадка, а не перебор вариантов. Приведу несколько советов.

- *Ищите мелкомасштабные решения.* Обратите внимание, что мои контрпримеры для задач поиска маршрута содержат шесть или меньше точек, а для задач календарного планирования — только три временных интервала. Это обстоятельство указывает на то, что если алгоритм неправильный, то доказать это можно на очень простом экземпляре. Алгористы-любители нередко создают большой запутанный экземпляр, с которым потом не могут справиться. А профессионалы внимательно изучат несколько небольших экземпляров, т. к. их легче осмыслить и проверить.
- *Рассмотрите все решения.* Для наименьшего нетривиального значения n имеется только небольшое количество экземпляров. Например, существуют только три представляющих интерес способа расположения двух интервалов на прямой: неперекрывающиеся интервалы, частично перекрывающиеся интервалы и полностью перекрывающиеся интервалы. Все случаи размещения на прямой трех интервалов (включая контрпримеры для обоих эвристических алгоритмов календарного планирования) можно создать, по-разному добавляя третий интервал к этим двум, расположенным указанными тремя способами.
- *Ищите слабое звено.* Если рассматриваемый вами алгоритм работает по принципу "всегда берем самое большее" (так называемый *жадный алгоритм*), то подумайте, почему этот подход может быть неправильным.
- *Ищите ограничения.* "Жадный" эвристический алгоритм можно сбить с толку необычным методом предоставления входного экземпляра, содержащего одинаковые элементы. В этой ситуации алгоритму будет не на чем основывать свое решение и он, возможно, возвратит неоптимальное решение.
- *Рассматривайте крайние случаи.* Многие контрпримеры представляют собой комбинацию большого и малого, правого и левого, многих и немногих, далекого и

близкого. Обычно легче осмыслить и проверить примеры по краям диапазона, чем из его середины. Рассмотрим в качестве примера два скопления точек, расстояние d между которыми намного превышает расстояние между точками в любом из них. Длина оптимального маршрута коммивояжера в этой ситуации будет практически равна $2d$, независимо от количества посещаемых точек, т. к. длина маршрута внутри каждого скопления точек несущественна.

Подведение итогов

Лучший способ опровергнуть правильность эвристического алгоритма — испытать его на контрпримерах.

1.3.4. Индукция и рекурсия

Один факт, что для данного алгоритма не был найден контрпример, вовсе не означает, что алгоритм правильный. Для этого требуется доказательство или демонстрация правильности. Для доказательства правильности алгоритма часто применяется математическая индукция.

Мои первые впечатления о математической индукции были таковы, словно это какое-то шаманство. Вы берете формулу типа $\sum_{i=1}^{n} i = n(n+1)/2$, доказываете ее для базового случая, например, 1 или 2, потом допускаете, что утверждение справедливо для $n-1$, и на основе этого допущения доказываете формулу для общего n. Это называется доказательством? Полнейший абсурд!

Мои первые впечатления о рекурсии в программировании были точно такими же — чистое шаманство. Программа проверяет входной аргумент на равенство базовому значению, например, 1 или 2. При отрицательном результате такой проверки более сложный случай решается путем разбивки его на части и вызова *этой же программы в качестве подпрограммы* для решения этих частей. Это называется программой? Полнейший абсурд!

Причиной, благодаря которой как рекурсия, так и математическая индукция кажутся шаманством, является тот факт, что рекурсия и *есть* математическая индукция. В обеих имеются общие и граничные условия, при этом общее условие разбивает задачу на все более мелкие части, а граничное, или начальное, условие завершает рекурсию. Если вы понимаете одно из двух, — или рекурсию, или индукцию, — вы в состоянии понять, как работает другое.

Мне приходилось слышать, что программист — это математик, который умеет строить доказательства только методом индукции. Частично дело в том, что программисты — никудышные построители доказательств, но главным образом в том, что многие изучаемые ими алгоритмы являются или рекурсивными, или инкрементными (поэтапными).

Рассмотрим, например, правильность алгоритма сортировки вставками, представленного в начале этой главы. Его правильность можно *обосновать* методом индукции:

- базовый экземпляр содержит всего лишь один элемент, а по определению одноэлементный массив является полностью отсортированным;

- мы можем допустить, что после первых $n - 1$ итераций сортировки вставками первые $n - 1$ элементов массива A будут полностью отсортированы;
- чтобы определить, куда следует вставить последний элемент x, нам нужно найти уникальную ячейку между наибольшим элементом, не превышающим x, и наименьшим элементом, большим чем x. Для этого мы сдвигаем все бóльшие элементы назад на одну позицию, создавая место для элемента x в требуемой позиции. ■

Но к индуктивным доказательствам нужно относиться с большой осторожностью, т. к. в цепь рассуждений могут вкрасться трудно распознаваемые ошибки. Прежде всего, это *граничные ошибки*. Например, в приведенном выше доказательстве правильности алгоритма сортировки вставками мы самоуверенно заявили, что между двумя элементами массива A имеется однозначно определяемая ячейка, в которую можно вставить наш элемент x, когда массив в нашем базовом экземпляре содержит только одну ячейку. Поэтому для правильной обработки частных случаев вставки минимальных или максимальных элементов необходимо соблюдать бóльшую осторожность.

Другой, более распространенный, тип ошибок в индуктивных доказательствах связан с небрежным подходом к расширению экземпляра задачи. Добавление всего лишь одного элемента к экземпляру задачи может полностью изменить оптимальное решение. Соответствующий пример для задачи календарного планирования показан на рис. 1.7.

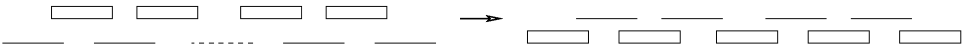

Рис. 1.7. Оптимальное решение (прямоугольники) до и после внесения изменений (пунктирная линия) в экземпляр задачи

После добавления нового временного интервала в экземпляр задачи новое оптимальное расписание может не содержать ни одного из временных интервалов любого оптимального решения, предшествующего изменению. Бесцеремонное игнорирование таких аспектов может вылиться в очень убедительное доказательство полностью неправильного алгоритма.

