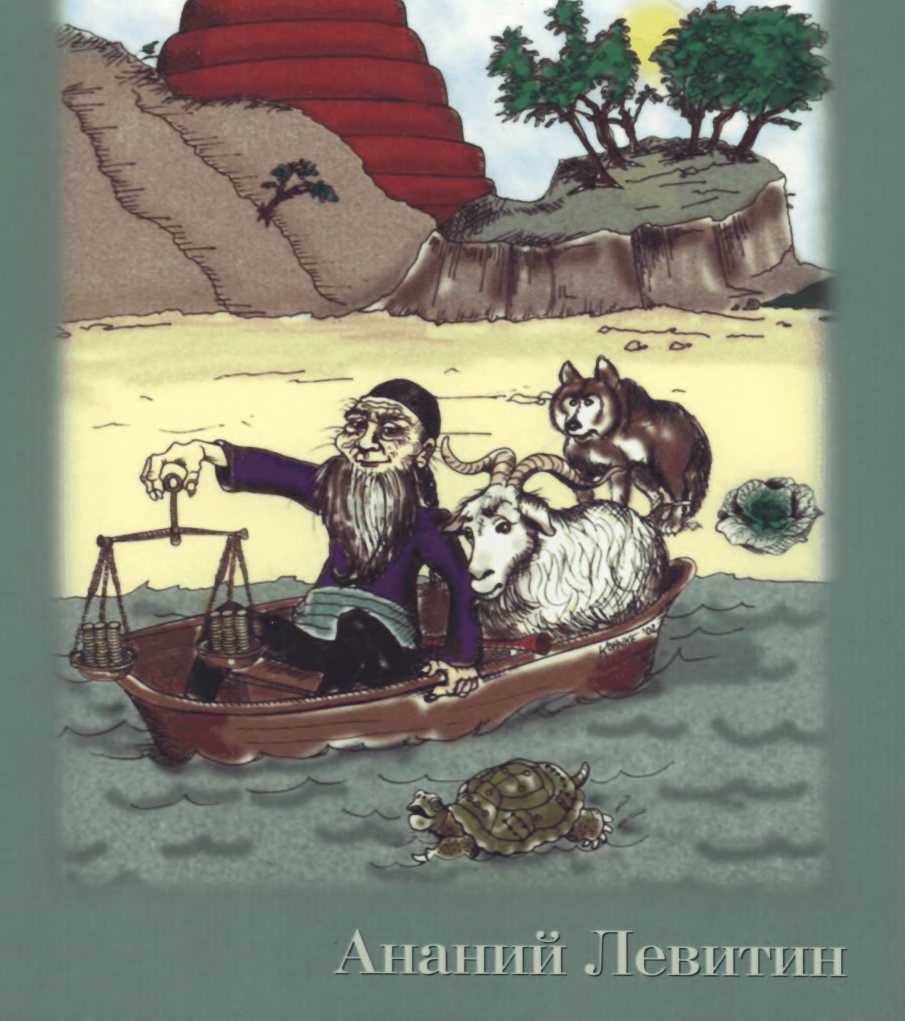
Алгоритмы

Введение в разработку

и анализ



Алгоритмы

Введение в разработку и анализ

Introduction to The Design & Analysis of Algorithms



Anany Levitin

Villanova University

▼ ▼

ADDISO N -WESLEY

Boston San Francisco New York London Toronto Sydney Tokyo Singapore Madrid

Mexico City Munich Paris Cape Town Hong Kong Montreal

Алгоритмы



Введение в разработку и анализ

Ананий Левитин

Университет Вилланова

image4



ББК 32.973.26-018.2.75 Л36 УДК 681.3.07

Издательский дом “Вильямс”

Зав. редакцией С.Я. Тригуб

Перевод с английского и редакция канд. физ.-мат.наук С.Г. Тригуб, канд.техн.наук И.В. Красикова

По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу: [info@williamspublishing.com](mailto:info@williamspublishing.com), <http://www.williamspublishing.com> 115419, Москва, а/я 783; 03150, Киев, а/я 152

Левитин, Ананий В.

Л36 Алгоритмы: введение в разработку и анализ. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2006. — 576 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0987-2 (рус.)

Эта книга, автором которой является преподаватель информатики, пред­ставляет собой один из лучших учебников, посвященных алгоритмам. Делая основной упор на понимание идей, а не на механическое рассмотрение работы того или иного алгоритма, автор излагает принципы разработки алгоритмов так, что они могут быть применены как универсальный инструментарий для широкого диапазона задач, а не только для разработки алгоритмов.

Книга ориентирована в первую очередь на студентов и аспирантов соответ­ствующих специальностей, поэтому для преподавателей она может стать хо­рошим пособием для подготовки к лекциям и источником интересных нетри­виальных задач. Книга может оказаться полезной и профессионалам в области разработки алгоритмов благодаря использованному автором новому подходу к классификации методов проектирования. Описание алгоритмов на есте­ственном языке дополняется псевдокодом, который позволяет каждому, кто имеет хотя бы начальные знания и опыт программирования, реализовать ал­горитм на используемом им языке программирования.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствую­щих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена 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 это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Addison-Wesley. Copyright © 2003.

All rights reserved. No part of this book may be reproduced or transmined 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.

Russian language edition is published by Williams Publishing House according to the Agreement with R&I Enterprises International. Copyright © 2006.

**ISBN 5-8459-0987-2 (pyc.) ISBN 0-201-74395-7 (англ.)**

© Издательский дом “Вильямс”, 2006 © Addison-Wesley. 2003

Оглавление

[Предисловие 14](#bookmark8)

Глава 1. Введение 23

[Глава 2. Основы анализа эффективности алгоритмов 73](#bookmark53)

Глава 3. Метод грубой силы 141

Глава 4. Метод декомпозиции 167

[Глава 5. Метод уменьшении размера задачи 203](#bookmark158)

[Глава 6. Метод преобразования 247](#bookmark186)

[Глава 7. Пространственно-временной компромисс 305](#bookmark222)

[Глава 8. Динамическое программирование 339](#bookmark247)

Глава 9. Жадные методы 369

[Глава 10. Ограничения мощи алгоритмов 401](#bookmark282)

[Глава 11. Преодоление ограничений 441](#bookmark307)

[Эпилог 487](#bookmark340)

[Приложение А. Формулы, использующиеся при анализе алгоритмов 491](#bookmark342)

[Приложение Б. Краткое руководство по рекуррентным соотношениям 495](#bookmark354)

[Список литературы 509](#bookmark374)

[Указания к упражнениям 517](#bookmark375)

[Предметный указатель 562](#bookmark432)

Содержание

Предисловие

Глава 1. Введение

1. Понятие алгоритма Упражнения 1.1
2. Основы решения алгоритмической задачи Понимание задачи

Определение возможностей вычислительного устройства Выбор между точным или приближенным методом решения задачи

Выбор подходящих структур данных Методы проектирования алгоритмов Методы представления алгоритмов Оценка корректности алгоритма Анализ алгоритма Кодирование алгоритма Упражнения 1.2

1. Важные типы задач Сортировка Поиск

Обработка строк Задачи из теории графов Комбинаторные задачи Геометрические задачи Численные задачи Упражнения 1.3

