Содержание

Предисловие редактора перевода ... 12

Предисловие ... 14

Педагогические принципы ... 15
Алгоритмы ... 16
Изменения, внесенные во второе издание ... 16
Планирование курса ... 17
Благодарности ... 17

Глава 1
Основы анализа алгоритмов ... 19

Необходимые предварительные знания ... 19
Цели ... 19
Советы по изучению ... 19
1.1. Что такое анализ? ... 20
1.1.1. Классы входных данных ... 24
1.1.2. Сложность по памяти ... 26
1.1.3. Упражнения ... 27
1.2. Что подсчитывать и что учитывать ... 27
1.2.1. Классы входных данных ... 28
1.2.2. Упражнения ... 30
1.3. Необходимые математические сведения ... 31
1.3.1. Логарифмы ... 31
1.3.2. Бинарные деревья ... 32
1.3.3. Вероятности ... 33
1.3.4. Формулы суммирования ... 33
1.3.5. Упражнения ... 35
1.4. Скорости роста ... 36
1.4.1. Классификация скоростей роста ... 38
1.4.2. Упражнения ... 40
1.5. Метод турниров ... 40
1.5.1. Нижние границы ... 41
1.5.2. Упражнения ... 43
1.6. Анализ программ ... 43

Глава 2
Рекурсивные алгоритмы ... 45

Необходимые предварительные знания ... 45
Цели ... 45

Советы по изучению ... 45
2.1. Анализ рекурсивных алгоритмов ... 46
2.1.1. Упражнения ... 49
2.2. Рекуррентные соотношения ... 50
2.2.1. Аппроксимация порядка роста рекуррентных соотношений ... 54
2.2.2. Упражнения ... 55
2.3. Ближайшая пара ... 56
2.3.1. Упражнения ... 61
2.4. Выпуклая оболочка ... 61
2.4.1. Упражнения ... 66
2.5. Генерирование перестановок ... 67
2.5.1. Упражнения ... 69
2.6. Рекурсии и стеки ... 70
2.7. Упражнения по программированию ... 71

Глава 3
Алгоритмы поиска и выборки ... 73

Необходимые предварительные знания ... 73
Цели ... 73
Советы по изучению ... 73
3.1. Последовательный поиск ... 74
3.1.1. Анализ наихудшего случая ... 76
3.1.2. Анализ среднего случая ... 76
3.1.3. Упражнения ... 77
3.2. Двоичный поиск ... 78
3.2.1. Анализ наихудшего случая ... 81
3.2.2. Анализ среднего случая ... 82
3.2.3. Упражнения ... 83
3.3. Выборка ... 85
3.3.1. Упражнения ... 88
3.4. Упражнение по программированию ... 88

Глава 4
Алгоритмы сортировки ... 89

Необходимые предварительные знания ... 89
Цели ... 89
Советы по изучению ... 89
4.1. Сортировка вставками ... 91
4.1.1. Анализ наихудшего случая ... 93

4.1.2. Анализ среднего случая ... 93
4.1.3. Упражнения ... 95
4.2. Пузырьковая сортировка ... 95
4.2.1. Анализ наилучшего случая ... 97
4.2.2. Анализ наихудшего случая ... 98
4.2.3. Анализ среднего случая ... 98
4.2.4. Упражнения ... 99
4.3. Сортировка Шелла ... 101
4.3.1. Анализ алгоритма ... 102
4.3.2. Влияние шага на эффективность ... 104
4.3.3. Упражнения ... 105
4.4. Корневая сортировка ... 105
4.4.1. Анализ ... 107
4.4.2. Упражнения ... 109
4.5. Пирамидальная сортировка ... 109
4.5.1. Анализ наихудшего случая ... 113
4.5.2. Анализ среднего случая ... 116
4.5.3. Упражнения ... 116
4.6. Сортировка слиянием ... 116
4.6.1. Анализ алгоритма MergeLists ... 120
4.6.2. Анализ алгоритма MergeSort ... 121
4.6.3. Упражнения ... 123
4.7. Быстрая сортировка ... 123
4.7.1. Анализ наихудшего случая ... 127
4.7.2. Анализ среднего случая ... 127
4.7.3. Упражнения ... 129
4.8. Внешняя многофазная сортировка слиянием ... 131
4.8.1. Число сравнений при построении отрезков ... 135
4.8.2. Число сравнений при слиянии отрезков ... 135
4.8.3. Число операций чтения блоков ... 136
4.8.4. Упражнения ... 136
4.9. Дополнительные упражнения ... 137
4.10. Упражнения по программированию ... 138

Глава 5
Численные алгоритмы ... 141

Необходимые предварительные знания ... 141
Цели ... 141
Советы по изучению ... 141
5.1. Вычисление значений многочленов ... 143
5.1.1. Схема Горнера ... 144

5.1.2. Предварительная обработка коэффициентов 144
5.1.3. Упражнения 146
5.2. Умножение матриц 147
5.2.1. Умножение матриц по Винограду 148
5.2.2. Умножение матриц по Штрассену 149
5.2.3. Упражнения 151
5.3. Решение линейных уравнений 151
5.3.1. Метод Жордана – Гаусса 152
5.3.2. Упражнения 154

Глава 6
Алгоритмы формальных языков 155

Необходимые предварительные знания 155
Цели 155
Советы по изучению 155
6.1. Основы формального языка 156
6.1.1. Лексикографический порядок 159
6.1.2. Классификация языков 159
6.1.3. Упражнения 164
6.2. Конечные автоматы 165
6.2.1. Регулярные языки 167
6.2.2. Регулярные выражения 168
6.2.3. Регулярные грамматики 169
6.2.4. Детерминированные и недетерминированные конечные автоматы 170
6.2.5. Конвертирование детерминированного конечного автомата в программу 171
6.2.6. Упражнения 173
6.3. Проектирование конечных автоматов 174
6.3.1. Ограничения возможностей конечных автоматов 175
6.3.2. Проектирование конечных автоматов 176
6.3.3. Проектирование регулярной грамматики 179
6.3.4. Упражнения 180
6.4. Эквивалентность и пределы возможностей конечных автоматов ... 181
6.4.1. Возможности конечных автоматов 183
6.4.2. Упражнения 187
6.5. Магазинные автоматы 188
6.5.1. Проектирование детерминированных магазинных автоматов 190
6.5.2. Детерминированные контекстно-свободные языки 192
6.5.3. Недетерминированные магазинные автоматы 193

6.5.4. Упражнения ... 194
6.6. Контекстно-свободные грамматики ... 194
6.6.1. Возможности контекстно-свободной грамматики ... 194
6.6.2. Проектирование контекстно-свободной грамматики ... 195
6.6.3. Преобразование грамматики ... 198
6.6.4. Нормальная форма Грейбаха ... 204
6.6.5. Конвертирование контекстно-свободной грамматики в магазинный автомат ... 206
6.6.6. Упражнения ... 207
6.7. Пределы возможностей магазинных автоматов ... 209
6.7.1. Упражнения ... 213
6.8. Компиляция языков программирования ... 213
6.8.1. Упражнения ... 218

Глава 7
Алгоритмы сравнения с образцом ... 219

Необходимые предварительные знания ... 219
Цели ... 219
Советы по изучению ... 219
7.1. Сравнение строк ... 220
7.1.1. Конечные автоматы ... 222
7.1.2. Алгоритм Кнута – Морриса – Пратта ... 222
7.1.3. Упражнения ... 232
7.2. Приблизительное сравнение строк ... 233
7.2.1. Анализ ... 235
7.2.2. Упражнения ... 235
7.3. Упражнения по программированию ... 236

Глава 8
Алгоритмы на графах ... 237

Необходимые предварительные знания ... 237
Цели ... 237
Советы по изучению ... 237
8.1. Основные понятия теории графов ... 239
8.1.1. Терминология ... 240
8.1.2. Упражнения ... 241
8.2. Структуры данных для представления графов ... 242
8.2.1. Матрица смежности ... 242
8.2.2. Список смежности ... 243
8.2.3. Упражнения ... 244

8.3. Алгоритмы обхода в глубину и по уровням ... 244
8.3.1. Обход в глубину ... 245
8.3.2. Обход по уровням ... 246
8.3.3. Анализ алгоритмов обхода ... 247
8.3.4. Упражнения ... 248
8.4. Алгоритм поиска минимального остовного дерева ... 249
8.4.1. Алгоритм Дейкстры – Прима ... 250
8.4.2. Алгоритм Крускала ... 252
8.4.3. Упражнения ... 254
8.5. Алгоритм поиска кратчайшего пути ... 255
8.5.1. Алгоритм Дейкстры ... 256
8.5.2. Упражнения ... 259
8.6. Алгоритм определения компонент двусвязности ... 260
8.6.1. Упражнения ... 262
8.7. Разбиения множеств ... 263
8.8. Упражнения по программированию ... 266

Глава 9
Параллельные алгоритмы ... 268

Необходимые предварительные знания ... 268
Цели ... 268
Советы по изучению ... 268
9.1. Введение в параллелизм ... 269
9.1.1. Категории компьютерных систем ... 269
9.1.2. Параллельные архитектуры ... 271
9.1.3. Принципы анализа параллельных алгоритмов ... 273
9.1.4. Упражнения ... 273
9.2. Модель PRAM ... 274
9.2.1. Упражнения ... 275
9.3. Простые параллельные операции ... 275
9.3.1. Распределение данных в модели CREW PRAM ... 276
9.3.2. Распределение данных в модели EREW PRAM ... 276
9.3.3. Поиск максимального элемента списка ... 277
9.3.4. Упражнения ... 279
9.4. Параллельный поиск ... 280
9.4.1. Упражнения ... 281
9.5. Параллельная сортировка ... 282
9.5.1. Сортировка на линейных сетях ... 282
9.5.2. Четно-нечетная сортировка перестановками ... 285
9.5.3. Другие параллельные сортировки ... 287
9.5.4. Упражнения ... 287

9.6. Параллельные численные алгоритмы ... 288
9.6.1. Умножение матриц в параллельных сетях ... 288
9.6.2. Умножение матриц в модели CRCW PRAM ... 292
9.6.3. Решение систем линейных уравнений алгоритмом SIMD ... 293
9.6.4. Упражнения ... 294
9.7. Параллельные алгоритмы на графах ... 294
9.7.1. Параллельный алгоритм поиска кратчайшего пути ... 295
9.7.2. Параллельный алгоритм поиска минимального остовного дерева ... 296
9.7.3. Упражнения ... 298