Подведение итогов

Математическая индукция обычно является верным способом проверки правильности рекурсивного алгоритма.

Остановка для размышлений: Правильность инкрементных алгоритмов

ЗАДАЧА. Доказать правильность рекурсивного алгоритма для инкрементации натуральных чисел, т. е. $y \to y + 1$, представленного в листинге 1.10.

Листинг 1.10. Алгоритм для инкрементации натуральных чисел

```
Increment (y)
  if y = 0 then return(1) else
    if (y mod 2) = 1 then
      return(2·Increment(⌊y/2⌋))
    else return(y + 1)
```

Решение. Лично мне правильность этого алгоритма определенно *не* очевидна. Но т. к. это рекурсивный алгоритм, а я — программист, мое естественное побуждение будет доказать его методом индукции.

Абсолютно очевидно, что алгоритм правильно обрабатывает базовый случай, когда $y = 0$. Совершенно ясно, что $0 + 1 = 1$ и, соответственно, возвращается значение 1.

Теперь допустим, что функция работает правильно для общего случая, когда $y = n - 1$. На основе этого допущения нам нужно продемонстрировать правильность алгоритма для случая, когда $y = n$. Для половины случаев алгоритм доказывается легко, в частности для четных чисел (для которых (`y mod 2) = 0`), т. к. $y + 1$ возвращается явно.

Но для нечетных чисел решение зависит от результата, возвращаемого выражением `Increment([y/2])`. Здесь нам хочется воспользоваться нашим индуктивным допущением, но оно не совсем правильно. Мы сделали допущение, что функция `Increment` работает правильно, когда $y = n - 1$, но для значения y, равного приблизительно половине этого значения, мы этого не допускали. Теперь мы можем усилить наше допущение, объявив, что общий случай выдерживается для всех $y \leq n - 1$. Это усиление никоим образом не затрагивает сам принцип, но необходимо, чтобы установить правильность алгоритма.

Теперь случаи нечетных y (т. е. $y = 2m + 1$ для целого числа m) можно обработать, как показано в листинге 1.11.

Листинг 1.11. Алгоритм для инкрементации нечетных натуральных чисел

```
2•Increment([(2m + 1)/2]) = 2•Increment(⌊m + 1/2⌋)
                          = 2•Increment(m)
                          = 2(m + 1)
                          = 2m + 2 = y + 1
```

решая, таким образом, общий случай. ■

1.3.5. Суммирование

При анализе алгоритмов часто приходится прибегать к математическим формулам суммирования. А процесс доказательства правильности формулы суммирования представляет собой классический пример применения математической индукции. В конце этой главы дается несколько упражнений, в которых требуется доказать формулу с помощью индукции. Чтобы сделать эти упражнения более понятными, напомню основные принципы суммирования.

Формулы суммирования представляют собой краткие выражения, описывающие сложение сколь угодно большой последовательности чисел. В качестве примера можно привести такую формулу:

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) + \ldots + f(n)$$

Суммирование многих алгебраических функций можно выразить простыми формулами в замкнутой форме. Например, поскольку сумма n единиц равна n, то:

$$\sum_{i=1}^{n} 1 = n$$

Сумму первых n целых чисел можно выразить через целые числа i и $(n - i + 1)$ следующим образом:

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n/2} (i + (n - i + 1)) = n(n+1) / 2$$

Очень пригодится в области анализа алгоритмов умение распознавать два основных класса формул суммирования:

- *Арифметические прогрессии.* Арифметическую прогрессию в виде формулы $S(n) = \sum_{i}^{n} i = n(n+1) / 2$ можно встретить в анализе алгоритма сортировки методом выбора. По большому счету, важным фактом является наличие квадратичной суммы, а не то, что константа равняется 1/2. В общем, для $p \geq 1$:

  $$S(n, p) = \sum_{i}^{n} i^p = \Theta(n^{p+1})$$

  Таким образом, сумма квадратов кубичная, а сумма кубов — "четверичная" (если вы не против употребления такого слова). Нотация $\Theta(x)$ (тета большое) подробно рассматривается в *разделе 2.2*.

  Для $p < -1$ эта сумма всегда стремится к константе, даже когда $n \to \infty$.

- *Геометрические прогрессии.* В геометрических прогрессиях индекс цикла играет роль показателя степени, т. е.:

  $$G(n, a) = \sum_{i=0}^{n} a^i = a(a^{n+1} - 1) / (a - 1)$$

  Сумма прогрессии зависит от ее знаменателя, т. е. числа a. При $a < 1$ эта сумма стремится к константе, даже когда $n \to \infty$.

  Данная сходимость последовательности оказывается большим подспорьем в анализе алгоритмов. Это означает, что если количество элементов растет линейно, то их сумма необязательно будет расти линейно, а может быть ограничена константой. Например, $1 + 1/2 + 1/4 + 1/8 + ... \leq 2$ независимо от количества элементов последовательности.

  При $a > 1$ сумма стремительно возрастает при добавлении каждого нового элемента, например, $1 + 2 + 4 + 8 + 16 + 32 = 63$.

  В самом деле, для $a > 1$ $G(n, a) = \Theta(a^{a+1})$.

Остановка для размышлений. Формулы факториала

ЗАДАЧА. Докажите методом индукции, что $\sum_{i=1}^{n} i \times i! = (n+1)! - 1$.