1. Базовые структуры данных Линейные структуры данных Графы

Деревья

Множества и словари

|  |  |  |
| --- | --- | --- |
|  | Упражнения 1.4 | 70 |
|  | Резюме | 71 |
| Глава 2. Основы анализа эффективности алгоритмов | | 73 |
| 2.1 | Основы анализа | 75 |
|  | Оценка размера входных данных | 75 |
|  | Единицы измерения времени выполнения алгоритма | 76 |
|  | Порядок роста | 78 |
|  | Эффективность алгоритма в разных случаях | 80 |
|  | Повторение пройденного | 84 |
|  | Упражнения 2.1 | 85 |
| 2.2 | Асимптотические обозначения и основные классы |  |
|  | эффективности | 87 |
|  | Нестрогое введение | 88 |
|  | О-обозначение | 88 |
|  | П-обозначение | 89 |
|  | ©-обозначение | 90 |
|  | Полезные свойства сделанных асимптотических обозначений Использование пределов для сравнения порядка роста двух | 91 |
|  | функций | 92 |
|  | Основные классы эффективности | 94 |
|  | Упражнения 2.2 | 96 |
| 2.3 | Математический анализ нерекурсивных алгоритмов | 98 |
|  | Упражнения 2.3 | 105 |
| 2.4 | Математический анализ рекурсивных алгоритмов | 107 |
|  | Упражнения 2.4 | 116 |
| 2.5 | Пример: числа Фибоначчи  Явная формула для определения n-го элемента | 119 |
|  | последовательности чисел Фибоначчи | 120 |
|  | Алгоритмы вычисления чисел Фибоначчи | 122 |
|  | Упражнения 2.5 | 125 |
| 2.6 | Эмпирический анализ алгоритмов | 127 |
|  | Упражнения 2.6 | 133 |
|  | Визуализация алгоритмов | 135 |
|  | Резюме | 139 |
| Глава 3. Метод грубой силы | | 141 |
| 3.1 | Сортировка выбором и пузырьковая сортировка | 142 |
|  | Сортировка выбором | 143 |
|  | Пузырьковая сортировка | 144 |
|  | Упражнения 3.1 | 146 |

1. Последовательный поиск и поиск подстрок методом грубой

силы 147

[Последовательный поиск 147](#bookmark121)

[Поиск подстроки 148](#bookmark122)

[Упражнения 3.2 150](#bookmark20)

1. Задачи поиска пары ближайших точек и вычисления

выпуклой оболочки с использованием грубой силы 152

[Поиск пары ближайших точек 152](#bookmark126)

[Поиск выпуклой оболочки 154](#bookmark128)

[Упражнения 3.3 157](#bookmark31)

1. [Исчерпывающий перебор 159](#bookmark131)

[Задача коммивояжера 159](#bookmark132)

[Задача о рюкзаке 160](#bookmark133)

[Задача о назначениях 163](#bookmark135)

[Упражнения 3.4 164](#bookmark40)

[Резюме 166](#bookmark52)

Глава 4. Метод декомпозиции 167

* 1. [Сортировка слиянием 169](#bookmark139)

[Упражнения 4.1 172](#bookmark61)

* 1. [Быстрая сортировка 174](#bookmark143)

[Упражнения 4.2 179](#bookmark71)

* 1. [Бинарный поиск 180](#bookmark146)

[Упражнения 4.3 183](#bookmark96)

* 1. [Обход бинарного дерева 184](#bookmark148)

[Упражнения 4.4 188](#bookmark108)

* 1. Умножение больших целых чисел и алгоритм умножения

матриц Штрассена 189

[Умножение больших целых чисел 189](#bookmark150)

[Алгоритм Штрассена для умножения матриц 192](#bookmark151)

[Упражнения 4.5 194](#bookmark119)

* 1. Решение задач о паре ближайших точек и о выпуклой

оболочке методом декомпозиции 195

[Задача о паре ближайших точек 196](#bookmark154)

[Задача о выпуклой оболочке 198](#bookmark155)

[Упражнения 4.6 200](#bookmark124)

[Резюме 201](#bookmark110)

Глава 5. Метод уменьшения размера задачи 203

1. [Сортировка вставкой 206](#bookmark159)

[Упражнения 5.1 209](#bookmark129)

1. [Поиск в глубину и поиск в ширину 211](#bookmark162)

[Поиск в глубину 212](#bookmark163)

[Поиск в ширину 215](#bookmark164)

[Упражнения 5.2 218](#bookmark136)

1. [Топологическая сортировка 220](#bookmark166)

[Упражнения 5.3 224](#bookmark142)

1. [Алгоритмы генерации комбинаторных объектов 226](#bookmark168)

[Генерация перестановок 227](#bookmark169)

[Генерация подмножеств 229](#bookmark170)

[Упражнения 5.4 231](#bookmark171)

1. Алгоритмы с использованием уменьшения на постоянный

множитель 232

[Задача поиска фальшивой монеты 233](#bookmark174)

[Умножение по-русски 234](#bookmark175)

[Задача Иосифа 235](#bookmark176)

[Упражнения 5.5 237](#bookmark167)

1. [Алгоритмы с переменным уменьшением размера 238](#bookmark177)

[Вычисление медианы и задача выбора 238](#bookmark178)

[Интерполяционный поиск 240](#bookmark180)

[Поиск и вставка в бинарное дерево поиска 242](#bookmark183)

[Упражнения 5.6 243](#bookmark147)

[Резюме 244](#bookmark137)

Глава 6. Метод преобразования 247

* 1. [Предварительная сортировка 248](#bookmark187)

[Упражнения 6.1 252](#bookmark152)

* 1. [Метод исключения Гаусса 254](#bookmark190)

[LU-разложение и другие приложения 259](#bookmark191)

[Вычисление обратной матрицы 261](#bookmark193)

[Вычисление определителя 262](#bookmark195)

[Упражнения 6.2 264](#bookmark156)

* 1. [Сбалансированные деревья поиска 265](#bookmark197)

[AVL-деревья 267](#bookmark198)

[2-3-деревья 271](#bookmark199)

[Упражнения 6.3 274](#bookmark161)

* 1. [Пирамиды и пирамидальная сортировка 275](#bookmark203)

[Понятие пирамиды 276](#bookmark204)

[Пирамидальная сортировка 281](#bookmark205)

[Упражнения 6.4 282](#bookmark206)

* 1. [Схема Горнера и возведение в степень 284](#bookmark207)

[Схема Горнера 284](#bookmark208)

[Бинарное возведение в степень 286](#bookmark210)

[Упражнения 6.5 289](#bookmark211)

* 1. [Приведение задачи 291](#bookmark213)

[Вычисление наименьшего общего кратного 292](#bookmark214)

[Подсчет путей в графе 293](#bookmark215)

[Приведение задач оптимизации 293](#bookmark216)

[Линейное программирование 295](#bookmark217)

[Приведение к задачам о графах 297](#bookmark219)

[Упражнения 6.6 299](#bookmark184)

[Резюме 301](#bookmark157)

Глава 7. Пространственно-временной компромисс 305

1. [Сортировка подсчетом 307](#bookmark223)

[Упражнения 7.1 310](#bookmark188)

1. [Улучшение входных данных в поиске подстрок 312](#bookmark227)

[Алгоритм Хорспула 313](#bookmark228)

[Алгоритм Бойера-Мура 317](#bookmark232)

[Упражнения 7.2 322](#bookmark202)

1. [Хеширование 323](#bookmark239)

[Открытое хеширование (раздельные цепочки) 325](#bookmark240)