Глава 10
Вычислимое и невычислимое ... 300

Необходимые предварительные знания ... 300
Цели ... 300
Советы по изучению ... 300
10.1. Машина Тьюринга ... 302
10.1.1. Машина Тьюринга как акцептор языка ... 304
10.1.2. Машина Тьюринга как вычислитель функций ... 306
10.1.3. Проектирование машин Тьюринга ... 308
10.1.4. Упражнения ... 311
10.2. Тезис Чёрча–Тьюринга ... 312
10.3. Разновидности машины Тьюринга ... 314
10.3.1. Машина Тьюринга с бесконечной в обе стороны лентой ... 314
10.3.2. Машины Тьюринга с несколькими лентами ... 316
10.3.3. Двумерные машины Тьюринга ... 318
10.3.4. Недетерминированные машины Тьюринга ... 320
10.3.5. Универсальная машина Тьюринга ... 323
10.3.6. Упражнения ... 327
10.4. Возможности машины Тьюринга ... 328
10.4.1. Языки, не относящиеся к рекурсивно перечислимым ... 328
10.4.2. Проблема остановки ... 330
10.4.3. Упражнения ... 333
10.5. Что такое NP? ... 334
10.5.1. Сведение задачи к другой задаче ... 336
10.5.2. NP-полные задачи ... 337
10.6. Типичные NP задачи ... 338
10.6.1. Раскраска графа ... 339
10.6.2. Раскладка по ящикам ... 340
10.6.3. Упаковка рюкзака ... 340
10.6.4. Задача о суммах элементов подмножеств ... 340

10.6.5. Задача об истинности КНФ выражения 340
10.6.6. Задача планирования работ 341
10.6.7. Упражнения 341
10.7. Какие задачи относятся к классу NP? 342
10.7.1. Выполнено ли равенство P = NP? 343
10.7.2. Упражнения 344
10.8. Проверка возможных решений 344
10.8.1. Задача о планировании работ 345
10.8.2. Раскраска графа 346
10.8.3. Упражнения 347

Глава 11
Другие алгоритмические инструменты 348

Необходимые предварительные знания 348
Цели 348
Советы по изучению 348
11.1. Жадные приближенные алгоритмы 349
11.1.1. Приближения в задаче о коммивояжере 350
11.1.2. Приближения в задаче о раскладке по ящикам 352
11.1.3. Приближения в задаче об упаковке рюкзака 353
11.1.4. Приближения в задаче о сумме элементов подмножества .. 353
11.1.5. Приближения в задаче о раскраске графа 356
11.1.6. Упражнения 356
11.2. Алгоритм с возвратом 358
11.2.1. Упражнения 361
11.3. Алгоритм ветвей и границ 362
11.3.1. Упражнения 366
11.4. Вероятностные алгоритмы 366
11.4.1. Численные вероятностные алгоритмы 367
11.4.2. Алгоритмы Монте-Карло 370
11.4.3. Алгоритмы Лас-Вегаса 372
11.4.4. Шервудские алгоритмы 375
11.4.5. Сравнение вероятностных алгоритмов 376
11.4.6. Упражнения 376
11.5. Динамическое программирование 377
11.5.1. Вычисление чисел Фибоначчи и биномиальных коэффициентов 378
11.5.2. Динамическое умножение матриц 380
11.5.3. Алгоритм Флойда нахождения кратчайших расстояний между всеми вершинами нагруженного ориентированного графа 382

11.5.4. Динамический алгоритм, решающий задачу об укладке рюкзака 385
11.5.5. Упражнения 387
11.6. Упражнения по программированию 388

Приложение А
Таблица случайных чисел 390

Приложение Б
Генерирование псевдослучайных чисел 392

Б.1. Случайная последовательность в произвольном интервале 393
Б.2. Пример применения 394
Б.2.1. Первый способ 394
Б.2.2. Второй способ 394
Б.2.3. Третий способ 395

Приложение В
Ответы к упражнениям 396

Приложение Г
Литература 408

Предметный указатель 413

Предисловие редактора перевода

Алгоритмы и программирование все чаще встречаются на пути каждого из нас и все глубже входят в повседневную жизнь. Банковские операции, бронирование билетов, организация систем связи, нефте- и газодобыча, торговля — невозможно перечислить все те области человеческой деятельности, которые уже немыслимы без применения компьютеров. Неизбежно растет и число людей, призванных разрабатывать и реализовывать алгоритмы.

Круг задач, допускающих алгоритмическое решение, довольно широк. Однако первое приходящее на ум решение подобной задачи обычно удручающе неэффективно (и зачастую трудореализуемо). Несмотря на постоянный рост производительности вычислительных систем, вопросы эффективности продолжают играть принципиальную роль. Объемы данных, подлежащих обработке, растут с неменьшей скоростью и если мы не хотим, скажем, ждать годами получения денег в банкомате по кредитной карте, то разработчики банкомата должны тщательно продумать алгоритмы его работы. К тому же эффективные решения обычно красивее, точнее и строже, их реализации изначально содержат меньше ошибок, а круг их применения заметно шире.

Как строить эффективные алгоритмы? Первый шаг в этом направлении, разумеется, — по возможности более строгая формулировка решаемой задачи, отделение существенного от неважного. А вот на втором этапе мы уже пытаемся либо самостоятельно улучшить алгоритм, пришедший на ум первым, либо воспользоваться уже известными эффективными алгоритмами, придуманными другими людьми.

В предлагаемой вниманию читателей книге Дж. МакКоннелла как раз и описан «джентльменский набор» базовых алгоритмов решения разнообразных задач (от сортировки и сравнения с образцом до вычислительных), находящих широкое применение. Книга основана на семестровом курсе лекций, неоднократно читавшемся автором в Канизиус Колледже и может составить основу подобного вводного курса в наших вузах. Предполагаемый ею объем предварительных знаний читателя минимален — он не превышает содержания нашего школьного курса информатики. Для того, чтобы приступить к ее изучению, достаточно владеть элементарными навыками программирования и уметь пользоваться основными типами данных — списками, деревьями, стеками, очередями.

По подходу к изложению материала книга занимает золотую середину спектра, на одном конце которого руководства по использованию языков программирования и прочих конкретных программных систем,

на другом — сугубо теоретические курсы, задействованный в которых мощный математический аппарат делает их недоступными для широкого круга читателей. Впрочем, это ее свойство порождает и некоторые недостатки: перенос описываемых в ней алгоритмов на компьютер требует дополнительной работы, а проводимый анализ алгоритмов страдает нестрогостью. Последнюю особенность я постарался несколько смягчить при переводе.

Книга знакомит читателя с широким кругом вопросов. В ней обсуждаются:

- алгоритмы сортировки и поиска, на которых основана работа современных систем управления базами данных;
- алгоритмы сравнения с образцом, лежащие в основе текстовых редакторов, компиляторов и программ синтаксического анализа текстов;
- анализ графов, применяемый в разработке транспортных и компьютерных сетей и систем связи;
- находящие все более разнообразное применение параллельные алгоритмы;
- вероятностные алгоритмы и доставляемые ими преимущества.

Кроме того, небольшие главы посвящены элементарным численным алгоритмам и вопросам теории сложности — разбиению задач на классы по эффективности решающих их алгоритмов. Изложение сопровождается большим количеством несложных примеров и задач, позволяющих читателю самостоятельно наработать технику и самому оценить уровень своих знаний.

Можно надеяться, что эта книга поможет читателю овладеть основами анализа алгоритмов и побудит его к углублению своих знаний.

Моей семье и друзьям
за любовь и поддержку

Предисловие

У этой книги три основные задачи: повысить мастерство разработки алгоритмов, усилить понимание влияния алгоритмов на эффективность программы и развить профессионализм, необходимый для анализа любых алгоритмов, используемых в программах. Глядя на некоторые современные коммерческие программные продукты, понимаешь, что некоторые их создатели не обращают внимания ни на временну́ю эффективность программ, ни на разумное использование памяти. Они полагают, что если программа занимает слишком много места, то пользователь приобретет дополнительную память, если она слишком долго работает, то он может купить более быстрый компьютер.

Однако скорость компьютеров не может увеличиваться бесконечно. Она ограничена скоростью перемещения электронов по проводам, скоростью распространения света по оптическим кабелям и скоростью коммутации каналов связи компьютеров, участвующих в вычислениях. Другие ограничения связаны не с производительностью компьютеров, а непосредственно со сложностью решаемой задачи. Есть задачи, для решения которых не хватит человеческой жизни, даже если при этом будут использованы самые оптимальные из известных алгоритмов. А поскольку среди этих задач есть и важные, необходимы алгоритмы получения приблизительных ответов.

В начале 80-х годов архитектура компьютеров серьезно ограничивала их скорость и объем памяти. Зачастую общий размер программ и данных не превышал 64 К, в то время как современные персональные компьютеры способны оперировать данными, в более чем 16 000 раз превышающими это количество. Нынешнее программное обеспечение гораздо сложнее, чем в 1980 году, и компьютеры стали заметно лучше, но это не повод, чтобы игнорировать вопросы эффективности программ при их разработке. В спецификации некоторых проектов включены ограничения на время выполнения и использование памяти конечным продуктом, которые могут побудить программистов экономить память и увеличивать скорость выполнения.

Небольшие размеры карманных компьютеров, мобильных телефонов и других встроенных систем также накладывают ограничения на размеры и скорость выполнения программ.

Педагогические принципы

Что я слышу, я забываю.
Что я вижу, я запоминаю.
Что я делаю, я знаю.
Конфуций

Книгу можно изучать либо самостоятельно, либо в небольших группах. Для этого главы сделаны независимыми и понятными; такую главу стоит прочитать перед встречей группы. Каждой главе предпосланы указания по ее изучению. Во многих главах приведены дополнительные данные, что позволяет читателю выполнять алгоритмы вручную, чтобы лучше их понять. Результаты применения алгоритмов к этим дополнительным данным приведены в приложении В. Каждый параграф снабжен упражнениями, от простых — на трассировку алгоритмов, до более сложных, требующих доказательств. Читатель должен научиться выполнять упражнения каждого параграфа. При чтении курса лекций на основе этой книги упражнения можно давать в качестве домашнего задания или использовать в аудитории как для индивидуальной работы студентов, так и для обсуждения в небольших группах. Помимо настоящего учебника, существует руководство для преподавателя, в котором содержатся решения упражнений и методические указания по обучению материалу при активном групповом изучении. Оно доступно в интернете. В главы 2, 3, 4, 7, 8 и 11 включены упражнения по программированию. Программные проекты позволяют читателю превратить алгоритмы этих глав в программы и протестировать их, а затем сравнить результаты работы реальных программ с полученными посредством теоретического анализа.

Активное обучение основано на том принципе, что студенты и учатся лучше, и дольше помнят усвоенную информацию, если являются непосредственными участниками процесса обучения. Чтобы добиться этого, обучаемым необходимо предоставить шанс узнать существенно больше, чем профессор им сообщает в течение лекции. Для достижения поставленной цели в курсе алгоритмов профессору стоит прочитать короткую вводную лекцию по новому материалу, после чего предоставить студентам возможность самостоятельно решать задачи, отвечая на возникающие вопросы.