Решение. Индукционная парадигма прямолинейна: сначала подтверждаем базовый случай (здесь мы принимаем $n = 1$, хотя случай $n = 0$ был бы еще более общим):

$$\sum_{i=1}^{1} i \times i! = 1 = (1+1)! - 1 = 2 - 1 = 1$$

Теперь допускаем, что данное утверждение верно для всех чисел вплоть до n. Для доказательства общего случая $n + 1$ видим, что если мы вынесем наибольший член из-под знака суммы

$$\sum_{i=1}^{n+1} i \times i! = (n+1) \times (n+1)! + \sum_{i=1}^{n} i \times i!$$

то получим левую часть нашего индуктивного допущения. Заменяя правую часть, получаем:

$$\begin{aligned}\sum_{i=1}^{n+1} i \times i! &= (n+1) \times (n+1)! + (n+1)! - 1 \\ &= (n+1)! \times ((n+1)+1) - 1 \\ &= (n+2)! - 1\end{aligned}$$

Этот общий прием выделения наибольшего члена суммы для выявления экземпляра индуктивного допущения лежит в основе всех таких доказательств. ■

1.4. Моделирование задачи

Моделирование является искусством формулирования приложения в терминах точно поставленных, хорошо понимаемых задач. Правильное моделирование является ключевым аспектом применения методов разработки алгоритмов решения реальных задач. Более того, правильное моделирование может сделать ненужным разработку или даже реализацию алгоритмов, соотнося ваше приложение с ранее решенной задачей. Правильное моделирование является ключом к эффективному использованию материала во второй части этой книги.

В реальных приложениях применяются реальные объекты. Вам, может быть, придется работать над системой маршрутизации сетевого трафика, планированием использования классных комнат учебного заведения или поиском закономерностей в корпоративной базе данных. Большинство же алгоритмов спроектировано для работы со строго определенными абстрактными структурами, такими как перестановки, графы или множества. Чтобы извлечь пользу из литературы по алгоритмам, вам нужно научиться выполнять постановку задачи абстрактным образом, в терминах процедур над фундаментальными структурами.

1.4.1. Комбинаторные объекты

Очень велика вероятность, что над решаемой вами алгоритмической задачей уже работали другие, хотя, возможно, и в совсем иных контекстах. Но не надейтесь узнать, что известно о вашей конкретной "задаче оптимизации процесса A", поискав в книге словосочетание "процесс A". Вам нужно сформулировать задачу оптимизации вашего процесса в терминах вычислительных свойств стандартных структур, таких как:

- *перестановка* — упорядоченное множество элементов. Например, {1, 4, 3, 2} и {4, 3, 2, 1} являются двумя разными перестановками одного множества целых чисел. Мы уже видели перестановки в задачах оптимизации маршрута манипулятора и

календарного планирования. Перестановки будут вероятным исследуемым объектом в задаче поиска "размещения", "маршрута", "границ" или "последовательности";

- *подмножество* — выборка из множества элементов. Например, множества {1, 3, 4} и {2} являются двумя разными подмножествами множества первых четырех целых чисел. В отличие от перестановок, порядок элементов подмножества не имеет значения, поэтому подмножества {1, 3, 4} и {4, 3, 1} являются одинаковыми. В проблеме календарного планирования нам пришлось иметь дело с подмножествами. Подмножества будут вероятным исследуемым объектом в задаче поиска "кластера", "коллекции", "комитета", "группы", "пакета" или "выборки";

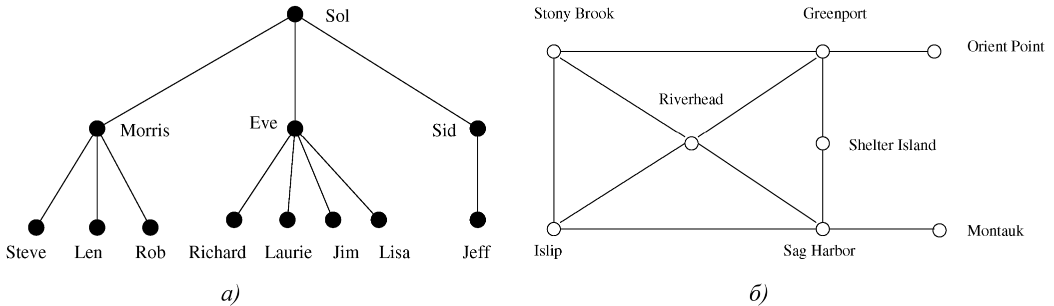

Рис. 1.8. Моделирование реальных структур с помощью деревьев и графов

- *дерево* — иерархическое представление взаимосвязей между объектами. Реальное применение деревьев показано на примере родословного древа семейства Скиена на рис. 1.8, *а*. Деревья будут вероятным исследуемым объектом в задаче поиска "иерархии", "отношений доминирования", "отношений предок/потомок" или "таксономии";
- *граф* — представление взаимоотношений между произвольными парами объектов. На рис. 1.8, *б* показана модель дорожной сети в виде графа, где вершины представляют населенные пункты, а ребра — дороги, соединяющие населенные пункты. Граф будет вероятным исследуемым объектом в задаче поиска "сети", "схемы", "инфраструктуры" или "взаимоотношений";
- *точка* — представление места в некотором геометрическом пространстве. Например, расположение автобусных остановок можно описать точками на карте (плоскости). Точка будет вероятным исследуемым объектом в задаче поиска "местонахождения", "позиции", "записи данных" или "расположения";
- *многоугольник* — представление области геометрического пространства. Например, с помощью полигона можно описать границы страны на карте. Многоугольники и многогранники будут вероятными исследуемыми объектами в задаче поиска "формы", "региона", "очертания" или "границы";
- *строка* — последовательность символов или шаблонов. Например, имена студентов можно представить в виде строк. Строка будет вероятным исследуемым объектом при работе с "текстом", "символами", "шаблонами" или "метками".

Для всех этих фундаментальных структур имеются соответствующие алгоритмы, которые представлены в каталоге задач во второй части этой книги. Важно ознакомиться с

этими задачами, т. к. они они изложены на языке, типичном для моделирования приложений. Чтобы научиться свободно владеть этим языком, просмотрите задачи в каталоге и изучите рисунки входа и выхода для каждой из них. Разобравшись в этих задачах, хотя бы на уровне рисунков и формулировок, вы будете знать, где искать возможный ответ в случае возникновения проблем в разработке вашего приложения.

В книге также представлены примеры успешного применения моделирования приложений в виде описаний решений реальных задач. Однако здесь необходимо высказать одно предостережение. Моделирование сводит разрабатываемое приложение к одной из многих существующих задач и структур. Такой процесс по своей природе является ограничивающим, и некоторые нюансы модели могут не соответствовать вашей конкретной задаче. Кроме того, встречаются задачи, которые можно моделировать несколькими разными способами.