[Закрытое хеширование (открытая адресация) 326](#bookmark241)

[Упражнения 7.3 329](#bookmark243)

1. [В-деревья 331](#bookmark244)

[Упражнения 7.4 335](#bookmark245)

[Резюме 336](#bookmark185)

Глава 8. Динамическое программирование 339

1. [Вычисление биномиальных коэффициентов 341](#bookmark248)

[Упражнения 8.1 343](#bookmark250)

1. [Алгоритмы Воршалла и Флойда 345](#bookmark252)

[Алгоритм Воршалла 345](#bookmark253)

Алгоритм Флойда поиска кратчайших путей между всеми

парами вершин 349

[Упражнения 8.2 353](#bookmark220)

1. [Оптимальные бинарные деревья поиска 354](#bookmark257)

[Упражнения 8.3 360](#bookmark226)

1. [Задача о рюкзаке и функции с запоминанием 361](#bookmark264)

[Функции с запоминанием 364](#bookmark265)

[Упражнения 8.4 366](#bookmark238)

[Резюме 368](#bookmark246)

Глава 9. Жадные методы 369

1. [Алгоритм Прима 371](#bookmark270)

[Упражнения 9.1 376](#bookmark272)

1. [Алгоритм Крускала 378](#bookmark273)

Непересекающиеся подмножества и алгоритмы поиска

объединений 381

[Упражнения 9.2 385](#bookmark275)

1. [Алгоритм Дейкстры 386](#bookmark276)

[Упражнения 9.3 390](#bookmark277)

1. [Деревья Хаффмана 392](#bookmark278)

[Упражнения 9.4 397](#bookmark279)

[Резюме 398](#bookmark221)

Глава 10. Ограничения мощи алгоритмов 401

1. [Доказательства нижних границ 402](#bookmark283)

[Тривиальные нижние границы 403](#bookmark284)

[Информационно-теоретические доказательства 404](#bookmark285)

[Доказательство “от противника” 405](#bookmark286)

[Приведение задачи 406](#bookmark287)

[Упражнения 10.1 408](#bookmark256)

1. [Деревья принятия решения 409](#bookmark289)

[Деревья принятия решения для алгоритмов сортировки 411](#bookmark291)

Деревья принятия решения для поиска в отсортированном

массиве 412

[Упражнения 10.2 415](#bookmark262)

1. [Р, NP и iVP-полные задачи 417](#bookmark295)

[Р и ЛгР-задачи 418](#bookmark296)

[N P-полные задачи 423](#bookmark297)

[Упражнения 10.3 426](#bookmark266)

1. [Численные алгоритмы 428](#bookmark300)

[Упражнения 10.4 437](#bookmark305)

[Резюме 438](#bookmark280)

Глава 11. Преодоление ограничений 441

1. [Поиск с возвратом 442](#bookmark308)

[Задача о п ферзях 443](#bookmark309)

[Задача о гамильтоновом цикле 444](#bookmark310)

[Задача о сумме подмножества 445](#bookmark311)

[Общие замечания 447](#bookmark312)

[Упражнения 11.1 449](#bookmark313)

1. [Метод ветвей и границ 451](#bookmark315)

[Задача о назначениях 452](#bookmark316)

[Задача о рюкзаке 455](#bookmark317)

[Задача коммивояжера 458](#bookmark319)

[Упражнения 11.2 460](#bookmark320)

1. [Приближенные алгоритмы для Л’Р-сложных задач 461](#bookmark321)

[Приближенный алгоритм для решения задачи коммивояжера 463](#bookmark324)

[Приближенные алгоритмы для задачи о рюкзаке 468](#bookmark327)

[Упражнения 1,1.3 473](#bookmark288)

1. [Алгоритмы для решения нелинейных уравнений 475](#bookmark329)

[Метод деления пополам 476](#bookmark330)

[Метод секущих 480](#bookmark332)

[Метод Ньютона 481](#bookmark334)

[Упражнения 11.4 484](#bookmark294)

[Резюме 485](#bookmark267)

Эпилог 487

Приложение А. Формулы, использующиеся при анализе алгоритмов 491

[Свойства логарифмов 491](#bookmark343)

[Комбинаторика 491](#bookmark344)

[Важные формулы суммирования 492](#bookmark345)

[Правила работы с суммами 492](#bookmark347)

[Приближение суммы определенным интегралом 493](#bookmark348)

[Формулы для округлений снизу и сверху 493](#bookmark349)

[Разное 493](#bookmark351)

Приложение Б. Краткое руководство по рекуррентным соотношениям 495

[Последовательности и рекуррентные соотношения 495](#bookmark355)

[Методы решения рекуррентных соотношений 497](#bookmark357)

Распространенные типы рекуррентных соотношений в анализе

[алгоритмов 501](#bookmark0)

Список литературы 509

Указания к упражнениям 517

Предметный указатель 562

С глубочайшей признательностью Марии и Мириам

Предисловие

Самое ценное в научном или техническом образовании — это развитие универсального мыслительного аппарата, который будет служить вам на протяжении всей жизни.

— Джордж Форсайт (George Forsythe), “Что предпринять до прихода специалиста по вычислительной технике” (“What to do till the computer scientist comes”) (1968)

А

лгоритмы имеют первостепенное значение как в научной, так и в техни­ческой сфере. Осознание данного факта привело к появлению огромного количества книг, посвященных этому предмету. Вообще говоря, в плане представ­ления алгоритмов все книги можно разделить на две большие группы. В одной из них алгоритмы классифицируются в соответствии с типом решаемой задачи. Как правило, в таких книгах алгоритмам сортировки, поиска, обработке графов и т.п. посвящены отдельные главы. Преимущество такого подхода заключается в том, что он позволяет непосредственно оценить эффективность применения различных алгоритмов для решения задачи. Недостаток же состоит в том, что при таком подходе акцент делается на решении самой задачи, а не на методологии проектирования алгоритма.

При использовании второго, альтернативного подхода основное внимание уде­ляется методике проектирования алгоритма. В таких книгах алгоритмы, относя­щиеся к различным областям вычислительной техники, группируются, если при их проектировании использованы одинаковые подходы. И здесь я также разделяю сложившееся мнение [11] по поводу того, что подобная организация книги боль­ше всего подходит для основного курса, посвященного проектированию и анализу алгоритмов. Имеется несколько причин для того, чтобы сосредоточиться на ме­тодологии проектирования алгоритмов. Во-первых, учащиеся смогут применить ее при разработке алгоритмов для решения неизвестной задачи. Во-вторых, они смогут классифицировать все множество известных алгоритмов согласно лежа­щей в их основе идее проектирования. Основной целью образования в области информатики должно быть изучение общих идей проектирования алгоритмов, относящихся к различным прикладным областям. В конце концов, при изучении любой научной дисциплины основное внимание уделяется рассмотрению системы ее основополагающих понятий. В-третьих, по моему мнению, изучение методоло­гии проектирования алгоритмов имеет огромную важность, поскольку дает ключ к пониманию методики поиска общего решения задач в области информатики.