Совместная проработка материала дает участникам процесса обучения возможность ответить на простые вопросы, всплывающие у их товарищей, оставляя профессорскому вниманию лишь сложные вопросы, возникающие у всей группы. При этом студент может *своевременно* задавать вопросы, волнующие его в данный момент, что ведет к ясному

пониманию предмета. Важно то, что наблюдая за работой студентов, профессор сможет легко убедиться, что его правильно поняли. Кроме того, такой способ обучения позволяет профессору своевременно выявлять неправильно понятые тезисы и приемы и исправлять их во время регулярной работы студентов над упражнениями.

В качестве помощи студентам в овладении материалом каждая глава содержит список начальных сведений, необходимых для ее успешного изучения, цели, которые ставит данная глава, и советы по наиболее успешному достижению поставленных целей.

Алгоритмы

Анализ алгоритмов не зависит от компьютера или используемого языка программирования, поэтому алгоритмы в книге записываются в псевдокоде. Эти алгоритмы может прочесть любой, кому знакомо понятие условного выражения (IF или CASE/SWITCH), цикла (FOR или WHILE) и рекурсии.

Изменения, внесенные во второе издание

Наиболее значительные изменения были внесены во второе издание книги с целью привести ее в соответствие с программными требованиями ACM[1] к курсу CS 210 — «Анализ и разработка алгоритмов». Теория автоматов была пополнена обсуждениями конечных автоматов, регулярных и контекстно-свободных языков, автоматов с магазинной памятью, синтаксического разбора и компилирования программ, машин Тьюринга, тезисов Черча–Тьюринга и задачи об остановке. Перебор с возвратом, ветвление и предельное значение, алгоритм Флойда были добавлены в главу 11. Туда же был включен пятый динамический алгоритм, дающий приближенное решение задачи о рюкзаке или укладке рюкзака.

Остальные дополнения к тексту предназначены для облегчения читателю понимания материала первого издания. Наиболее существенное изменение — перемещение рекурсивных алгоритмов из главы 1 в новую главу, которая разъясняет часть старого материала и содержит обсуждение приближения последовательности рекуррентных соотношений, а также новый материал о поиске ближайшей пары точек, выпуклой оболочке множества точек и генерировании перестановок множества чисел. Нако-

[1] ACM — Association for Computing Machinery (Ассоциация по вычислительной технике). — *Прим. перев.*

нец, в главу о поиске и хранении была добавлена трассировка выполнения алгоритма.

Планирование курса

Ниже приведен один из возможных графиков освоения материала в односеместровом курсе:

глава 1	1 неделя
глава 2	1 неделя
глава 3	1 неделя
глава 4	2 недели
глава 5	1 неделя
глава 6	2 недели
глава 7	1 неделя
глава 8	1 неделя
глава 9	1 неделя
глава 10	2 недели
глава 11	1 неделя

Изучение глав 3, 5 и 7 скорее всего не потребует полной недели; оставшееся время можно использовать для введения в курс, знакомства с активным коллективным обучением и для часового экзамена. При наличии у студентов соответствующей подготовки первая глава также может быть пройдена быстрее. При желании главу 9 можно исключить из курса.

Благодарности

Я хотел бы поблагодарить всех тех, кто помогал мне при подготовке этой книги. Прежде всего, студентов моего курса «Автоматы и алгоритмы» за комментарии к ее первым версиям, служившим основой курса многие годы.

Нашлось довольно много рецензентов первого издания, которые оказались настолько любезны, что предложили полезные изменения для второго издания и указали на опечатки. Слушатели моего курса «Автоматы и алгоритмы» внесли многочисленные предложения по облегчению восприятия учебника. Я благодарен всем, кто прислал мне исправления текста первого издания, в особенности Хангу Лау из университета МакГилл и Паулю Мьюир из университета Санта Марии, с кем я находился в интенсивной переписке по электронной почте.

Рецензии первого издания из издательства «Джонс и Бартлетт», полученные в 2000 г., были очень полезны. В результате текст стал яснее и появилось несколько дополнений.

Я хотел бы также поблагодарить Дугласа Кэмпбелла (университет Брайэм Янг), Нэнси Киннерсли (университет Канзаса) и Кирка Пруса (университет Питтсбурга) за комментарии.

Я хотел бы поблагодарить сотрудников издательства «Джонс и Бартлетт» — редакторов Эми Роз и Майкла Штранца, а также технического редактора Тару МакКормик за работу над этой книгой. Также хочу выразить признательность их ассистентам Лауре Паглуиса и Саре Бейли. Особенно я благодарен Эми за помощь в знакомстве с изданием «Джонс и Бартлетт», что в конечном счете привело меня в число авторов этого издания. Я очень высоко ценю ее усилия. Кроме того, я хотел бы выразить признательность техническому редактору Сюзане Флешман и корректору Тару Крэмер. Все оставшиеся в тексте ошибки исключительно на совести автора.

И наконец, я признателен Фреду Дансеро за поддержку и советы на многих этапах работы над книгой и Барни (1992–2005) за ту замечательную возможность отвлечься, которую может дать только собака.

ГЛАВА 1

ОСНОВЫ АНАЛИЗА АЛГОРИТМОВ

Необходимые предварительные знания

Приступая к чтению этой главы, вы должны уметь:

– читать и разрабатывать алгоритмы;
– опознавать операции сравнения и арифметические операции;
– пользоваться элементарной алгеброй.

Цели

Освоив эту главу, вы должны уметь:

– описывать анализ алгоритма;
– объяснять принципы выбора подсчитываемых операций;
– проводить анализ в наилучшем, наихудшем и среднем случаях;
– работать с логарифмами, вероятностями и суммированием;
– описывать функции $\theta(f)$, $\Omega(f)$, $O(f)$, скорость роста и порядок алгоритма;
– использовать дерево решений для определения нижней границы сложности;
– выводить формулу общего члена последовательности из простых рекуррентных соотношений.

Советы по изучению

Изучая эту главу, самостоятельно проработайте все приведенные примеры и убедитесь, что вы их поняли. Кроме того, прежде чем читать ответ на вопрос, следует попробовать ответить на него самостоятельно. Подсказка или ответ следуют непосредственно после вопроса.

Одну и ту же задачу может решать множество алгоритмов. Эффективность работы каждого из них описывается разнообразными характеристиками. Прежде чем анализировать эффективность алгоритма, нужно доказать, что данный алгоритм правильно решает задачу. В противном случае вопрос об эффективности не имеет смысла. Если алгоритм решает поставленную задачу, то мы можем посмотреть, насколько это решение эффективно. Данная глава закладывает основу для анализа и сравнения достаточно сложных алгоритмов, с которыми мы познакомимся позднее.

При анализе алгоритма определяется количество «времени», необходимое для его выполнения. Это не реальное число секунд или других временны́х единиц, а приблизительное число операций, выполняемых алгоритмом. Число операций и измеряет относительное время выполнения алгоритма. Таким образом, иногда мы будем называть «временем» вычислительную сложность алгоритма. Фактическое количество секунд, требуемое для выполнения алгоритма на компьютере, непригодно для анализа, поскольку нас интересует только относительная эффективность алгоритма, решающего конкретную задачу. Вы также увидите, что и вправду время, требуемое на решение задачи, — не очень хороший способ измерять эффективность алгоритма, потому что алгоритм не становится лучше, если его перенести на более быстрый компьютер, или хуже, если его исполнять на более медленном.

На самом деле фактическое количество операций алгоритма на тех или иных входных данных не представляет большого интереса и не очень много сообщает об алгоритме. Вместо этого нас будет интересовать зависимость числа операций конкретного алгоритма от размера входных данных. Мы можем сравнить два алгоритма по скорости роста числа операций. Именно скорость роста играет ключевую роль, поскольку при небольшом размере входных данных алгоритм A может требовать меньшего количества операций, чем алгоритм B, но при росте объема входных данных ситуация может поменяться на противоположную.

Мы начнем эту главу с описания того, что же такое анализ и зачем он нужен. Затем мы очертим круг рассматриваемых операций и параметров, по которым будет производиться анализ. Поскольку для нашего анализа необходима математика, несколько следующих разделов будут посвящены важным математическим понятиям и свойствам, используемым при анализе итеративных и рекурсивных алгоритмов.

1.1. Что такое анализ?

Анализируя алгоритм, можно получить представление о том, сколько времени займет решение данной задачи при помощи данного алгоритма. Для каждого рассматриваемого алгоритма мы оценим, насколько быстро решается задача на массиве входных данных длины N. Например, мы можем оценить, сколько сравнений потребует алгоритм сортировки при упорядочении списка из N величин по возрастанию, или подсчитать, сколько арифметических операций нужно для умножения двух матриц размером $N \times N$.

Одну и ту же задачу можно решить с помощью различных алгоритмов, и анализ алгоритмов дает нам инструмент для выбора оптимального.

Рассмотрим, например, два алгоритма, в которых выбирается наибольшая из четырех величин:

```
largest = a
if b > largest then
   largest = b
end if
return a
if c > largest then
   largest = c
end if
if d > largest then
   largest = d
end if
return largest
```

```
if a > b then
   if a > c then
      if a > d then
         return a
      else
         return d
      end if
   else
      if c > d then
         return c
      else
         return d
      end if
   end if
else
   if b > c then
      if b > d then
         return b
      else
          return d
      end if
   else
      if c > d then
          return c
      else
          return d
      end if
   end if
end if
```

При сопоставлении этих алгоритмов видно, что в каждом делается три сравнения. Первый алгоритм легче прочесть и понять, но с точки зрения выполнения на компьютере у них одинаковый уровень сложности. По временны́м затратам эти алгоритмы одинаковы, но первый требует больше памяти из-за вре́менной переменной с именем `largest`. Это дополнительное место не играет роли, если сравниваются числа или символы, но при работе с другими типами данных оно может стать существенным. Многие современные языки программирования позволяют определить операторы

сравнения для больших и сложных объектов или записей. В этих случаях размещение вре́менной переменной может потребовать много места. При анализе эффективности алгоритмов нас будет в первую очередь интересовать вопрос времени, но в тех случаях, когда память играет существенную роль, мы будем обсуждать и ее.

Мы вводим понятия «время выполнения» и «объем памяти» с целью сравнения эффективности двух разных алгоритмов, решающих одну задачу. Поэтому мы никогда не будем сравнивать между собой алгоритм сортировки и алгоритм умножения матриц, а будем сравнивать друг с другом два разных алгоритма сортировки.

Результат анализа алгоритмов — не формула для точного количества секунд или компьютерных циклов, которые потребует конкретный алгоритм. Такая информация бесполезна, так как в этом случае нужно указывать также тип компьютера, используется ли он одним пользователем или несколькими, какой у него процессор и тактовая частота, полный или редуцированный набор команд на чипе процессора и насколько хорошо компилятор оптимизирует выполняемый код. Эти условия влияют на скорость работы программы, реализующей алгоритм. Учет этих условий означал бы, что при переносе программы на более быстрый компьютер алгоритм становится лучше, так как он работает быстрее. Но это не так, и поэтому наш анализ не учитывает особенностей компьютера.