Моделирование является всего лишь первым шагом в разработке алгоритма решения задачи. Внимательно отнеситесь к отличиям вашего приложения от потенциальной модели, но не спешите с заявлением, что ваша задача является уникальной и особенной. Временно игнорируя детали, которые не вписываются в модель, вы сможете найти ответ на вопрос, действительно ли они являются принципиально важными.

Подведение итогов

Моделирование разрабатываемого приложения в терминах стандартных структур и алгоритмов является важнейшим шагом в поиске решения.

1.4.2. Рекурсивные объекты

Научившись мыслить рекурсивно, вы научитесь определять большие сущности, состоящие из меньших сущностей *точно такого же типа, как и большие*. Например, если рассматривать дом как набор комнат, то при добавлении или удалении комнаты дом остается домом.

В мире алгоритмов рекурсивные структуры встречаются повсеместно. Более того, каждую из ранее описанных абстрактных структур можно считать рекурсивной. На рис. 1.9 видно, как легко они разбиваются на составляющие.

Перечислим возможные рекурсивные объекты.

- *Перестановки*. Удалив первый элемент перестановки $\{1,...,n\}$, мы получим перестановку остающихся $n - 1$ элементов.
- *Подмножества*. Каждое множество элементов $\{1,...,n\}$ содержит подмножество $\{1,...,n - 1\}$, являющееся результатом удаления элемента n, если такой имеется.
- *Деревья*. Что мы получим, удалив корень дерева? Правильно, коллекцию меньших деревьев. А что мы получим, удалив какой-либо лист дерева? Немного меньшее дерево.
- *Графы*. Удалите любую вершину графа, и вы получите меньший граф. Теперь разобьем вершины графа на две группы, правую и левую. Разрезав все ребра, соединяющие эти группы, мы получим два меньших графа и набор разорванных ребер.

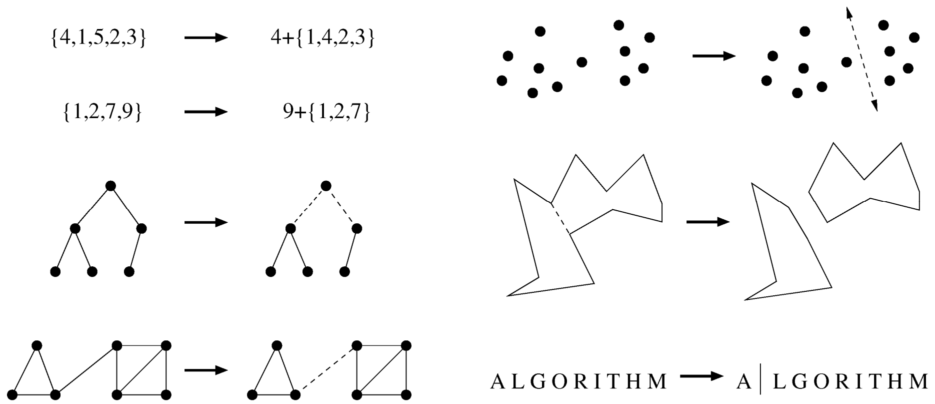

Рис. 1.9. Рекурсивное разложение комбинаторных объектов: перестановки, подмножества, деревья, графы, множества точек, многоугольники и строки

- *Точки*. Возьмем облако точек и разобьем его линией на две группы. Теперь у нас есть два меньших облака точек.
- *Многоугольники*. Соединив хордой две несмежные вершины простого многоугольника с n вершинами, мы получим два меньших многоугольника.
- *Строки*. Что мы получим, удалив первый символ строки? Другую, более короткую строку.

Для рекурсивного описания объектов требуются как правила разложения, так и базовые объекты, а именно спецификация простейшего объекта, далее которого разложение не идет. Такие базовые объекты обычно легко определить. Перестановки и подмножества нулевого количества объектов обозначаются как {}. Наименьшее дерево или граф состоят из одной вершины, а наименьшее облако точек состоит из одной точки. С многоугольниками немного посложнее; наименьшим многоугольником является треугольник. Наконец, пустая строка содержит нулевое количество знаков. Решение о том, содержит ли базовый объект нулевое количество элементов или один элемент, является, скорее, вопросом вкуса и удобства, нежели какого-либо фундаментального принципа.

Такие рекурсивные разложения применяются для определения многих алгоритмов, рассматриваемых в этой книге. Обращайте на них внимание.

1.5. Истории из жизни

Самый лучший способ узнать, какое огромное влияние тщательная разработка алгоритма может иметь на производительность, — ознакомиться с примерами из реальной жизни. Внимательно изучая опыт других людей, мы учимся использовать этот опыт в нашей собственной работе.

В различных местах этой книги приводится несколько рассказов об успешном (а иногда и неуспешном) опыте нашей команды в разработке алгоритмов решения реальных задач. Я надеюсь, что вы сможете перенять и усвоить опыт и знания, полученные нами

в процессе этих разработок, чтобы использовать их в качестве моделей для ваших собственных решений.

Все, изложенное в этих рассказах, действительно произошло. Конечно же, в изложении истории слегка приукрашены, и диалоги в них были отредактированы. Но я приложил все усилия, чтобы правдиво описать весь процесс, начиная от постановки задачи до выдачи решения, чтобы вы могли проследить его в действии.

В своих рассказах я пытаюсь уловить образ мышления алгориста в процессе решения задачи.

Эти истории из жизни обычно затрагивают, по крайней мере, одну, а часто несколько, задач из каталога во второй части книги. Когда такая задача встречается, дается ссылка на соответствующий раздел каталога. Таким образом подчеркиваются достоинства моделирования разрабатываемого приложения в терминах стандартных алгоритмических задач. Пользуясь каталогом задач, вы сможете в любое время получить известную информацию о решаемой задаче.

1.6. История из жизни. Моделирование проблемы ясновидения

Я занимался обычными делами, когда на меня нежданно-негаданно свалился этот запрос в виде телефонного звонка.