Существует несколько учебников, материал которых структурирован в соот­ветствии с упомянутой выше методологией проектирования алгоритмов (в частно­сти, [22, 54, 82]). На мой взгляд, все они имеют один недостаток: их авторы слепо следуют одной и той же классификации методик проектирования алгоритмов. Данную классификацию нельзя считать удачной, поскольку она имеет несколько серьезных недостатков как с теоретической, так и с образовательной точек зре­ния. Один из основных ее недостатков заключается в том, что она не позволяет классифицировать большое количество важных алгоритмов. Все это вынужда­ет авторов существующих учебников, отклоняясь от рассмотрения методологии проектирования, включать в них главы, посвященные решению конкретных за­дач. К сожалению, подобный уход от основной темы приводит к потере логики изложения курса и практически всегда запутывает учащихся.

Новая таксономия методологии проектирования алгоритмов

Описанные выше недостатки существующей системы классификации вынуди­ли меня разработать новую таксономию[[1]](#footnote-1) методологии проектирования алгорит­мов [74], которая и была положена в основу этой книги. Позволю себе перечислить основные преимущества новой таксономии.

* Новая таксономия является более полной, чем существующая система классификации. Она включает ряд стратегий — решение задачи “в лоб”, методом разделения, методом преобразования; компромисс между вре­менем выполнения алгоритма и объемом оперативной памяти, — кото­рые не так часто относят к важным примерам проектирования.
* Новая таксономия естественным образом охватывает большое количе­ство классических алгоритмов, которые традиционная система не в со­стоянии классифицировать. Достаточно привести лишь несколько на­званий: алгоритм Евклида, пирамидальная сортировка, дерево поиска, хеширование, топологическая сортировка, схема Горнера. В результате, появляется возможность представить стандартный набор классических алгоритмов в единообразной и понятной форме.
* Она естественным образом приспособлена к существующему разнооб­разию основных методик проектирования.
* Она лучше всего подходит для анализа эффективности с помощью аналитических методов (см. приложение Б).

Методология проектирования как стратегия решения общих задач

Описанные в книге методики проектирования алгоритмов, как правило, при­меняются для решения классических задач вычислительной техники. Однако здесь есть одно новшество. В книгу включен материал, посвященный алгорит­мам решения численных задач, описанный в рамках той же принятой структуры. (Включение этих алгоритмов было одобрено учебным планом по информатике за 2001 год Computing Curricula 2001 [29] — разработанным с учетом современных требований.) Тем не менее описанные в книге методики проектирования можно считать универсальным средством решения задач; их применение не ограничи­вается только традиционными задачами вычислительной техники и математики. Важность этого подтверждается двумя факторами. Во-первых, все больше ком­пьютерных приложений выходят за рамки традиционной области их примене­ния, поэтому можно надеяться, что такая тенденция сохраниться и в будущем. Во-вторых, основной целью университетского образования считается выработка у студентов навыков самостоятельного решения поставленных задач. Поэтому среди всех курсов, читаемых в рамках программы по информатике, именно курс, посвященный разработке и анализу алгоритмов, как нельзя лучше подходит для этой цели, поскольку он развивает у студентов специфические навыки решения задач. Нр это отнюдь не означает, что данный курс следует рассматривать как курс, посвященный решению общих задач. Тем не менее я считаю, что не сле­дует упускать представившуюся вам уникальную возможность изучении методик разработки и анализа алгоритмов. Чтобы достичь поставленной цели, в книгу включены примеры головоломок и игр, построенных на их основе. Хотя идея изучения алгоритмов на основе головоломок, конечно, не нова, в этой книге сде­лана попытка систематизировать этот процесс, не ограничиваясь несколькими стандартными примерами.

Педагогика учебника

При написании книги я не ставил перед собой цель упростить излагаемый ма­териал, но в то же время хотел сделать так, чтобы он был доступен для понимания большинству студентов во время самостоятельной работы. Поэтому ниже я приве­ду несколько отличительных особенностей книги, способствующих достижению моего замысла.

* Разделяя мнение Джорджа Форсайта (см. эпиграф), я постарался под­черкнуть основные идеи, лежащие в основе разработки и анализа ал­горитмов. Поэтому, отбирая конкретные алгоритмы для иллюстрации этих идей, я ограничил их круг и привел в книге описание только тех из них, которые лучше всего проясняют основные подходы к проек­тированию или методы анализа алгоритмов. К счастью, большинство классических алгоритмов удовлетворяет этому критерию.
* В главе 2, посвященной анализу эффективности, мы будем различать методы анализа нерекурсивных алгоритмов и методы, которые обыч­но используются для анализа рекурсивных алгоритмов. В главу также включен раздел, посвященный эмпирическому анализу и алгоритмам визуализации.
* В текст всех глав включены вопросы к читателю. Часть из них носит риторический характер, и заданы они только потому, что имеют непо­средственное отношение к излагаемой теме и могут рассеять сомнения читателей. Поэтому ответы на них даются сразу. Остальные вопросы предназначены для проверки полученных знаний. Правильный ответ на них является индикатором достаточного уровня знаний у читате­ля и сигналом к тому, что можно переходить к изучению следующих разделов книги.
* В конце каждой главы приведено краткое резюме, в котором подыто­жены основные понятия и результаты, рассмотренные в главе.
* В книгу включено порядка 600 упражнений. Часть из них — учебные. В остальных отрабатываются важные моменты, о которых шла речь в материале главы, или описываются алгоритмы, не упоминавшиеся ранее. В некоторых упражнениях используются ресурсы Internet. Ряд упражнений предназначен для подготовки читателя к восприятию ма­териала, описанного в последующих разделах книги.

image6

Упражнения, содержащие игры, головоломки и вопросы, построенные на их основе, отмечены специальной пиктограммой.

* В книге приведены советы и подсказки по решению всех упражне­ний. Дополнительную информацию читатели могут получить по адре­су http: //www. aw. сот/cssupport.

Необходимые условия для чтения книги

Для того чтобы понимать материал книги, читатель должен прослушать на­чальный курс по программированию и один из стандартных курсов по дискрет­ным структурам. Такого багажа знаний будет вполне достаточно для свободного освоения описанного здесь материала. Тем не менее основные структуры дан­ных, необходимые формулы суммирования и рекуррентные соотношений описаны в разделе 1.4, приложениях А и Б, соответственно. Численные методы использу­ются только в трех разделах (2.2, 10.4 и 11.4), и то в очень ограниченной степени. Поэтому если у читателя нет твердых знаний в области численных методов, он вполне может пропустить эти три раздела, и это не осложнит понимания осталь­ного материала книги.

Использование книги в учебном процессе

Материал этой книги можно использовать в качестве основы учебного кур­са, посвященного проектированию и анализу алгоритмов, с упором на методику проектирования алгоритмов. Его вполне достаточно для стандартного односе­местрового курса. Вообще говоря, часть материала глав с 3 по 11 можно смело пропустить, и это никак не скажется на понимании читателями последующих глав книги. Любую из частей книги можно отдать для самостоятельной работы. В частности, изучение разделов 2.6 и 2.7, посвященных, соответственно, эмпири­ческому анализу и алгоритмам визуализации, можно совместить с выполнением программных проектов.

В табл. приведен примерный план 80-часового (40-лекционного) односемест­рового учебного курса.