В случае небольшой или простой программы количество выполненных операций как функцию от размера входных данных N можно посчитать точно. Однако в большинстве случаев в этом нет нужды. В § 1.4 показано, что разница между алгоритмом, который делает $N + 5$ операций, и тем, который делает $N + 250$ операций, становится незаметной, как только N становится очень большим. Тем не менее, мы начнем анализ алгоритмов с подсчета точного количества операций.

Еще одна причина, по которой мы не будем пытаться подсчитать все операции, выполняемые алгоритмом, состоит в том, что даже самая аккуратная его настройка может привести лишь к незначительному улучшению производительности. Рассмотрим, например, алгоритм подсчета числа вхождений различных символов в файл. Алгоритм для решения такой задачи мог бы выглядеть примерно так:

```
for all 256 символов do
   обнулить счетчик
end for
while в файле еще остались символы do
   ввести очередной символ
   увеличить счетчик вхождений прочитанного символа на единицу
end while
```

Посмотрим на этот алгоритм. Он делает 256 проходов в цикле инициализации. Если во входном файле N символов, то во втором цикле N проходов. Возникает вопрос: «Что же считать?». В цикле `for` сначала инициализируется переменная цикла, затем в каждом проходе проверяется, что переменная не выходит за границы выполнения цикла, и переменная получает приращение. Это означает, что цикл инициализации делает 257 присваиваний (одно для переменной цикла и 256 для счетчика), 256 приращений переменной цикла и 257 проверок того, что эта переменная находится в пределах границ цикла (одна дополнительная для остановки цикла). Во втором цикле нужно сделать $N + 1$ раз проверку условия (+1 для последней проверки, когда файл пуст) и N приращений счетчика. Всего операций:

Приращения	$N + 256$
Присваивания	257
Проверки условий	$N + 258$

Таким образом, при размере входного файла в 500 символов в алгоритме делается 1771 операция, из которых 770 приходится на инициализацию (43%). Теперь посмотрим, что происходит при увеличении величины N. Если файл содержит 50 000 символов, то алгоритм сделает 100 771 операцию, из которых только 770 связаны с инициализацией (что составляет менее 1% общего числа операций). Число операций инициализации не изменилось, но в процентном отношении при увеличении N их становится значительно меньше.

Теперь посмотрим с другой стороны. Данные в компьютере организованы таким образом, что копирование больших блоков данных происходит с той же скоростью, что и присваивание. Мы могли бы сначала присвоить 16 счетчикам начальное значение, равное 0, а затем скопировать этот блок 15 раз, чтобы заполнить остальные счетчики. Это приведет к тому, что в циклах инициализации число проверок уменьшится до 33, присваиваний — до 33 и приращений — до 31. Количество операций инициализации уменьшается с 770 до 97, то есть на 87%. Если же мы сравним полученный выигрыш с числом операций по обработке файла в 50 000 символов, экономия составит менее 0,7% (вместо 100 771 операций мы затратим 100 098). Заметим, что экономия по времени могла бы быть даже больше, если бы мы сделали все эти инициализации без циклов, поскольку понадобилось бы только 31 простое присваивание, но этот дополнительный выигрыш дает лишь 0,07% экономии. Овчинка не стоит выделки.

Мы видим, что вес инициализации по отношению ко времени выполнения алгоритма незначителен. В терминах анализа при увеличении объема входных данных стоимость инициализации становится пренебрежимо малой.

В главе 10 мы познакомимся с абстрактной машиной, называемой машиной Тьюринга, которую обычно рассматривают как эквивалент алгоритма. В самой ранней работе по анализу алгоритмов определена *вычислимость* алгоритма на машине Тьюринга. При анализе подсчитывается число переходов, необходимое для решения задачи. Анализ пространственных потребностей алгоритма подразумевает подсчет числа ячеек в ленте машины Тьюринга, необходимых для решения задачи. Такого рода анализ разумен, и он позволяет правильно определить относительную скорость двух алгоритмов, однако его практическое осуществление чрезвычайно трудно и занимает много времени. Сначала нужно строго описать процесс выполнения функций перехода в машине Тьюринга, выполняющей алгоритм, а затем подсчитать время выполнения — весьма утомительная процедура.

Другой, не менее осмысленный способ анализа алгоритмов предполагает, что алгоритм записан на каком-либо языке высокого уровня вроде C, C++, Java или достаточно общем псевдокоде. Особенности псевдокода не играют существенной роли, если он реализует основные структуры управления, общие для всех алгоритмов. Такие структуры, как циклы вида `for` или `while`, механизм ветвления вида `if`, `case` или `switch`, присутствуют в любом языке программирования высокого уровня, и любой такой язык подойдет для наших целей. Всякий раз нам предстоит рассматривать один конкретный алгоритм — в нем редко будет задействовано больше одной функции или фрагмента программы, и поэтому мощность упомянутых выше языков вообще не играет никакой роли. Вот мы и будем пользоваться псевдокодом.

В некоторых языках программирования значения булевых выражений вычисляются сокращенным образом. Это означает, что член `B` в выражении `A and B` вычисляется, только если выражение `A` истинно, поскольку в противном случае результат оказывается ложным независимо от значения `B`. Аналогично член `B` в выражении `A or B` не будет вычисляться, если выражение `A` истинно. Как мы увидим, неважно, считать ли число сравнений при проверке сложного условия равным 1 или 2. Поэтому, освоив эту главу, мы перестанем обращать внимание на сокращенные вычисления булевых выражений.

1.1.1. Классы входных данных

Роль входных данных в анализе алгоритмов чрезвычайно велика, поскольку последовательность действий алгоритма определяется не в последнюю очередь входными данными. Например, для того, чтобы найти наибольший элемент в списке из N элементов, можно воспользоваться следующим алгоритмом:

```
largest = list [1]
for i = 2 to N do
   if (list [i] > largest) then
      largest = list[i]
   end if
end for
```

Ясно, что если список упорядочен в порядке убывания, то перед началом цикла будет сделано одно присваивание, а в теле цикла присваиваний не будет. Если список упорядочен по возрастанию, то всего будет сделано N присваиваний (одно перед началом цикла и $N-1$ в цикле). При анализе мы должны рассмотреть различные возможные множества входных значений, поскольку если мы ограничимся одним множеством, оно может оказаться тем самым, на котором решение самое быстрое (или самое медленное). В результате мы получим ложное представление об алгоритме. Вместо этого мы рассматриваем все типы входных множеств.

Мы попытаемся разбить различные входные множества на классы в зависимости от поведения алгоритма на каждом множестве. Такое разбиение позволяет уменьшить количество рассматриваемых ситуаций. Например, число различных расстановок 10 различных чисел в списке есть $10! = 3\,628\,800$. Применим к списку из 10 чисел алгоритм поиска наибольшего элемента. Имеется 362 880 входных множеств, у которых первое число является наибольшим; их все можно поместить в один класс. Если наибольшее по величине число стоит на втором месте, то алгоритм сделает ровно два присваивания. Множеств, в которых наибольшее по величине число стоит на втором месте, 362 880. Их можно отнести к другому классу. Видно, как будет меняться число присваиваний при изменении положения наибольшего числа от 1 до N. Таким образом, мы должны разбить все входные множества на N разных классов по числу сделанных присваиваний. Как видите, нет необходимости выписывать или описывать детально все множества, помещенные в каждый класс. Нужно знать лишь количество классов и объем работы на каждом множестве класса.

Число возможных наборов входных данных может стать очень большим при увеличении N. Например, 10 различных чисел можно расположить в списке 3 628 800 способами. Невозможно рассмотреть все эти способы. Вместо этого мы разбиваем списки на классы в зависимости от того, что будет делать алгоритм. Для вышеуказанного алгоритма разбиение основывается на местоположении наибольшего значения. В результате получается 10 разных классов. Для другого алгоритма, например алгоритма поиска наибольшего и наименьшего значения, наше разбиение могло бы основываться на том, где располагаются наибольшее и наимень-

шее значения. В таком разбиении 90 классов. Как только мы выделили классы, мы можем посмотреть на поведение алгоритма на одном множестве из каждого класса. Если классы выбраны правильно, то на всех множествах входных данных одного класса алгоритм производит одинаковое количество операций, а на множествах из другого класса это количество операций скорее всего будет другим.

1.1.2. Сложность по памяти

Мы будем обсуждать в основном сложность алгоритмов по времени, однако кое-что можно сказать и про то, сколько памяти нужно тому или иному алгоритму для выполнения работы. На ранних этапах развития компьютеров при ограниченных объемах компьютерной памяти (как внешней, так и внутренней) этот анализ носил принципиальный характер. Все алгоритмы разделяются на такие, которым достаточно ограниченной памяти, и те, которым нужно дополнительное пространство. Нередко программистам приходилось выбирать более медленный алгоритм просто потому, что он обходился имеющейся памятью и не требовал внешних устройств.

Спрос на компьютерную память был очень велик, поэтому изучался и вопрос, какие данные будут сохраняться, а также эффективные способы такого сохранения. Предположим, например, что мы записываем вещественное число из интервала от -10 до $+10$, имеющее один десятичный знак после запятой. При записи этого числа в вещественном виде большинство компьютеров потратит от 4 до 8 байтов памяти, однако если предварительно умножить число на 10, то мы получим целое число из интервала от -100 до $+100$, а для его хранения выделяется всего один байт. По сравнению с первым вариантом экономия составляет от 3 до 7 байтов. Программа, в которой сохраняется 1000 таких чисел, экономит от 3000 до 7000 байтов. Если принять во внимание, что еще недавно — в начале 80-х годов прошлого века — у компьютеров была память объемом лишь 65 536 байтов, экономия получается существенной. Именно эта необходимость экономить память наряду с долголетием компьютерных программ привела к проблеме 2000-го года. Если ваша программа использует много различных дат, то половину места для записи года можно сэкономить, сохраняя значение 99 вместо 1999. Да и авторы программ не предполагали в 80-х годах, что их продукция доживет до 2000-го года.

При взгляде на программные продукты, предлагаемые на рынке сегодня, ясно, что подобный анализ памяти проведен не был. Объем памяти, необходимый даже для простых программ, исчисляется мегабайтами или даже гигабайтами. Программисты, похоже, не ощущают потребности в экономии места, полагая, что если у пользователя недостаточно памяти для выполнения программы, то он пойдет и купит недостающую или

новый жесткий диск. В результате компьютеры приходят в негодность задолго до того, как они действительно устаревают.

Некоторую новую ноту внесло распространение миниатюрных электронных устройств например карманных компьютеров и мобильных телефонов. У типичного такого устройства ограничено пространство как для хранения данных, так и для программного обеспечения. И поэтому разработка маленьких программ, обеспечивающих компактное хранение данных, вновь становится актуальной.