— Профессор Скиена, я надеюсь, что Вы сможете помочь мне. Я — президент компании Lotto System Group, Inc., и нам нужен алгоритм решения проблемы, возникающей в нашем последнем продукте.

— Буду рад помочь Вам, — ответил я. В конце концов, декан моего инженерного факультета всегда призывает преподавательский состав к более широкому взаимодействию с производством.

— Наша компания продает программу, предназначенную для развития способностей наших клиентов к ясновидению и предсказанию выигрышных лотерейных номеров[1]. Стандартный лотерейный билет содержит шесть номеров, выбираемых из большего количества последовательных номеров, например, от 1 до 44. Таким образом, шансы выигрыша любой комбинации номеров очень небольшие. Но после соответствующей тренировки наши клиенты могут мысленно увидеть, скажем, 15 номеров из 44 возможных, по крайней мере, четыре из которых будут на выигрышном билете. Вы все пока понимаете?

— Скорее нет, чем да, — сказал я, но тут вспомнил, что наш декан призывает нас взаимодействовать с производством.

— Наша проблема заключается в следующем. После того, как ясновидец сузил выбор номеров от 44 до 15, из которых он уверен в правильности, по крайней мере, четырех, нам нужно найти эффективный способ применить эту информацию. Допустим, угадавшему, по крайней мере, три правильных номера выдается денежный приз. Нам ну-

[1] Это реальный случай.

жен алгоритм, чтобы составить наименьший набор билетов, которые нужно купить, чтобы гарантировать выигрыш, по крайней мере, одного приза.

— При условии, что ясновидец не ошибается?

— Да, при условии, что ясновидец не ошибается. Нам нужна программа, которая распечатывает список всех возможных комбинаций выигрышных номеров билетов, которые он должен купить с минимальными затратами. Можете ли вы помочь нам с решением этой задачи?

Может быть, они и в самом деле были ясновидцами, т. к. они обратились как раз туда, куда надо. Определение наилучшего подмножества номеров билетов попадает в разряд комбинаторных задач. А точнее, это какой-то тип задачи о покрытии множества, в которой каждый покупаемый билет "покрывает" некоторые из возможных четырехэлементных подмножеств увиденного ясновидцем пятнадцатиэлементного множества. Определение наименьшего набора билетов для покрытия всех возможностей представляет собой особый экземпляр NP-полной задачи о *покрытии множества* (рассматривается в *разделе 18.1*) и считается вычислительно неразрешимой.

Данная задача действительно была особым экземпляром задачи о покрытии множества, полностью определяемая всего лишь четырьмя числами: размером n возможного множества S ($n \approx 15$), количеством номеров k на каждом билете ($k \approx 6$), количеством обещаемых ясновидцем выигрышных номеров j из множества S ($j = 4$) и, наконец, количеством совпадающих номеров l, необходимых, чтобы выиграть приз ($l = 3$). На рис. 1.10 показано покрытие экземпляра меньшего размера, где $n = 5$, $j = k = 3$, $l = 2$.

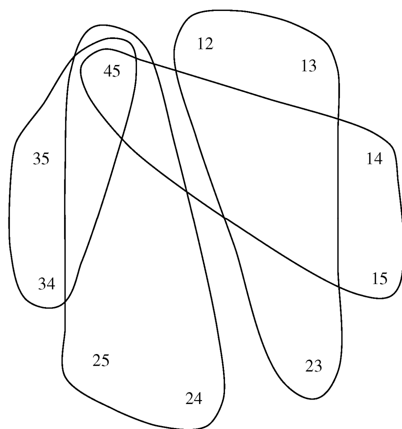

Рис. 1.10. Покрытие всех пар множества {1, 2, 3, 4, 5} номерами билетов {1, 2, 3}, {1, 4, 5}, {2, 4, 5}, {3, 4, 5}

— Хотя найти точный минимальный набор билетов с помощью эвристических методов будет трудно, я должен буду дать вам покрывающий набор номеров, достаточно близкий к самому дешевому, — ответил я ему. — Вам этого будет достаточно?

— При условии, что ваша программа создает лучший набор номеров, чем программа моего конкурента, это будет то, что надо. Его система не всегда гарантирует выигрыш. Очень признателен за вашу помощь, профессор Скиена.

— Один вопрос напоследок. Если с помощью вашей программы люди могут натренироваться выбирать выигрышные лотерейные билеты, то почему вы сами не пользуетесь ею, чтобы выигрывать в лотерею?

— Надеюсь встретиться с Вами в ближайшее время, профессор Скиена. Благодарю за помощь.

Я повесил трубку и начал обдумывать дальнейшие действия. Задача выглядела, как идеальный проект для какого-либо смышленого студента. После моделирования задачи посредством множеств и подмножеств основные компоненты решения выглядели довольно просто.

- Нам было нужна возможность генерировать все подмножества k номеров из потенциального множества S. Алгоритмы генерирования и ранжирования подмножеств представлены в *разделе 14.5*.
- Нам была нужна правильная формулировка, что именно означает требование иметь покрывающее множество приобретенных билетов. Очевидным критерием того, что требование выполнено, мог быть наименьший набор билетов, включающий, по крайней мере, один билет, содержащий каждое из $\binom{n}{l}$ l-подмножеств множества S, за которое может выдаваться приз.
- Нам нужно было отслеживать уже рассмотренные призовые комбинации. Нам нужны такие комбинации номеров билетов, чтобы покрыть как можно больше еще не охваченных призовых комбинаций. Текущие охваченные комбинации являются подмножеством всех возможных комбинаций. Структуры данных для подмножеств рассматриваются в *разделе 12.5*. Наилучшим кандидатом выглядел вектор битов, который бы за постоянное время давал ответ, охвачена ли уже данная комбинация.
- Нужен был механизм поиска, чтобы решить, какой следующий билет покупать. Для небольших множеств можно было проверить все возможные подмножества комбинаций и выбрать из них самую меньшую. Для множеств большего размера выигрышные комбинации номеров билетов для покупки можно было выбирать с помощью какого-либо процесса рандомизированного поиска наподобие имитации отжига (см. *раздел 7.5.3*), чтобы покрыть как можно больше комбинаций. Выполняя эту рандомизированную процедуру несколько раз и выбирая самые лучшие решения, мы смогли бы, скорее всего, получить хороший набор комбинаций номеров билетов.