Примерный план 80-часового односеместрового учебного курса

Номер Тема Разделы

лекции

1. Введение 1.1-1.3

3, 4 Изучение основ; условные обозначения О, 0 и £) 2.1, 2.2

5 Математический анализ нерекурсивных алгорит- 2.3

мов

6, 7 Математический анализ рекурсивных алгоритмов 2.4, 2.5 и при­

ложение Б

1. Алгоритмы решения задач “в лоб” 3.1, 3.2 и раз­

дел 3.3

1. Поиск методом перебора 3.4

Номер Тема

**Разделы**

1. 4.3

4.4 или 4.5 или 4.6

1. 5.3

5.5

5.6

6.1-6.3

6.4

6.5

6.6

1.2-1 Л

три из 8.1-8.4 9.1-9.4

10.1

10.2

10.3

1. и 11.4 11.1

11.2

11.3

лекции

10-12 Алгоритмы разбиения: сортировки слиянием,

быстрой сортировки и двоичного поиска

13 Другие примеры алгоритмов декомпозиции

14-16 Алгоритмы уменьшения на единицу: сортировка

методом вставки, поиск в глубину и ширину, то­пологическая сортировка

1. Алгоритмы уменьшения на постоянное значение
2. Алгоритмы уменьшения на переменное значение

19-21 Упрощение экземпляра, предварительная сорти­

ровка, исключение Гаусса, сбалансированные де­ревья поиска

1. Изменение представления: множества и пирами­дальная сортировка
2. Изменение представления: схема Горнера и дво­ичный порядок
3. Приведение задачи

25-27 Компромисс между временем выполнения алго­

ритма и объемом оперативной памяти: поиск строк, хеширование, В-деревья

28-30 Алгоритмы динамического программирования

31-33 Жадные алгоритмы: Прима, Крускала, Дейкстры,

Хаффмана

1. Доказательство нижней границы
2. Деревья принятия решений
3. R, NP и NP-полные задачи
4. Алгоритмы численных методов
5. Поиск с возвратом
6. Метод ветвей и границ
7. Приближенные алгоритмы решения NP-сложных задач

Благодарности

Прежде всего хочу выразить признательность авторам других книг, посвящен­ных алгоритмам, чья интуиция ш идеи изложения материала прямо или косвенно помогли мне. Советы и критические замечания рецензентов, несомненно, суще­ственно улучшили содержание книги. Выражаю благодарность Саймону Беркови­чу (Simon Berkovich) из университета Джорджа Вашингтона (George Washington University), Ричарду Бори (Richard Borie) из университета штата Алабама (Univer­sity of Alabama), Дугласу М. Кэмпбеллу (Douglas М. Campbell) из университета Брайхем Янг (Brigham Young University), Бину Конгу (Bin Cong) из университета штата Калифорния в Фуллерт\*оне (California State University, Fullerton), Стиву Хо­меру (Steve Homer) из Бостонского университета (Boston University), Роланду Хаб- шеру (Roland Hubscher) из университета в Оберне (Auburn University), Сухамею Кунду (Sukhamay Kundu) из университета штата Луизиана (Louisiana State Univer­sity), Ши-Донг Лангу (Sheau-Dong Lang) из университета центральной Флориды (University of Central Florida), Джону С. Ласту (John С. Lusth) из Арканзаско- го университета (University of Arkansas), Джону Ф. Мейеру (John F. Meyer) из Мичиганского университета (University of Michigan), Стивену Р. Зейделю (Steven R. Seidel) из Мичиганского технологического университета, Али Шокофандеху (АН Shokoufandeh) из Дрексельского университета (Drexel University) и Джорджу X. Вильямсу (George Н. Williams) из колледжа в Юнионе.

Чрезвычайно признателен моей коллеге Мэри-Анджеле Папаласкари (Магу- Angela Papalaskari) за то, что она использовала текст рукописи этой книги при преподавании курса, посвященного алгоритмам, в университете г. Вилланова (Vil- lanova University) и внесла существенные улучшения в текст книги и упражнения. Она с большим энтузиазмом поддержала идею широкого использования в книге различных головоломок. Еще один коллега, Джон Матулис (John Matulis), также использовал текст рукописи книги в учебном процессе и высказал ценные заме­чания. Помощь в подготовке рукописи оказывал мой бывший студент Эндисва Хейнегт (Andiswa Heinegg). Его критические замечания позволили сделать более понятным содержимое книги и более стройным стиль ее изложения.

На протяжении нескольких последних семестров рукопись этой книги служила в качестве основного учебника для студентов университета в Вилланове, что, несомненно, причиняло им неудобство. Я восхищаюсь их терпением и благодарю за ценные советы и найденные ошибки и опечатки в тексте рукописи. Наверняка в книге остались ошибки, которые я, конечно же, никак не могу отнести на их счет: ошибки вполне могли появиться по моей вине при внесении правок уже после того, как студенты прослушали курс.

Хочу выразить благодарность сотрудникам издательства Addison-Wesley и их коллегам — всем тем, кто работал над моей книгой. Особенно хочу поблагодарить моего редактора Майте Суарез-Риваз (Maite Suarez-Rivas) и ее бывшего заме­стителя Лайзу Хог (Lisa Hogue) за то, что они разделяли и поддерживали мой энтузиазм в отношении этого проекта.

И в заключение мне хочется выразить глубокую признательность членам моей семьи. Ведь жить под одной крышей с супругом, который постоянно занят тем, что пишет книгу, гораздо тяжелее, чем писать саму книгу. Моя жена Мария пережила два таких года, посильно помогая мне. Помощь была огромной: более 250 ил­люстраций к этой книге являются ее творением! Моя дочь Мириам — большой авторитет для меня в области английской прозы. Она не только прочла большие куски этой книги, но и подобрала остроумные эпиграфы к главам. Огромное им спасибо!

Ананий Левитин

anany.levitinQvillanova.edu

Август 2002 г.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать любые ваши замечания, касающиеся книги.

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

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

E-mail: inf oQwilliamspublishing. com

WWW: http: //www. williamspublishing. com

Информация для писем из:

России: 115419, Москва, а/я 783

Украины: 03150, Киев, а/я 152

Глава

Введение

В основе мироздания лежат две идеи: исчисление и алгоритм. Исчис­ление, базирующееся на мощном аппарате математического анализа, является основой современной науки. Однако наука невозможна без ал­горитма, лежащего в основе современного мира.

— Дэвид Берлински (David Berlinski), “Появление алгоритма” (“The Advent of the Algorithm”) (2000)

З

ачем изучать алгоритмы? Для настоящего компьютерного профессионала су­ществует две причины: практическая и теоретическая. С практической точки зрения он должен иметь представление о стандартном наборе основных алго­ритмов, относящихся к разным областям вычислительной техники. Кроме того, он должен уметь разрабатывать новые алгоритмы и анализировать их эффек­тивность. С теоретической точки зрения процесс изучения алгоритмов, который иногда называют алгоритмикой (algorithmics), считается краеугольным камнем информатики (computer science). Дэвид Харел (David Harel) в своей великолеп­ной книге, остроумно озаглавленной Algorithmics: the Spirit of Computing1, пишет следующее:

Алгоритмика — это нечто большее, чем просто раздел информатики. Она является основой информатики и, положа руку на сердце, можно сказать, что она существенно повлияла на современную науку, технику и бизнес [[48], с. 6].

И даже если вы не являетесь студентом вуза, специализирующимся в компью­терных науках, существуют достаточно веские причины для того, чтобы заняться изучением алгоритмов. Попросту говоря, без алгоритмов невозможно создать ни одну компьютерную программу. Поэтому по мере того, как компьютерные про­граммы становятся все более значимыми практически во всех сферах профессио­нальной деятельности (да и личной жизни!), изучение алгоритмов становится все более и более важной задачей для широкого круга людей.