1.1.3. Упражнения

1. Напишите на псевдокоде алгоритм, подсчитывающий количество прописных букв в текстовом файле. Сколько сравнений требуется этому алгоритму? Каково максимально возможное значение числа операций увеличения счетчика? Минимальное такое число? (Выразите ответ через число N символов во входном файле.)
2. В файле записано несколько чисел, однако мы не знаем, сколько. Напишите на псевдокоде алгоритм для подсчета среднего значения чисел в файле. Какого типа операции делает ваш алгоритм? Сколько операций каждого типа он делает?
3. Напишите алгоритм, не использующий сложных условий, который по трем введенным целым числам определяет, различны ли они все между собой. Сколько сравнений в среднем делает ваш алгоритм? Не забудьте исследовать все классы входных данных.
4. Напишите алгоритм, который получает на входе три целых числа, и находит наибольшее из них. Каковы возможные классы входных данных? На каком из них ваш алгоритм делает наибольшее число сравнений? На каком меньше всего? (Если разницы между наилучшим и наихудшим классами нет, перепишите свой алгоритм с простыми сравнениями так, чтобы он не использовал вре́менных переменных и чтобы в наилучшем случае он работал быстрее, чем в наихудшем.)
5. Напишите алгоритм для поиска второго по величине элемента в списке из N значений. Сколько сравнений делает ваш алгоритм в наихудшем случае? (Позднее мы обсудим алгоритм, которому требуется около N сравнений.)

1.2. Что подсчитывать и что учитывать

Решение вопроса о том, что считать, состоит из двух шагов. На первом шаге выбирается значимая операция или группа операций, а на втором —

какие из этих операций содержатся в теле алгоритма, а какие составляют накладные расходы или уходят на регистрацию и учет данных.

В качестве значимых обычно выступают операции двух типов: сравнение и арифметические операции. Все операторы сравнения считаются эквивалентными, и их учитывают в алгоритмах поиска и сортировки. Важным элементом таких алгоритмов является *сравнение* двух величин для определения (при поиске) того, совпадает ли данная величина с искомой, а при сортировке — вышла ли она за пределы данного интервала. Операторы сравнения проверяют, равна или не равна одна величина другой, меньше она или больше, меньше или равна, больше или равна.

Мы разбиваем арифметические операции на две группы: аддитивные и мультипликативные. *Аддитивные операции* (называемые для краткости *сложениями*) включают в себя сложение, вычитание, увеличение и уменьшение счетчика. *Мультипликативные операции* (или короче *умножения*) включают в себя умножение, деление и взятие остатка по модулю. Разбиение на эти две группы связано с тем, что умножения работают дольше, чем сложения. На практике некоторые алгоритмы считаются предпочтительнее других, если в них меньше умножений, даже если число сложений при этом пропорционально возрастает. За пределами нашей книги остались алгоритмы, использующие логарифмы и тригонометрические функции, которые образуют еще одну, даже более времяемкую, чем умножения, группу операций (обычно компьютеры вычисляют их значения с помощью разложений в ряд).

Целочисленное умножение или *деление на степень двойки* образуют специальный случай. Эта операция сводится к сдвигу, а последний по скорости эквивалентен сложению. Однако случаев, где эта разница существенна, совсем немного, поскольку умножение и деление на 2 встречаются в первую очередь в алгоритмах типа «разделяй и властвуй», где значимую роль зачастую играют операторы сравнения.

1.2.1. Классы входных данных

При анализе алгоритма выбор входных данных может существенно повлиять на его выполнение. Скажем, некоторые алгоритмы сортировки могут работать очень быстро, если входной список уже отсортирован, тогда как другие алгоритмы покажут весьма скромный результат на таком списке. А вот на случайном списке результат может оказаться противоположным. Поэтому мы не будем ограничиваться анализом поведения алгоритмов на одном входном наборе данных. Практически мы будем искать такие данные, которые обеспечивают как самое быстрое, так и самое медленное выполнение алгоритма. Кроме того, мы будем оценивать и среднюю эффективность алгоритма на всех возможных наборах данных.

Наилучший случай

Как показывает название раздела, наилучшим случаем для алгоритма является такой набор данных, на котором алгоритм выполняется за минимальное время. Такой набор данных представляет собой комбинацию значений, на которой алгоритм выполняет меньше всего действий. Если мы исследуем алгоритм поиска, то набор данных является наилучшим, если искомое значение (обычно называемое *целевым значением*) записано в первой проверяемой алгоритмом ячейке. Такому алгоритму, вне зависимости от его сложности, потребуется одно сравнение. Заметим, что при поиске в списке, каким бы длинным он ни был, наилучший случай требует постоянного времени. Вообще, время выполнения алгоритма в наилучшем случае очень часто оказывается маленьким или просто постоянным, поэтому мы будем редко проводить подобный анализ.

Наихудший случай

Анализ наихудшего случая чрезвычайно важен, поскольку он говорит о максимальном времени работы алгоритма. При анализе наихудшего случая необходимо найти входные данные, на которых алгоритм будет выполнять больше всего работы. Для алгоритма поиска подобные входные данные — это список, в котором целевое значение окажется последним из рассматриваемых или вообще отсутствует. В результате может потребоваться N сравнений. Анализ наихудшего случая дает верхние оценки для времени работы частей нашей программы в зависимости от выбранных алгоритмов.

Средний случай

Анализ среднего случая является самым сложным, поскольку он требует учета множества разнообразных деталей. В основе анализа лежит определение различных групп, на которые следует разбить возможные входные наборы данных. На втором шаге определяется вероятность, с которой входной набор данных принадлежит каждой группе. На третьем шаге подсчитывается время работы алгоритма на данных из каждой группы. Время работы алгоритма на всех входных данных одной группы должно быть одинаковым, в противном случае группу следует разбить еще раз. Среднее время работы вычисляется по формуле[1]

$$A(n) = \sum_{i=1}^{m} p_i t_i, \tag{1.1}$$

где через n обозначен размер входных данных, через m — число групп,

[1]Несоответствие левой и правой частей формулы (левая зависит от n, а правая нет) является кажущимся: и разбиение на группы, и значения параметров p_i и t_i в правой части тоже зависят от n. — *Прим. перев.*

через p_i — вероятность того, что входные данные принадлежат группе с номером i, а через t_i — время, необходимое алгоритму для обработки данных из группы с номером i.

В некоторых случаях мы будем предполагать, что вероятности попадания входных данных в каждую из групп одинаковы. Например, если групп пять, то вероятность попасть в первую группу такая же, как вероятность попасть во вторую, и т. д., то есть вероятность попасть в каждую группу равна 0,2. В этом случае среднее время работы можно либо оценить по предыдущей формуле, либо воспользоваться эквивалентной ей упрощенной формулой

$$A(n) = \frac{1}{m}\sum_{i=1}^{m} t_i, \tag{1.2}$$

справедливой при равной вероятности всех групп.

1.2.2. Упражнения

1. Напишите алгоритм подсчета среднего значения, или медианы, трех целых чисел. Входные данные для такого алгоритма распадаются на шесть групп; опишите их. Какой случай для алгоритма является наилучшим? Наихудшим? Средним? (Если наилучший и наихудший случаи совпадают, то перепишите ваш алгоритм с простыми условиями, не пользуясь вре́менными переменными, так, чтобы наилучший случай был действительно лучше наихудшего.)
2. Напишите алгоритм, проверяющий, верно ли, что данные четыре целых числа попарно различны. Деление на группы возможных наборов входных данных зависит от структуры вашего алгоритма либо от структуры задачи. Какой из классов обеспечивает наилучший случай для вашего алгоритма? Наихудший? Средний? (Если наилучший и наихудший случаи совпадают, то перепишите ваш алгоритм с простыми условиями, не пользуясь вре́менными переменными, так, чтобы наилучший случай был лучше наихудшего.)
3. Напишите алгоритм, который по данному списку чисел и среднему значению этих чисел определяет, превышает ли число элементов списка, больших среднего значения, число элементов, меньших этого значения, или наоборот. Опишите группы, на которые распадаются возможные наборы входных данных. Какой случай для алгоритма является наилучшим? Наихудшим? Средним? (Если наилучший и наихудший случаи совпадают, то перепишите ваш алгоритм так, чтобы он останавливался, как только ответ на поставленный вопрос становится известным, делая наилучший случай лучше наихудшего.)

1.3. Необходимые математические сведения

На протяжении всей книги мы будем пользоваться немногими математическими понятиями. К их числу принадлежат *округление числа* влево и вправо. Округлением влево (или *целой частью*) числа x мы назовем наибольшее целое число, не превосходящее x (обозначается $\lfloor x \rfloor$). Так, $\lfloor 2{,}5 \rfloor = 2$, а $\lfloor -7{,}3 \rfloor = -8$. Округлением вправо числа x называется наименьшее целое число, которое не меньше, чем x (обозначается через $\lceil x \rceil$). Так, $\lceil 2{,}5 \rceil = 3$, а $\lceil -7{,}3 \rceil = -7$. Чаще всего мы будем применять эти операции к положительным числам.

Округление влево и вправо будет применяться, когда мы хотим подсчитать, сколько раз было выполнено то или иное действие, причем результат будет представлять собой частное некоторых величин. Например, при попарном сравнении N элементов, когда первый элемент сравнивается со вторым, третий с четвертым и т. д., число сравнений будет равно $\lfloor N/2 \rfloor$. Если $N = 10$, то попарных сравнений будет 5, и $\lfloor 10/2 \rfloor = \lfloor 5 \rfloor = 5$. А если $N = 11$, то число сравнений не изменится и останется равным пяти, и $\lfloor 11/2 \rfloor = \lfloor 5{,}5 \rfloor = 5$.

Факториал натурального числа N обозначается через $N!$ и представляет собой произведение всех натуральных чисел от 1 до N. Например $3! = 3 \cdot 2 \cdot 1 = 6$, а $6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$. Видно, что факториал становится большим очень быстро. Мы внимательно изучим этот вопрос в § 1.4.

1.3.1. Логарифмы

Логарифмам предстоит играть существенную роль в нашем анализе, поэтому нам следует обсудить их свойства. *Логарифмом*[2] числа x по основанию b называется такой показатель степени a, в которую нужно возвести b, чтобы получить x. Так, $\log_{10} 45$ приблизительно равен 1,653, поскольку $10^{1{,}653} \approx 45$. *Основанием* логарифма может служить любое число[3], однако в нашем анализе чаще всего будут встречаться логарифмы по основаниям 10 и 2.