Опуская детали механизма поиска, требуемый алгоритм можно выразить на псевдокоде, как показано в листинге 1.12.

Листинг 1.12. Поиск набора призовых комбинаций

```
LottoTicketSet (n,k,l)

  Инициализируем как "ложь" все элементы (n l)-элементного
                  вектора разрядов V
    While V содержит элементы "ложь"
      Выбираем k-элементное подмножество T множества {1,..., n}
      в качестве следующей комбинации номеров покупаемого билета
```

```
    For каждого из l-элементных подмножеств Ti множества T,
    V[rank(Ti}] = "истина"
  Выдаем набор комбинаций номеров билетов для покупки
```

Способный студент Фаяз Юнас (Fayyaz Younas) принял вызов реализовать алгоритм. На основе представленной основы он реализовал алгоритм поиска методом полного перебора и смог найти оптимальные решения задач, в которых $n \leq 5$. Для решения задачи с большим значением n он реализовал процедуру рандомизированного поиска, которую отлаживал, пока не остановился на самом лучшем варианте. Наконец наступил день, когда мы могли позвонить в компанию Lotto Systems Group и сказать им, что мы решили задачу.

— Результат работы нашей программы таков: оптимальным решением для $n = 15$, $k = 6$, $j = 4$, $l = 3$ будет покупка 28 билетов.

— Двадцать восемь билетов! — выразил недовольство президент. — В вашей программе, должно быть, есть ошибка. Вот пять билетов, которых будет достаточно, чтобы покрыть все варианты *дважды*: {2, 4, 8, 10, 13, 14}, {4, 5, 7, 8, 12, 15}, {1, 2, 3, 6, 11, 13}, {3, 5, 6, 9, 10, 15}, {1, 7, 9, 11, 12, 14}.

Мы повозились с этим примером немного и должны были признать, что он был прав. *Мы неправильно смоделировали задачу!* В действительности, нам не нужно было покрывать явно все возможные выигрышные комбинации. Объяснение представлено на рис. 1.11 в виде двухбилетного решения нашего предыдущего четырехбилетного примера.

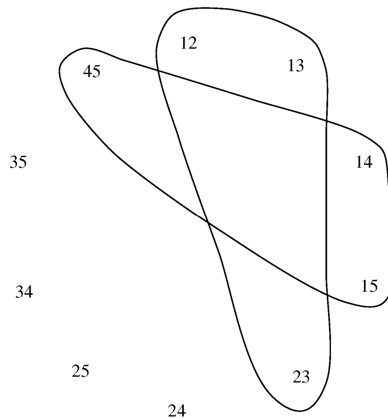

Рис. 1.11. Гарантирование выигрышной комбинации из двух номеров множества {l, 2, 3, 4, 5} при использовании только комбинации номеров {l, 2, 3} и {l, 4, 5}

Такие малообещающие комбинации, как {2, 3, 4} и {3, 4, 5}, имеют совпадающие пары комбинаций номеров в билетах, показанных на рис. 1.11. Мы пытались покрыть слишком много комбинаций, а дрожащие над каждой копейкой ясновидцы не желали платить за такую расточительность.

К счастью, у этой истории хороший конец. Общий принцип нашего поискового решения оставался применимым для реальной задачи. Все, что нужно было сделать, так это

исправить, какие подмножества засчитываются за покрытие данным набором комбинаций номеров билетов. Сделав это исправление, мы получили требуемые результаты. В компании Lotto Systems Group с благодарностью приняли нашу программу для внедрения в свой продукт. Будем надеяться, что они сорвут большой куш с ее помощью.

Мораль этой истории заключается в том, что необходимо удостовериться в правильности моделирования задачи, прежде чем пытаться решить ее. В нашем случае мы разработали правильную модель, но недостаточно глубоко проверили ее, перед тем, как начинать создавать программу на ее основе. Наше заблуждение было бы быстро выявлено, если бы, прежде чем приступать к решению реальной проблемы, мы проработали небольшой пример вручную и обсудили результаты с нашим клиентом. Но мы смогли выйти из этой ситуации с минимальными отрицательными последствиями благодаря правильности нашей первоначальной формулировки и использованию четко определенных абстракций для таких задач, как генерирование k-элементных подмножеств методом ранжирования, структуры данных множества и комбинаторного поиска.

Замечания к главе

В каждой хорошей книге, посвященной алгоритмам, отражается подход ее автора к их разработке. Тем, кто хочет ознакомиться с альтернативными точками зрения, особенно рекомендуется прочитать книги [CLRS01], [KT06] и [Man89].

Формальные доказательства правильности алгоритма являются важными и заслуживают более полного рассмотрения, чем можно предоставить в этой главе. Методы верификации программ обсуждаются в книге [Gri89].

Задача календарного планирования ролей в фильмах является особым случаем общей задачи *независимого множества*, которая рассматривается в *разделе 16.2*. В качестве входных экземпляров допускаются только интервальные графы, в которых вершины графа G можно представить линейными интервалами, а (i, j) является ребром G тогда и только тогда, когда интервалы пересекаются. Этот интересный и важный класс графов подробно рассматривается в книге [Gol04].

Колонка "Программистские перлы" Джона Бентли (Jon Bentley) является, наверное, самой известной коллекцией историй из жизни разработчиков алгоритмов. Колонка первоначально публиковалась в журнале "Communications of the ACM", а потом ее материалы были изданы в двух книгах, [Ben90] и [Ben99]. Еще одна прекрасная коллекция историй из жизни собрана в книге [Bro95]. Хотя эти истории имеют явный уклон в сторону разработки и проектирования программного обеспечения, они также представляют собой кладезь мудрости. Все программисты должны прочитать эти книги, чтобы получить и пользу, и удовольствие.