'Это название можно перевести на русский язык как **Алгоритмика: дух вычислений.** — **Прим. ред.**

Еще одна причина для изучения алгоритмов заключается в том, что этот про­цесс развивает у учащихся умение аналитически мыслить. В конце концов, алго­ритмы можно рассматривать как особый подход к решению задач, когда важны не столько сами ответы, сколько точные инструкции для их получения. Следователь­но, специальные методы проектировании алгоритмов можно считать стратегиче­ским планом решения задачи, который пригодится в любом случае, независимо от того, используется компьютер или нет. Само собой разумеется, что круг задач, которые могут быть решены с помощью какого-либо алгоритма, по существу зави­сит от точности алгоритмического мышления. Например, невозможно сформули­ровать алгоритм, который позволит прожить счастливую жизнь или стать богатым и знаменитым. С другой стороны, выдвинутое требование точности имеет важ­ное преимущество с точки зрения обучения. Дональд Кнут, один из выдающихся ученых в области информатики и истории алгоритмики, пишет следующее:

Хорошо обученный в области информатики специалист обязан знать, как работать с алгоритмами: как их создавать, изменять, понимать и ана­лизировать. Эти знания позволят не только писать хорошие компьютер­ные программы, но и станут основой универсального мыслительного аппарата, который окажет неоценимую помощь при постижении других наук, будь то химия, лингвистика, музыка и т.д. Причину этого можно объяснить следующим образом: часто говорят, что человек ничего не по­нимает, пока не объяснит это кому-то другому. Я бы перефразировал это так: человек глубоко не понимает предмет до тех пор, пока не научит этому компьютер, т.е. выразит что-либо в виде алгоритма... Попытка формализовать нечто в виде набора алгоритмов приводит к более глу­бокому пониманию сути вещей, чем при их осмыслении традиционным способом [[64], с. 9].

О том, что такое алгоритм речь пойдет в разделе 1.1. Там в качестве при­меров мы рассмотрим три алгоритма решения одной и той же задачи — поиска наибольшего общего делителя (НОД). Я выбрал эту задачу по трем причинам. Во-первых, она знакома практически каждому со времени обучения в средней школе. Во-вторых, она позволяет продемонстрировать важный принцип: то, что одну и ту же задачу можно решить с помощью нескольких алгоритмов. Вполне естественно, что эти алгоритмы отличаются по своей сути, уровню сложности и эффективности. В-третьих, один из этих алгоритмов вполне достоин того, что­бы быть представленным первым, — как по “возрасту” (впервые он был описан в известном трактате Евклида более двух тысяч лет тому назад), так и по неисчер­паемым возможностям и важности. Наконец, метод определения НОД, которому учат в школе, позволит продемонстрировать одно из важных условий, которому должен удовлетворять любой алгоритм.

В разделе 1.2 рассматривается решение алгоритмической задачи. Там мы об­судим несколько важных моментов, связанных с разработкой и анализом алго­ритмов. К различным аспектам решения алгоритмической задачи относятся как анализ задачи и средства выражения алгоритма, так и методы оценки его пра­вильности и эффективности. В этом разделе вы не найдете описания магического средства для создания алгоритма решения любой задачи. Должно быть уже понят­но, что такого средства попросту не существует. Тем не менее материал раздела 1.2 поможет вам систематизировать проектирование и анализ алгоритмов.

Раздел 1.3 посвящен некоторым типам задач, считающимся особенно важны­ми при изучении алгоритмов, и их применению. На самом деле существует ряд учебников, материал которых структурирован по типу решаемых задач. Однако мы придерживаемся иного мнения, с которым согласны многие преподаватели: структурирование материала на основе метода проектирования алгоритма — го­раздо важнее. Тем не менее очень важно разбираться в основных типах задач и не только потому, что они находят широкое применение в реальной жизни. В этой книге они используются для демонстрации особенностей метода проектирования алгоритмов.

В разделе 1.4 приведен обзор основных структур данных. Однако его следует рассматривать скорее как справочник, а не подробное руководство. Если же вам нужна детальная информация по этой теме, обратитесь к дополнительной лите­ратуре. Хороших книг, посвященных этой теме, написано довольно много, однако в большинстве из них акцент сделан на том или ином языке программирования.

1. Понятие алгоритма

Что же такое алгоритм? Универсального определения этого понятия нет, од­нако существует общее мнение по поводу того, что оно должно означать:

Алгоритм — это последовательность четко определенных инструкций, предназначенных для решения некоторой задачи. Другими словами, это последовательность команд, позволяющих получить из корректных входных данных требующиеся выходные данные за ограниченный про­межуток времени.

Это определение можно проиллюстрировать с помощью простой схемы (рис. 1.1).

Упомянув в приведенном выше определении слово “инструкции”, мы подразу­мевали, что существует некоторое абстрактное устройство (или человек), способ­ное распознать эти инструкции и выполнить предписываемые ими действия. На рис. 1.1 это устройство названо “вычислительным”. Однако следует иметь в виду, что до появления компьютеров под термином “вычислительное устройство” по­нимался человек, выполняющий числовые расчеты. Естественно, если речь идет о современности, под словом “вычислительное устройство” понимается компью­тер, т.е. популярное электронное устройство, которое практически повсеместно

Задача

|  |  |  |  |
| --- | --- | --- | --- |
| Входные | W | Вычислительное | W |
| данные |  | устройство |  |

Алгоритм

Выходные

данные

Рис. 1.1. Иллюстрация понятия алгоритма

вторглось в нашу жизнь. Тем не менее обратите внимание на то, что, хотя большая часть алгоритмов в конечном счете предназначена для реализации на компьютере, само понятие алгоритма никак не связано с этим допущением.

В этом разделе в качестве примеров, иллюстрирующих понятие алгоритма, мы рассмотрим три способа решения одной и той же задачи — поиска НОД двух целых чисел. Эти примеры помогут нам проиллюстрировать перечисленные ниже важные моменты.

* Каждый шаг алгоритма должен быть четко и однозначно определен. Это требование является обязательным и не должно нарушаться ни при каких обстоятельствах.
* Должны быть точно указаны диапазоны допустимых значений входных данных, которые обрабатываются с помощью алгоритма.
* Один и тот же алгоритм можно представить несколькими разными способами.
* Для решения одной и той же задачи может существовать несколько разных алгоритмов.
* В основу алгоритмов для решения одной и той же задачи могут быть положены совершенно разные принципы, что может существенно по­влиять на скорость решения этой задачи.

Обозначим функцию поиска НОД двух неотрицательных целых чисел тип (причем ш и п не могут одновременно равняться нулю) через gcd (771,71).[[2]](#footnote-2) По определению эта функция должна найти наибольшее целое число, которое де­лится без остатка как на тп, так и на п. Древнегреческий математик Евклид из Александрии (III в до н.э.), который прославился тем, что впервые систематиче­ски изложил курс геометрии, описал алгоритм решения этой задачи в одном из своих трудов под названием Начала. Выражаясь современным языком, алгоритм

Евклида основан на рекуррентном вычислении следующего равенства:

gcd (m, п) — gcd (n, т mod п).

Здесь выражение (т mod п) является остатком от деления т на п. Выполнение алгоритма заканчивается, когда выражение (ш mod п) становится равным нулю. Поскольку gcd (m, 0) = т (понятно, почему?), последнее полученное значение т будет также являться НОД исходных чисел тип.