Логарифм — строго возрастающая функция[4]. Это означает, что если $x > y$, то $\log_b x > \log_b y$ для любого основания b. Логарифм — взаимно однозначная функция, т. е. если $\log_b x = \log_b y$, то $x = y$. Нужно знать также следующие важные свойства логарифма, справедливые при поло-

[2] $\log_b x = a$ тогда и только тогда, когда $b^a = x$. — *Прим. перев.*

[3] Любое положительное число, отличное от 1. — *Прим. перев.*

[4] Если его основание больше 1. — *Прим. перев.*

жительных значениях входящих в них переменных:

$$\log_b 1 = 0; \tag{1.3}$$

$$\log_b b = 1; \tag{1.4}$$

$$\log_b(xy) = \log_b x + \log_b y; \tag{1.5}$$

$$\log_b x^y = y \log_b x; \tag{1.6}$$

$$\log_a x = \frac{\log_b x}{\log_b a}. \tag{1.7}$$

С помощью этих свойств можно упрощать выражения, содержащие логарифмы. Тождество (1.7) позволяет менять основание логарифма. Большинство калькуляторов позволяют вычислять логарифмы по основанию 10 и *натуральные* логарифмы (по основанию e), а что если вам нужно вычислить $\log_{42} 75$? С помощью равенства (1.7) вы легко получаете ответ:

$$\log_{42} 75 = \frac{\log_e 75}{\log_e 42} \approx 1{,}155.$$

1.3.2. Бинарные деревья

Бинарное дерево представляет собой структуру, в которой каждый узел (или вершина) имеет не более двух *узлов-потомков* и в точности одного *родителя.* Самый верхний узел дерева является единственным узлом без родителей; он называется *корневым узлом*, или корнем. Бинарное дерево с N узлами имеет не меньше $\lfloor \log_2 N + 1 \rfloor$ уровней (при максимально плотной упаковке узлов). Например, у *полного* бинарного дерева с 15 узлами один корень, два узла на втором уровне, четыре узла на третьем уровне и восемь узлов на четвертом уровне; наше равенство также дает $\lfloor \log_2 15 \rfloor + 1 = \lfloor 3{,}9 \rfloor + 1 = 4$ уровня. Заметим, что добавление еще одного узла к дереву приведет к появлению нового уровня, и их число станет равным $\lfloor \log_2 16 \rfloor + 1 = \lfloor 4 \rfloor + 1 = 5$. В самом большом бинарном дереве с N узлами N уровней: у каждого узла этого дерева в точности один потомок (и само дерево представляет собой просто список).

Если уровни дерева занумеровать, считая, что корень лежит на уровне 1, то на уровне с номером k лежит не более чем 2^{k-1} узла. У полного бинарного дерева с j уровнями (занумерованными от 1 до j) все *листья*[5] лежат на уровне с номером j, и у каждого узла на уровнях с первого по $j-1$ в точности два непосредственных потомка. В полном бинарном дереве с j уровнями $2^j - 1$ узел. Эта информация не раз нам понадобится в дальнейшем. Советуем для лучшего понимания этих формул порисовать бинарные деревья и сравнить свои результаты с приведенными выше формулами.

[5] Узлы без потомков. — *Прим. перев.*

1.3.3. Вероятности

Мы собираемся анализировать алгоритмы в зависимости от входных данных, а для этого нам необходимо оценивать, насколько часто встречаются те или иные наборы входных данных. Тем самым нам придется работать с вероятностями того, что входные данные удовлетворяют тем или иным условиям. Вероятность того или иного события представляет собой число в интервале между нулем и единицей, причем вероятность 0 означает, что событие не произойдет никогда, а вероятность 1 — что оно произойдет наверняка. Если нам известно, что число различных возможных значений входа в точности равно 10, то мы можем с уверенностью сказать, что вероятность каждого такого входа заключена между 0 и 1, и что сумма всех этих вероятностей равна 1, поскольку один из них наверняка должен быть реализован. Если возможности реализации каждого из входов одинаковы, то вероятность каждого из них равна 0,1 (один из 10, или 1/10).

По большей части наш анализ будет заключаться в описании всех возможностей, а затем мы будем предполагать, что все они равновероятны. Если общее число возможностей равно N, то вероятность реализации каждой из них равна $1/N$.

1.3.4. Формулы суммирования

При анализе алгоритмов нам придется складывать величины из некоторых наборов величин. Пусть, скажем, у нас есть алгоритм с циклом. Если переменная цикла принимает значение 5, то цикл выполняется 5 раз, а если ее значение равно 20, то двадцать. Вообще, если значение переменной цикла равно M, то цикл выполняется M раз. В целом если переменная цикла пробегает все значения от 1 до N, то суммарное число выполнений цикла равно сумме всех натуральных чисел от 1 до N. Мы записываем эту сумму в виде $\sum\limits_{i=1}^{N} i$. В нижней части знака суммы стоит начальное значение переменной суммирования, а в его верхней части — ее конечное значение. Понятно, как такое обозначение связано с интересующими нас суммами.

Если какое-нибудь значение записано в виде подобной суммы, то его зачастую стоит упростить, чтобы результат можно было сравнивать с другими подобными выражениями. Не так уж просто сообразить, какое из двух чисел $\sum\limits_{i=11}^{N}(i^2-i)$ и $\sum\limits_{i=0}^{N}(i^2-20i)$ больше. Поэтому для упрощения сумм мы будем пользоваться выписанными ниже формулами, в которых C — не зависящая от i постоянная.

$$\sum_{i=1}^{N} Ci = C\sum_{i=1}^{N} i; \tag{1.8}$$

$$\sum_{i=L}^{N} i = \sum_{i=0}^{N-L} (i+L); \tag{1.9}$$

$$\sum_{i=L}^{N} i = \sum_{i=0}^{N} i - \sum_{i=0}^{L-1} i; \tag{1.10}$$

$$\sum_{i=1}^{N} (A+B) = \sum_{i=1}^{N} A + \sum_{i=1}^{N} B; \tag{1.11}$$

$$\sum_{i=0}^{N} (N-i) = \sum_{i=0}^{N} i. \tag{1.12}$$

Равенство (1.12) означает просто, что сумма чисел от 0 до N равна сумме чисел от N до 0. В некоторых случаях применение этого равенства упрощает вычисления.

$$\sum_{i=1}^{N} 1 = N; \tag{1.13}$$

$$\sum_{i=1}^{N} C = CN; \tag{1.14}$$

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}. \tag{1.15}$$

Равенство (1.15) легко запомнить, если разбить числа от 1 до N на пары. Складывая 1 с N, 2 с $N-1$ и т. д., мы получим набор сумм, каждая из которых равна $N+1$. Сколько всего будет таких сумм? Разумеется, их число равно половине числа разбиваемых на пары элементов, т. е. $N/2$. Поэтому сумма всех N чисел равна

$$\frac{N}{2}(N+1) = \frac{N(N+1)}{2}.$$

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} = \frac{2N^3+3N^2+N}{6}; \tag{1.16}$$

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1. \tag{1.17}$$

Равенство (1.17) легко запомнить через двоичные числа. Сумма степеней двойки от нулевой до десятой равна двоичному числу 11111111111. Прибавив к этому числу 1, мы получим 100000000000, т. е. 2^{11}. Но этот результат на 1 больше суммы степеней двойки от нулевой до десятой, поэтому сама сумма равна $2^{11} - 1$. Если теперь вместо 10 подставить N, то мы придем к равенству (1.17).

$$\sum_{i=1}^{N} A^i = \frac{A^{N+1} - 1}{A - 1} \quad \text{для любого числа } A; \tag{1.18}$$

$$\sum_{i=1}^{N} i2^i = (N-1)2^{N+1} + 2; \tag{1.19}$$

$$\sum_{i=1}^{N} \frac{1}{i} \approx \ln N; \tag{1.20}$$

$$\sum_{i=1}^{N} \log_2 i \approx N \log_2 N - 1{,}5. \tag{1.21}$$

При упрощении сумм можно сначала разбивать их на более простые суммы с помощью равенств (1.8)–(1.12), а затем заменять суммы с помощью остальных тождеств.

1.3.5. Упражнения

1. Типичный калькулятор умеет вычислять натуральные логарифмы (по основанию e) и логарифмы по основанию 10. Как с помощью этих операций вычислить $\log_{27} 59$?

2. Допустим, у нас есть честная кость с пятью гранями, на которых написаны числа от 1 до 5. Какова вероятность выпадения каждого из чисел $1, \ldots, 5$ при бросании кости? В каком диапазоне могут меняться суммы значений при бросании двух таких костей? Какова вероятность появления каждого из чисел в этом диапазоне?

3. Допустим, у нас есть честная 8-гранная кость, на гранях которой написаны числа $1, 2, 3, 3, 4, 5, 5, 5$. Какова вероятность выбросить каждое из чисел от 1 до 5? В каком диапазоне могут меняться суммы значений при бросании двух таких костей? Какова вероятность появления каждого из чисел в этом диапазоне?

4. На гранях четырех кубиков написаны такие числа:

$$d_1\colon 1, 2, 3, 9, 10, 11; \quad d_3\colon 5, 5, 6, 6, 7, 7;$$
$$d_2\colon 0, 1, 7, 8, 8, 9; \quad d_4\colon 3, 4, 4, 5, 11, 12.$$

Вычислите для каждой пары кубиков вероятность того, что на первом кубике выпадет бо́льшее значение, чем на втором, и наоборот. Результаты удобно представлять в виде (4×4)-матрицы, в которой строчка соответствует первому кубику, а столбец — второму. (Мы предполагаем, что бросаются разные кубики, поэтому диагональ матрицы следует оставить пустой.) У этих кубиков есть интересные свойства. Обнаружите ли вы их?

5. На столе лежат пять монет. Вы переворачиваете случайную монету. Определите для каждого из четырех приведенных ниже случаев вероятность того, что после переворота решка будет на бо́льшем числе монет.

   а) два орла и три решки; в) четыре орла и одна решка;
   б) три орла и две решки; г) один орел и четыре решки.

6. На столе лежат пять монет. Вы переворачиваете один раз каждую монету. Определите для каждого из четырех приведенных ниже случаев вероятность того, что после переворота решка будет на бо́льшем числе монет.

   а) два орла и три решки; в) четыре орла и одна решка;
   б) три орла и две решки; г) один орел и четыре решки.

7. Найдите эквивалентное представление следующих выражений, не содержащее знака суммы:

   а) $\sum_{i=1}^{N}(3i+7)$; в) $\sum_{i=7}^{N} i$; д) $\sum_{i=1}^{N} 6^i$;
   б) $\sum_{i=1}^{N}(i^2-2i)$; г) $\sum_{i=5}^{N}(2i^2+1)$; е) $\sum_{i=7}^{N} 4^i$.

1.4. Скорости роста

Точное знание количества операций, выполненных алгоритмом, не играет существенной роли в его анализе. Куда более важным оказывается скорость роста этого числа при возрастании объема входных данных. Она называется *скоростью роста алгоритма*. Небольшие объемы данных не столь интересны, как то, что происходит при возрастании этих объемов.