Наше решение задачи о покрытии множества лотерейных билетов более полно представлено в книге [YS96].

1.7. Упражнения

Поиск контрпримеров

1. [3] Докажите, что значение $a + b$ может быть меньшим, чем значение $\min(a, b)$.
2. [3] Докажите, что значение $a \times b$ может быть меньшим, чем значение $\min(a, b)$.

3. [5] Начертите сеть дорог с двумя точками a и b, такими, что маршрут между ними, преодолеваемый за кратчайшее время, не является самым коротким.
4. [5] Начертите сеть дорог с двумя точками a и b, самый короткий маршрут между которыми не является маршрутом с наименьшим количеством поворотов.
5. [4] Задача о рюкзаке: имея множество целых чисел $S = \{s_1, s_2, ..., s_n\}$ и целевое число T, найти такое подмножество множества S, сумма которого в точности равна T. Например, множество $S = \{l, 2, 5, 9, 10\}$ содержит такое подмножество, сумма элементов которого равна $T = 22$, но не $T = 23$.

   Найти контрпримеры для каждого из следующих алгоритмов решения задачи о рюкзаке, т. е., нужно найти такое множество S и число T, при которых подмножество, выбранное с помощью данного алгоритма, не до конца заполняет рюкзак, хотя правильное решение и существует:

   - вкладывать элементы множества S в рюкзак в порядке слева направо, если они подходят (т. е., алгоритм "первый подходящий");
   - вкладывать элементы множества S в рюкзак в порядке от наименьшего до наибольшего (т. е., используя алгоритм "первый лучший");
   - вкладывать элементы множества S в рюкзак в порядке от наибольшего до наименьшего.
6. [5] Задача о покрытии множества: имея семейство подмножеств $S_1, ..., S_m$ универсального множества $U = \{l,...,n\}$, найдите семейство подмножеств $T \subset S$ наименьшей мощности, чтобы $\bigcup_{t_i} \in T^{t_i} = U$. Например, для семейства подмножеств $S_1 = \{l, 3, 5\}$, $S_2 = \{2, 4\}$, $S_3 = \{l, 4\}$ и $S_4 = \{2, 5\}$ покрытием множества будет семейство подмножеств S_1 и S_2. Приведите контрпример для следующего алгоритма: выбираем самое мощное подмножество для покрытия, после чего удаляем все его элементы из универсального множества; повторяем добавление подмножества, содержащего наибольшее количество неохваченных элементов, пока все элементы не будут покрыты.

Доказательство правильности

7. [3] Докажите правильность следующего рекурсивного алгоритма умножения двух натуральных чисел для всех целочисленных констант $c \geq 2$:

```
function multiply(y, z)
               comment Return произведение yz.
       1.      if z = 0 then return(0) else
       2.      return(multiply(cy,[z/c]) + y·(z mod c))
```

8. [3] Докажите правильность следующего алгоритма вычисления полинома $P(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$:

```
function horner(A, x)
  p = An
  for i from n-1 to 0
    p = p * x + Ai
  return p
```

9. [3] Докажите правильность следующего алгоритма сортировки:

```
function bubblesort (A : list[l... n])
  var int i,j
```

```
for i from n to 1
  for j from 1 to i - 1
    if (A[j]>A[j+1]
      меняем местами значения A[j] и A[j + 1]
```

Математическая индукция

Для доказательства пользуйтесь методом математической индукции.

10. [3] Докажите, что $\sum_{i=1}^{n} i = n(n+1)/2$ для $n \geq 0$.

11. [3] Докажите, что $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$ для $n \geq 0$.

12. [3] Докажите, что $\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4$ для $n \geq 0$.

13. [3] Докажите, что $\sum_{i=1}^{n} i(i+1)(i+2) = n(n+1)(n+2)(n+3)/4$.

14. [5] Докажите, что $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$ для $n \geq 1$, $a \neq 1$.

15. [3] Докажите, что $\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$ для $n \geq 0$.

16. [3] Докажите, что $n^3 + 2n$ делится на 3 для $n \geq 0$.

17. [3] Докажите, что дерево с n вершинами имеет в точности $n - 1$ ребер.

18. [3] Докажите, что сумма кубов первых n положительных целых чисел равна квадрату суммы этих целых чисел, т. е.,

$$\sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$$

Приблизительные подсчеты

19. [3] Содержат ли все ваши книги, по крайней мере, миллион страниц? Каково общее количество страниц всех книг в вашей институтской библиотеке?

20. [3] Сколько слов содержит эта книга?

21. [3] Сколько часов составляет один миллион секунд? А сколько дней? Выполните все необходимые вычисления в уме.

22. [3] Сколько городов и поселков в Соединенных Штатах?

23. [3] Сколько кубических километров воды изливается из устья Миссисипи каждый день? Не пользуйтесь никакой справочной информацией. Опишите все предположения, сделанные вами для получения ответа.

24. [3] В каких единицах измеряется время доступа к жесткому диску, в миллисекундах (тысячных долях секунды) или микросекундах (миллионных долях секунды)? Сколько времени занимает доступ к слову в оперативной памяти вашего компьютера, больше или меньше микросекунды? Сколько инструкций может выполнить центральный про-

цессор вашего компьютера в течение года, если компьютер постоянно держать включенным?

25. [4] Алгоритм сортировки выполняет сортировку 1 000 элементов за 1 секунду. Сколько времени займет сортировка 10 000 элементов,
    - если время исполнения алгоритма прямо пропорционально n^2?
    - если время исполнения алгоритма, по грубым оценкам, пропорционально $n \log n$?

Проекты по реализации

26. [5] Реализуйте два эвристических алгоритма решения задачи коммивояжера из *раздела 1.1*. Какой из них выдает на практике более качественные решения? Можете ли вы предложить эвристический алгоритм, работающий лучше любого из них?
27. [5] Опишите способ проверки достаточности покрытия данным множеством комбинаций номеров лотерейных билетов из задачи в *разделе 1.6*. Напишите программу поиска хороших множеств комбинаций номеров билетов.