Например, вычисление НОД пары чисел (60, 24) можно выполнить следую­щим образом:

gcd (60,24) = gcd (24,12) = gcd (12,0) = 12.

(Если этот алгоритм не произвел на вас впечатления, попытайтесь определить НОД двух больших чисел, например таких, которые используются в задаче 4 упражнения 1.1.)

Ниже приводится более структурированное описание рассматриваемого нами алгоритма.

Вычисление НОД чисел шип при помощи алгоритма Евклида Шаг 1 Если п = 0, вернуть т в качестве ответа и закончить работу; иначе пе­рейти к шагу 2.

Шаг 2 Поделить нацело шнапи присвоить значение остатка переменной г.

Шаг 3 Присвоить значение п переменной т, а значение г — переменной п. Пе­рейти к шагу 1.

В качестве альтернативы запишем тот же алгоритм в виде псевдокода.

**Алгоритм Euclid** (m, n)

// Алгоритм Евклида вычисляет значение функции gcd(m, п)

// Входные данные: два неотрицательных целых числа тип,

// которые не могут одновременно быть равны нулю

// Выходные данные: наибольший общий делитель чисел тип while п ф 0 do г <— т mod п т <— п п <— г return т

Можем ли мы убедиться, что в конечном счёте выполнение алгоритма Ев­клида завершится? Это следует из констатации следующего факта: на каждом шаге итерации значение второго числа пары (п) будет уменьшаться, причем, по определению, оно не может быть меньше нуля. В самом деле, новое значение числа п, получаемое на следующей итерации в результате вычисления выражения (m mod п), будет всегда меньше, чем предыдущее значение числа п. Следователь­но, рано или поздно значение второго числа пары станет равным 0, и выполнение алгоритма завершится.

Как и при решении большинства других задач, существует несколько алго­ритмов вычисления НОД. Давайте рассмотрим два других способа решения этой задачи. Первый из них основан на подборе наибольшего целого числа — такого, чтобы числа тип делились на него без остатка. Очевидно, что такой общий дели­тель не может быть больше наименьшего из чисел пары, которое можно записать как t = min {m,n}. Поэтому выполнение алгоритма можно начать с проверки того, делятся ли оба числа, т и п, на t без остатка. Если это так, то число t является ответом; если нет, нужно уменьшить значение t на единицу и снова вы­полнить проверку. (Можем ли мы убедиться, что в конечном итоге этот процесс завершится?). Например, для рассмотренной выше пары чисел (60,24), выполне­ние алгоритма начинается с проверки числа 24, затем — 23 и т.д. до тех пор, пока значение числа t не станет равным 12, после чего алгоритм должен завершить свою работу.

**Вычисление НОД чисел** тип **методом последовательного перебора**

Шаг 1 Присвоить значение функции min {m, п} переменной t.

Шаг 2 Разделить т на t. Если остаток равен нулю, перейти к шагу 3; иначе перейти к шагу 4.

Шаг 3 Разделить п на t. Если остаток равен нулю, вернуть t в качестве ответа и закончить работу; иначе перейти к шагу 4.

Шаг 4 Вычесть 1 из t. Перейти к шагу 2.

Обратите внимание, что в отличие от алгоритма Евклида, рассматриваемый нами алгоритм поиска НОД в описанной выше форме не будет корректно рабо­тать, если хотя бы один из его входных параметров равен нулю. Таким образом, этот пример позволяет показать, почему так важно явно определять диапазоны допустимых значений входных параметров алгоритма.

Третий способ поиска НОД должен быть вам знаком из курса средней школы.

**Вычисление НОД чисел** тип “школьным” **методом Шаг 1** Разложить на простые множители число т.

Шаг 2 Разложить на простые множители число п.

Шаг 3 Для простых множителей чисел тип, найденных на шаге 1 и 2, выделить

их общие делители. (Если р является общим делителем чисел тип

и встречается в их разложении на простые множители, соответственно, рт и рп раз, то при выделении нужно повторить это min {рт,Рп} раз.)

Шаг 4 Вычислить произведение всех выделенных общих делителей и вернуть его в качестве результата поиска НОД двух указанных чисел.

Таким образом, для рассмотренной выше пары чисел (60,24), получим:

60 = 2 • 2 • 3 - 5 24 = 2 • 2 • 2 • 3 gcd (60,24) = 2 • 2 • 3 = 12.

Ностальгия по школьным годам не должна помешать нам сделать вывод о том, что последний из рассмотренных способов поиска НОД гораздо сложнее и мед­леннее, чем алгоритм Евклида. (Методы определения и сравнения времени вы­полнения алгоритмов будут описаны в следующей главе.) Но даже не принимая во внимание его низкую эффективность, метод определения НОД, изучаемый в сред­ней школе, нельзя считать законным алгоритмом (по крайней мере в представлен­ном здесь виде). Вы спросите, почему? Все дело в том, что этапы разложения на простые множители не определены однозначно. Для их выполнения требуется иметь список простых чисел, а я почему-то уверен, что школьный учитель не объяснил вам, как составить такой список. Вы, конечно же, можете возразить, что это не повод для подобных придирок. Однако до тех пор, пока этот момент не будет прояснен, мы не сможем, например, написать на каком-либо из языков программирования программу, реализующую этот алгоритм. (Кстати, с этой точки зрения шаг 3 также недостаточно четко определен. Однако его неопределенность гораздо легче прояснить, чем этапы разложения на простые множители. И еще, как вы собираетесь выделять общие элементы из двух списков?)

Итак, подошло время рассмотреть несложный алгоритм генерации последова­тельности простых чисел, не превышающих произвольно заданного целого числа п. Вероятнее всего он был придуман в древней Греции и поэтому назван решетом Эратосфена (прим. 200 год до н.э.). Для начала составим список, содержащий последовательность целых чисел от 2 до п, из которого затем мы должны будем выбрать простые числа. Далее, на первом проходе алгоритма нужно удалить из списка все числа, которые делятся на 2, т.е. 4, 6 и т.д. Затем необходимо выбрать из списка следующий элемент (в данном случае это 3) и удалить из списка все числа, которые делятся на него. (В описываемой простой версии алгоритма существует небольшая накладка, поскольку некоторые из чисел, например 6, должны удалять­ся из списка более одного раза.) Для числа 4 не нужно выполнять специальный проход по списку, так как число 4 кратно 2, поэтому оно уже будет удалено из списка на первом проходе. (Точно так же в этом алгоритме не нужно выполнять проход и для всех других чисел, которые были удалены из списка на предыду­щих проходах.) Следующим числом, которое осталось в списке и используется на третьем проходе, является 5. Работа алгоритма продолжается так до тех пор, покав списке существуют числа, которые можно удалить. Числа, оставшиеся в списке после выполнения алгоритма, являются простыми.

В качестве примера использования описанного выше алгоритма рассмотрим процесс поиска простых чисел, не превышающих п = 25:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

2 3 5 7 9 11 13 15 17 19 21 23 25

2 3 5 7 11 13 17 19 23 25

2 3 5 7 11 13 17 19 23

Для нашего примера не требуется большего количества проходов по списку, поскольку из него на рассмотренных проходах были удалены все составные числа. Таким образом, в списке остались только упорядоченные простые числа, меньшие или равные 25.