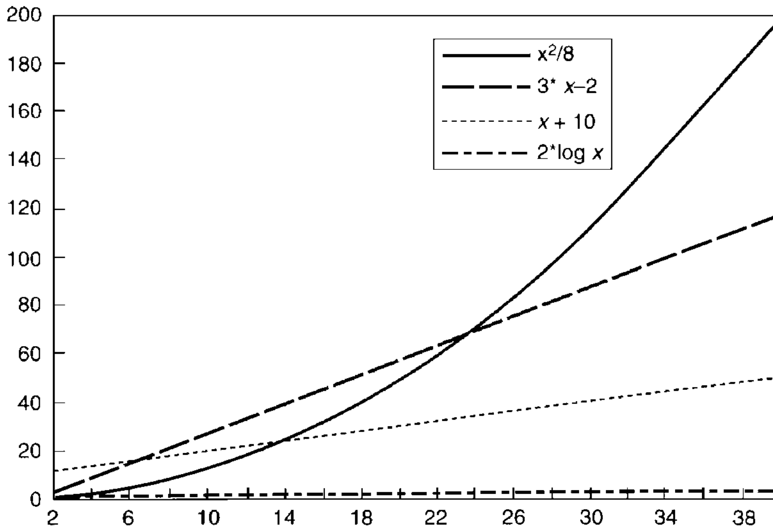

Рис. 1.1. Графики четырех функций

Нас интересует только общий характер поведения алгоритмов, а не его подробности. Если внимательно посмотреть на рис. 1.1, то можно отметить следующие особенности поведения графиков функций. Функция, содержащая x^2, сначала растет медленно, однако чем больше x, тем выше скорость роста функции. Скорость роста функций, содержащих x, постоянна на всем интервале значений переменной. Функция $2\log x$ вообще кажется постоянной, но это обман зрения. На самом деле она растет, только очень медленно. Относительная высота значений функций также зависит от того, большие или маленькие значения переменной мы рассматриваем. Сравним значения функций при $x = 2$. Функцией с наименьшим значением в этой точке является $x^2/8$, а с наибольшим — функция $x + 10$. Однако с возрастанием x функция $x^2/8$ становится и остается впоследствии наибольшей.

Подводя итоги, отметим, что при анализе алгоритмов нас будет интересовать скорее класс скорости роста, к которому относится алгоритм, нежели точное количество выполняемых им операций каждого типа. Относительный «размер» функции будет интересовать нас лишь при больших значениях переменной x.

Некоторые часто встречающиеся классы функций приведены в табл. 1.1. В этой таблице собраны значения функций из данного класса на широком диапазоне значений аргумента. Видно, что при небольших размерах входных данных значения функций отличаются незначительно, однако

при росте этих размеров разница существенно возрастает. Эта таблица усиливает впечатление от рис. 1.1. Поэтому мы и будем изучать, что происходит при больших объемах входных данных, поскольку на малых объемах принципиальная разница оказывается скрытой.

Таблица 1.1. Классы роста функций

	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
1	0,0	1,0	0,0	1,0	1,0	2,0
2	1,0	2,0	2,0	4,0	8,0	4,0
5	2,3	5,0	11,6	25,0	125,0	32,0
10	3,3	10,0	33,2	100,0	1000,0	1024,0
15	3,9	15,0	58,6	225,0	3375,0	32768,0
20	4,3	20,0	86,4	400,0	8000,0	1048576,0
30	4,9	30,0	147,2	900,0	27000,0	1073741824,0
40	5,3	40,0	212,9	1600,0	64000,0	1099511627776,0
50	5,6	50,0	282,2	2500,0	125000,0	1125899906842620,0
60	5,9	60,0	354,4	3600,0	216000,0	1152921504606850000,0
70	6,1	70,0	429,0	4900,0	343000,0	1180591620717410000000,0
80	6,3	80,0	505,8	6400,0	512000,0	1208925819614630000000000,0
90	6,5	90,0	584,3	8100,0	729000,0	1237940039285380000000000000,0
100	6,6	100,0	664,4	10000,0	1000000,0	1267650600228230000000000000000,0

Данные рис. 1.1 и табл. 1.1 иллюстрируют еще одно свойство функций. Быстрорастущие функции доминируют над функциями с более медленным ростом. Поэтому если мы обнаружим, что сложность алгоритма представляет собой сумму двух или нескольких таких функций, то будем часто отбрасывать все функции, кроме тех, которые растут быстрее всего. Если, например, мы установим при анализе алгоритма, что он делает $x^3 - 30x$ сравнений, то мы будем говорить, что сложность алгоритма растет как x^3. Причина этого в том, что уже при $x = 100$ входных данных разница между x^3 и $x^3 - 30x$ составляет лишь 0,3%. В следующем разделе мы формализуем эту мысль.

1.4.1. Классификация скоростей роста

Скорость роста сложности алгоритма играет важную роль, и мы видели, что скорость роста определяется старшим, доминирующим членом формулы. Поэтому мы будем пренебрегать младшими членами, которые растут медленнее. Отбросив все младшие члены, мы получаем то, что называется *порядком* функции или алгоритма, *скоростью роста* сложности которого она является. Алгоритмы можно сгруппировать по скорости роста их сложности. Мы вводим три категории: алгоритмы, сложность

которых растет по крайней мере так же быстро, как данная функция, алгоритмы, сложность которых растет с той же скоростью, и алгоритмы, сложность которых растет медленнее, чем эта функция.

Омега большое

Класс функций, растущих по крайней мере так же быстро, как f, мы обозначаем через $\Omega(f)$ (читается *омега большое*). Функция g принадлежит этому классу, если при всех значениях аргумента n, больших некоторого порога n_0, значение $g(n) > cf(n)$ для некоторого положительного числа c. Можно считать, что класс $\Omega(f)$ задается указанием своей нижней границы: все функции из него растут по крайней мере так же быстро, как f.

Мы занимаемся эффективностью алгоритмов, поэтому класс $\Omega(f)$ не будет представлять для нас большого интереса: например в $\Omega(n^2)$ входят все функции, растущие быстрее, чем n^2, скажем, n^3 и 2^n.

О большое

На другом конце спектра находится класс $O(f)$ (читается *о большое*). Этот класс состоит из функций, растущих не быстрее f. Функция f представляет собой верхнюю границу класса $O(f)$. С формальной точки зрения функция g принадлежит классу $O(f)$, если $g(n) \leqslant cf(n)$ для всех n, начиная с некоторого порога n_0, и для некоторой положительной константы c.

Этот класс чрезвычайно важен для нас. При сравнении двух алгоритмов нас будет интересовать, принадлежит ли сложность первого из них классу «о большое» от сложности второго. Если это так, то значит второй алгоритм не лучше первого решает поставленную задачу.

Тета большое

Через $\Theta(f)$ (читается *тета большое*) мы обозначаем класс функций, растущих с той же скоростью, что и f. С формальной точки зрения этот класс представляет собой пересечение двух предыдущих классов, $\Theta(f) = \Omega(f) \cap O(f)$.

При сравнении алгоритмов нас будут интересовать такие, которые решают задачу быстрее, чем уже изученные. Поэтому если найденный алгоритм относится к классу Θ, то он нам не очень интересен. Мы не будем часто ссылаться на этот класс.

Описание класса О большое

Проверить, принадлежит ли данная функция классу $O(f)$, можно двумя способами: либо с помощью данного выше определения, либо воспользовавшись следующим описанием:

$$g \in O(f), \text{ если } \lim_{n\to\infty} \frac{g(n)}{f(n)} = c \text{ для некоторой константы } c. \qquad (1.22)$$

Это означает, что если предел отношения $g(n)/f(n)$ существует и он меньше бесконечности, то $g \in O(f)$. Для некоторых функций это не так-то просто проверить. По правилу Лопиталя, в таком случае можно заменить предел отношения самих функций пределом отношения их производных.

Обозначение
Каждый из классов $\Theta(f)$, $\Omega(f)$ и $O(f)$ является множеством, и поэтому имеет смысл выражение «g — элемент этого множества». В анализе, однако, нередко пишут $g = O(f)$, что на самом деле означает $g \in O(f)$.

1.4.2. Упражнения

1. Расположите следующие функции в порядке возрастания скорости роста. Если некоторые из них растут с одинаковой скоростью, то объедините их овалом.

2^n;	$\log_2 \log_2 n$;	$n^3 + \log_2 n$;	$\log_2 n$;	$n - n^2 + 5n^3$;
2^{n-1};	n^2;	n^3;	$n \log_2 n$;	$(\log_2 n)^2$;
$\sqrt{n}$;	6;	$n!$;	n;	$(3/2)^n$.

2. Для каждой из приведенных ниже пар функций f и g выполняется одно из равенств: либо $f = O(g)$, либо $g = O(f)$, но не оба сразу. Определите, какой из случаев имеет место.

а) $f(n) = (n^2 - n)/2$, $g(n) = 6n$;
б) $f(n) = n + 2\sqrt{n}$, $g(n) = n^2$;
в) $f(n) = n + n \log_2 n$, $g(n) = n\sqrt{n}$;
г) $f(n) = n^2 + 3n + 4$, $g(n) = n^3$;
д) $f(n) = n \log_2 n$, $g(n) = n\sqrt{n}/2$;
е) $f(n) = n + \log_2 n$, $g(n) = \sqrt{n}$;
ж) $f(n) = 2\log_2 n^2$, $g(n) = \log_2 n + 1$;
з) $f(n) = 4n \log_2 n + n$, $g(n) = (n^2 - n)/2$.

1.5. Метод турниров

Метод турниров можно использовать для решения ряда задач, в которых информация, полученная в результате первого прохода по данным, может облегчить последующие проходы. Если мы воспользуемся им для поиска наибольшего значения, то он потребует построения бинарного дерева, все элементы которого являются листьями. На каждом уровне два элемента объединены в пару, причем наибольший из двух элементов копируется в родительский узел. Процесс повторяется до достижения корневого узла. Полное дерево турнира для фиксированного набора данных изображено на рис. 1.2.

В упражнении 5 раздела 1.1.3 упоминалось, что мы разработаем алгоритм поиска второго по величине элемента списка из N значений, требующий около N сравнений. Поможет нам в этом метод турниров. В результате каждого сравнения мы получаем «победителя» и «проигравшего».

Проигравших мы забываем, и вверх по дереву двигаются только победители. Всякий элемент, за исключением наибольшего, «проигрывает» в точности в одном сравнении. Поэтому для построения дерева турнира требуется $N - 1$ сравнение.

Второй по величине элемент мог проиграть только наибольшему. Спускаясь по дереву вниз, мы составляем список элементов, проигравших наибольшему. Формулы для деревьев из раздела 1.3.2 показывают, что число таких элементов не превосходит $\lceil \log_2 N \rceil$. Их поиск по дереву потребует $\lceil \log_2 N \rceil$ сравнений; еще $\lceil \log_2 N \rceil - 1$ сравнений необходимо для выбора наибольшего из них. На всю работу уйдет $N + 2\lceil \log_2 N \rceil - 2$, т. е. $O(N)$ сравнений.