Задачи, предлагаемые на собеседовании

28. [5] Напишите функцию деления целых чисел, которая не использует ни оператор деления (/), ни оператор умножения (*). Функция должна быть быстродействующей.
29. [5] У вас есть 25 лошадей. В каждой скачке может участвовать не больше 5 лошадей. Требуется определить первую, вторую и третью по скорости лошадь. Найдите минимальное количество скачек, позволяющих решить эту задачу.
30. [3] Сколько настройщиков пианино во всем мире?
31. [3] Сколько бензоколонок в Соединенных Штатах?
32. [3] Сколько весит лед на хоккейном поле?
33. [3] Сколько километров дорог в Соединенных Штатах?
34. [3] Сколько раз, в среднем, нужно открыть наугад телефонный справочник Манхэттена, чтобы найти определенного человека?

Задачи по программированию

Эти задачи доступны на сайтах **http://www.programming-challenges.com** и **http://uva.onlinejudge.org**. Идентификатор задачи на соответствующем сайте указывается после названия задачи в форме "число/число". Так как сайты англоязычные, названия задач даются на исходном языке.

1. The $3n + 1$ problem. 110101/100.
2. The Trip. 110103/10137.
3. Australian Voting. 110108/10142.

ГЛАВА 2

Анализ алгоритмов

Алгоритмы являются принципиально важным компонентом информатики, т. к. их изучение не требует использования языка программирования или компьютера. Это означает необходимость в методах, позволяющих сравнивать эффективность алгоритмов, не прибегая к их реализации. Самыми значимыми из этих инструментов являются модель вычислений RAM и асимптотический анализ сложности наихудших случаев.

Для оценки производительности алгоритмов применяется асимптотическая нотация. Хотя практик может прийти в ужас от самой идеи теоретического анализа алгоритмов, этот материал представлен здесь в силу его исключительной ценности при работе с алгоритмами.

Этот способ оценки производительности является наиболее трудным материалом в данной книге. Но когда вы поймете основу этих идей на интуитивном уровне, вам будет намного легче разобраться в формальной составляющей.

2.1. Модель вычислений RAM

Разработка машинно-независимых алгоритмов основывается на гипотетическом компьютере, называющемся *машиной с произвольным доступом к памяти* (Random Access Machine) или RAM-машиной. Согласно этой модели наш компьютер работает таким образом:

- для исполнения любой *простой* операции (`+`, `*`, `–`, `=`, `if`, `call`) требуется ровно один временной шаг;
- циклы и подпрограммы не считаются простыми операциями, а состоят из нескольких простых операций. Нет смысла считать подпрограмму сортировки одношаговой операцией, т. к. для сортировки 1 000 000 элементов потребуется определенно намного больше времени, чем для сортировки десяти элементов. Время исполнения цикла или подпрограммы зависит от количества итераций или специфического характера подпрограммы;
- каждое обращение к памяти занимает один временной шаг. Кроме этого, наш компьютер обладает неограниченным объемом оперативной памяти. Кэш и диск в модели RAM не применяются.

Время исполнения алгоритма в RAM-модели вычисляется по общему количеству шагов, требуемых алгоритму для решения данного экземпляра задачи. Допуская, что наша RAM-машина исполняет определенное количество шагов/операций за секунду, количество шагов легко перевести в единицы времени.

Может показаться, что RAM-модель является слишком упрощенным представлением работы компьютеров. В конце концов, на большинстве процессоров умножение двух

чисел занимает больше времени, чем сложение, что не вписывается в первое предположение модели. Второе предположение может быть нарушено удачной оптимизацией цикла компилятором или гиперпотоковыми возможностями процессора. Наконец, время обращения к данным может значительно разниться в зависимости от расположения данных: в кэше, в оперативной памяти или на диске. Таким образом, по сравнению с настоящим компьютером, все три основные допущения для RAM-машины неверны.

Тем не менее, несмотря на эти несоответствия настоящему компьютеру, RAM-модель является *превосходной* моделью для понимания того, как алгоритм будет работать на настоящем компьютере. Она обеспечивает хороший компромисс, отражая поведение компьютеров и одновременно являясь простой в использовании. Эти характеристики делают RAM-модель полезной для практического применения.

Любая модель полезна лишь в определенных рамках. Возьмем, например, модель плоской Земли. Можно спорить, что это неправильная модель, т. к. еще древние греки знали, что в действительности Земля круглая. Но модель плоской Земли достаточно точна для закладки фундамента дома. Более того, в данном случае с моделью плоской Земли настолько удобнее работать, что использование модели сферической Земли[1] для этой цели даже не приходит в голову.

Та же самая ситуация наблюдается и в случае с RAM-моделью вычислений — мы создаем, вообще говоря, очень полезную абстракцию. Довольно трудно создать алгоритм, для которого RAM-модель выдаст существенно неверные результаты. Устойчивость RAM-модели позволяет анализировать алгоритмы машинно-независимым способом.

Подведение итогов

Алгоритмы можно изучать и анализировать, не прибегая к использованию конкретного языка программирования или компьютерной платформы.

2.1.1. Анализ сложности наилучшего, наихудшего и среднего случая

С помощью RAM-модели можно подсчитать количество шагов, требуемых алгоритму для исполнения любого экземпляра задачи. Но чтобы получить общее представление о том, насколько хорошим или плохим является алгоритм, нам нужно знать, как он работает со *всеми* экземплярами задачи.

Чтобы понять, что означает наилучший, наихудший и средний случай сложности алгоритма (т. е. время его исполнения в соответствующем случае), нужно рассмотреть исполнение алгоритма на всех возможных экземплярах входных данных. В случае задачи сортировки множество входных экземпляров состоит из всех возможных компоновок ключей n по всем возможным значениям n. Каждый входной экземпляр можно представить в виде точки графика (рис. 2.1), где ось x представляет размер входа задачи (для сортировки это будет количество элементов, подлежащих сортировке), а ось y — количество шагов, требуемых алгоритму для обработки данного входного экземпляра.

---

[1] В действительности, Земля не совсем сферическая, но модель сферической Земли удобна для работы с такими понятиями, как широта и долгота.