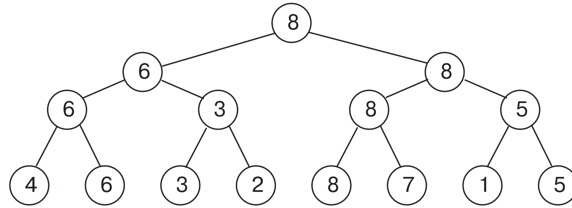

Рис. 1.2. Дерево турнира для набора из восьми значений

Метод турниров годится и для упорядочивания списка значений. В главе 4 нам встретится сортировка куч, основанная на этом методе.

1.5.1. Нижние границы

Алгоритм является *оптимальным*, если любой алгоритм, решающий ту же задачу, работает не быстрее данного. Как узнать, оптимален ли наш алгоритм? Или, быть может, он не оптимален, но все-таки достаточно хорош? Для ответа на эти вопросы мы должны знать наименьшее количество операций, необходимое для решения конкретной задачи, которое называется нижней границей. Для этого нам следует изучать именно задачу, а не решающие ее алгоритмы. Нижняя граница определяет объем работы, необходимый для решения поставленной задачи, и показывает, что любой алгоритм, претендующий на то, что решает ее быстрее, обязан неправильно обрабатывать некоторые ситуации.

Для анализа процесса сортировки списка из трех чисел вновь воспользуемся бинарным деревом. Внутренние узлы дерева будем помечать парами сравниваемых элементов. Порядок элементов, обеспечивающий проход по данной ветви дерева, изображается в соответствующем листе дерева. Дерево для списка из трех элементов изображено на рис. 1.3. Такое дерево называется *деревом решения*.

Всякий алгоритм сортировки создает свое дерево решений в зависимости от того, какие элементы он сравнивает. Самый длинный путь в дереве решений от корня к листу соответствует наихудшему случаю. Наилучшему случаю отвечает кратчайший путь. Средний случай описывается

частным от деления числа ребер в дереве решений на число листов в нем. На первый взгляд, ничего не стоит нарисовать дерево решений и подсчитать нужные числа. Представьте себе, однако, размер дерева решений при сортировке 10 чисел. Как уже говорилось, число возможных порядков равно 3 628 800. Поэтому в дереве будет по меньшей мере 3 628 800 листьев, а может и больше, поскольку к одному и тому же порядку можно прийти в результате различных последовательностей сравнений. В таком дереве должно быть не меньше 22 уровней.

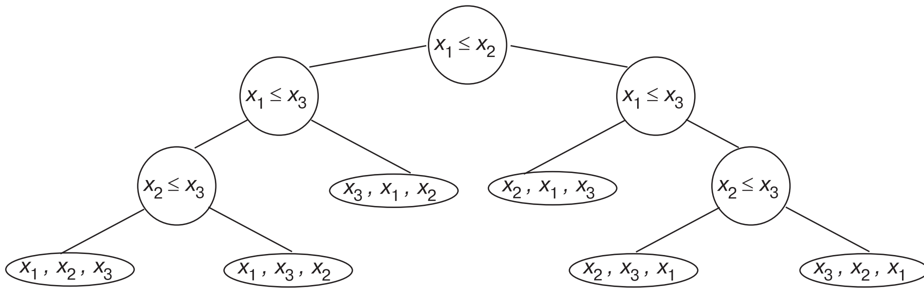

Рис. 1.3. Дерево решений для сортировки трехэлементного списка

Как же можно получить границы для сложности алгоритма с помощью дерева решений? Мы знаем, что корректный алгоритм должен упорядочивать любой список данных, независимо от исходного порядка элементов в нем. Для любой перестановки входных значений должен существовать по крайней мере один лист, а это означает, что в дереве решений должно быть не меньше $N!$ листьев. Если алгоритм по-настоящему эффективен, то каждая перестановка появится в нем лишь однажды. Сколько уровней должно быть в дереве с $N!$ листьями? Мы уже видели, что в каждом очередном уровне вдвое больше узлов, чем в предыдущем. Число узлов на уровне K равно 2^{K-1}, поэтому в нашем дереве решений будет L уровней, где L — наименьшее целое число, для которого $N! \leqslant 2^{L-1}$. Логарифмируя это неравенство, получаем

$$\log_2 N! \leqslant L - 1.$$

Можно ли избавиться от факториала, чтобы найти наименьшее значение L? Обратимся к свойствам факториала. Воспользуемся тем, что

$$\log_2 N! = \log_2(N(N-1)(N-2)\ldots 1),$$

откуда в силу равенства (1.5)

$$\log_2(N(N-1)(N-2)\ldots 1) = \log_2 N + \log_2 (N-1) + \cdots + \log_2 1 = \sum_{i=1}^{N} \log_2 i.$$

Из равенства (1.21) получаем

$$\sum_{i=1}^{N} \log_2 i \approx N \log_2 N - 1{,}5, \log_2(N!) \approx N \log_2 N.$$

Это означает, что минимальная глубина L дерева решений для сортировки имеет порядок $O(N \log_2 N)$. Теперь мы знаем, что любой алгоритм сортировки порядка $O(N \log N)$ является наилучшим, и его можно считать оптимальным. Более того, из этого исследования вытекает, что алгоритм, решающий задачу сортировки быстрее, чем за $O(N \log N)$ операций, не может работать правильно.

При выводе нижней границы для алгоритма сортировки предполагалось, что алгоритм работает путем последовательного попарного сравнения элементов списка. В главе 4 мы познакомимся с другим алгоритмом сортировки (корневая сортировка), количество операций в котором линейно зависит от длины списка. Для достижения цели этот алгоритм не сравнивает значения ключей, а разбивает их на «кучки».

1.5.2. Упражнения

1. Нарисуйте дерево турнира для следующего набора значений: [14, 3, 10, 7, 13, 4, 15, 5, 6, 22, 16, 1, 8, 2, 9, 12]. Какие элементы будут сравниваться на втором этапе поиска второго по величине значения?
2. Нарисуйте дерево турнира для следующего набора значений: [13, 1, 12, 3, 9, 5, 2, 11, 10, 8, 6, 4, 7]. Какие элементы будут сравниваться на втором этапе поиска второго по величине значения?
3. Найдите нижнюю границу числа сравнений при поиске по списку из N элементов. При обдумывании ответа постарайтесь представить себе, как выглядит дерево решений. (*Подсказка*: узлы должны быть помечены местоположением ключа.) Что можно сказать о числе необходимых для поиска сравнений, если упаковывать узлы настолько плотно, насколько это возможно?

1.6. Анализ программ

Предположим, что у нас есть большая сложная программа, которая работает медленнее, чем хотелось бы. Как обнаружить части программы, тонкая настройка которых привела бы к заметному ускорению ее работы?

Можно исследовать программу и найти подпрограммы (иногда называемые также методами, процедурами или функциями), в которых много вычислений или циклов, и попытаться усовершенствовать их. Прило-

жив значительные усилия, мы можем обнаружить, что эффект не слишком заметен, поскольку отобранные подпрограммы используются нечасто. Лучше сначала найти часто используемые подпрограммы и попробовать улучшить их. Один из способов поиска состоит в том, чтобы ввести набор глобальных счетчиков, по одному на каждую подпрограмму. При начале работы программы все счетчики обнуляются. Затем в каждую подпрограмму первой строкой вставляется команда увеличения соответствующего счетчика на 1. При всяком обращении к подпрограмме будет происходить увеличение счетчика, и в конце работы наш набор счетчиков укажет, сколько раз происходил вызов каждой подпрограммы. Тогда можно будет увидеть, какие подпрограммы вызывались часто, а какие — всего несколько раз.

Предположим, что в нашей программе некоторая простая подпрограмма вызывалась 50 000 раз, а каждая из сложных подпрограмм — лишь однажды. Тогда нужно уменьшить число операций в сложных программах на 50 000, чтобы достичь того же эффекта, что производит удаление всего одной операции в простой подпрограмме. Понятно, что простое улучшение одной подпрограммы найти гораздо проще, чем 50 000 улучшений в группе подпрограмм.

Счетчики можно использовать и на уровне подпрограмм. В этом случае мы создаем набор глобальных счетчиков, по одному в каждой значимой точке, которую мы можем предугадать. Предположим, что мы хотим узнать, сколько раз выполняется каждая из частей `then` и `else` некоторого оператора `if`. Тогда можно создать два счетчика и увеличивать первый из них при попадании в часть `then`, а второй — при попадании в часть `else`. В конце работы программы эти счетчики будут содержать интересующую нас информацию. Другими значимыми точками могут оказаться вхождения в циклы и операторы ветвления. Вообще говоря, счетчики стоит устанавливать в любых местах возможной передачи управления.

В конце работы программы установленные счетчики будут содержать информацию о числе обращений к каждому блоку подпрограммы. Затем можно исследовать возможность улучшить те части подпрограммы, которые выполняют наибольший объем работы.

Этот процесс важен, и во многих компьютерах и системах разработки программного обеспечения есть средства автоматического получения такой информации о программах.

ГЛАВА 2

РЕКУРСИВНЫЕ АЛГОРИТМЫ

Необходимые предварительные знания

Приступая к чтению этой главы, вы должны уметь:

– читать и разрабатывать рекурсивные алгоритмы;
– опознавать операции сравнения и арифметические операции;
– пользоваться элементарной алгеброй.

Цели

Освоив эту главу, вы должны уметь:

– выписывать рекуррентные соотношения для рекурсивных алгоритмов;
– переводить простые рекуррентные соотношения в компактную формулу общего члена;
– объяснять алгоритм ближайшей пары;
– объяснять алгоритм выпуклой оболочки;
– генерировать перестановки как рекурсивно, так и итеративно;
– объяснять связь между рекурсиями и стеками.

Советы по изучению

Изучая эту главу, самостоятельно проработайте все приведенные примеры и убедитесь, что вы их поняли. Кроме того, прежде чем читать ответ на вопрос, вам следует попробовать ответить на него самостоятельно. Подсказка или ответ на вопрос следуют непосредственно после вопроса.

Рекурсивные алгоритмы решают задачи следующим образом. Сначала алгоритм применяется к первой, малой части задачи, а затем полученный результат используется для поиска решения бо́льшей части задачи.

В некоторых случаях «часть задачи» может содержать лишь на один элемент меньше, чем задача целиком. Такие алгоритмы иногда называют «*сокращай и властвуй*». В других ситуациях «часть задачи» включает в себя половину входных данных всей задачи. Соответствующие алгоритмы принято называть «*разделяй и властвуй*».

В любом случае рекурсивный алгоритм может оказаться мощным инструментом решения задачи, поскольку включает в себя меньший, а значит и более понятный алгоритм.