Возникает общий вопрос: при каком наибольшем значении числа р в списке еще остаются кратные ему числа? Прежде чем ответить на него, заметим, что, если р является числом, кратные которому числа должны быть удалены из списка на текущем проходе, то начать рассмотрение следует с числа р ■ р, поскольку все меньшие кратные ему числа — 2р,..., (р — 1) р — уже были удалены из списка на предыдущих проходах. Это наблюдение позволит избежать удаления из списка одного и того же числа более одного раза. Очевидно, что р ■ р не должно быть больше, чем п, поэтому р не может превышать значения s/n, округленного до целого числа в нижнюю сторону. На математическом языке это записывается как |Уп], т.е. с помощью так называемой функции “пол” (округления вниз, floor). В приведенном ниже псевдокоде алгоритма предполагается, что для вычисления используется готовая функция [\/п\ ■ Как альтернативный вариант можно в ка­честве условия продолжения вычисления цикла проверять значение неравенства р х р ^ п.

**Алгоритм Sieve** (п) **(решето Эратосфена)**

// Реализация решета Эратосфена

// Входные данные: Положительное целое число п ^ 2

// Выходные данные: Массив L простых чисел, меньших или // равных п

for **р** <— 2 to **п** do

А\р] <- р

for р <— 2 to Lv^J do // См. абзац перед псевдокодом if А\р] ф 0 // Элементы, кратные р еще не были

j <— р \* р И удалены на предыдущих проходах

*А[з]* <- 0 j\*-J+P

while **j** ^ **п**

II Пометим этот элемент и все // последующие, кратные р, как удаленные

// Скопируем оставшиеся элементы массива А в массив // простых чисел L г <— О

for **р** <— 2 to гг do

**if** А\р] ф О

Щ — АЫ

г г + 1 return L

Итак, теперь можно объединить алгоритм решета Эратосфена с методом, ко­торый вы проходили в средней школе, получив совершенно законный алгоритм поиска НОД двух положительных чисел. Обратите внимание, что в этом алгорит­ме нужно предусмотреть частный случай, когда один или оба входных параметра равны 1. Поскольку математики не считают число 1 простым, то, строго говоря, этот метод и не должен работать в подобном случае.

Прежде чем закончить этот раздел, необходимо сделать еще одно важное заме­чание. Примеры, рассмотренные в этом разделе, не относятся к математическим задачам, несмотря на то, что большинство этих алгоритмов широко использу­ются, а некоторые даже реализованы в виде компьютерных программ. Умение раскладывать задачи на алгоритмы пригодится в решении повседневных рутин­ных проблем, возникающих как на работе, так и дома. Вполне возможно, что, осознав важность алгоритмов в современном мире, вам захочется еще больше узнать об этом замечательном двигателе информационного века.

Упражнения 1.1

1. Ознакомьтесь с учениями Аль Хорезми — человека, от имени кото­рого произошло слово “алгоритм”. В частности, узнайте, что общего в словах “алгоритм” и “алгебра”.
2. Известно, что основной целью деятельности патентной системы США является содействие “прикладным искусствам”. Как вы думаете, можно ли запатентовать алгоритмы в этой стране?
3. а) Составьте подробную инструкцию (как это делается при описании

алгоритма) вашего возвращения из института домой, б) Составьте подробный рецепт приготовления вашего любимого блю­да (как это делается при описании алгоритма).

1. а) Найдите значение функции gcd (31415,14142) при помощи алгорит­

ма Евклида.

б) Оцените, во сколько раз быстрее вычисляется значение функции gcd (31415,14142) при помощи алгоритма Евклида по сравнению

с алгоритмом последовательного перебора чисел сверху вниз от зна­чения min {m, п} до значении gcd (m, п).

1. Докажите, что для любых двух положительных чисел тип выполня­ется равенство gcd (m, n) = gcd (n, m mod n).
2. Что произойдет, если при использовании алгоритма Евклида первое из чисел будет меньше второго? Каково будет максимальное время выполнения алгоритма при подстановке таких входных данных?
3. а) При каких значениях входных параметров т и п в алгоритме Ев­

клида выполняется минимальное количество операций деления, при условии, что 1 < ш, п < 10? б) При каких значениях входных параметров тип в алгоритме Евкли­да выполняется максимальное количество операций деления, при условии, что 1 < тп, п < 10?

1. а) В своем трактате Евклид описал алгоритм поиска НОД, в котором

image7

вместо операций целочисленного деления используются операции вычитания. Запишите этот вариант алгоритма Евклида на псевдо­коде.

б) Игра Евклида (см. [20]). Напишите на доске два неравных поло­жительных числа. В игре участвуют два игрока. Игроки должны по очереди писать на доске положительное число, равное разности двух чисел, уже написанных на доске. Причем это число должно быть новым, т.е. оно не должно уже находиться на доске. Проиг­равшим считается тот игрок, кто не сможет написать новое число. Какой игрок, по вашему мнению, имеет преимущество в этой игре: первый или второй?

1. Придумайте алгоритм вычисления функции [у/п\.

10. В усовершенствованном алгоритме Евклида определяется не только НОД двух положительных чисел тип (обозначим его через d), но и два числа х и у (не обязательно положительные), такие, что выполняется равенство тх + пу = d.

а) Найдите описание усовершенствованного алгоритма Евклида (на­пример, см. [[65], с. 28]) и реализуйте его в виде программы на своем любимом языке программирования.

б) Видоизмените программу так, чтобы она могла находить целочис­ленное решение диофантова уравнения ах + by = с для любых целых коэффициентов а, b и с. (Учтите, что для некоторых комби­наций коэффициентов уравнение может не иметь решения.)

1. Основы решения алгоритмической задачи

Начнем этот раздел с повторения важного вывода, сделанного во введении к этой главе:

Алгоритмы можно считать процедурным решением задач.

Причем эти решения являются не столько ответами, сколько точно опреде­ленными инструкциями для получения ответов. Именно этот акцент на точности определения конструктивных процедур отличает информатику от других дисци­плин, в частности от теоретической математики, в которой обычно ограничива­ются доказательством существования решения и (иногда) исследованиями харак­теристик решения.

А теперь перечислим и кратко опишем последовательность этапов проектиро­вания и анализа алгоритмов (рис. 1.2).

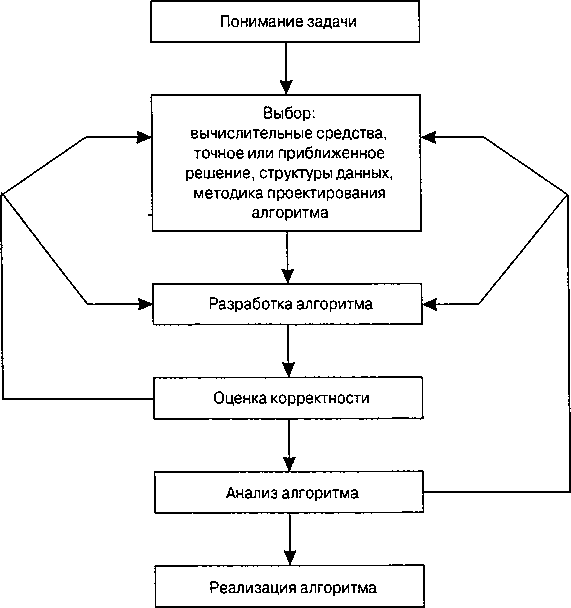


Рис. 1.2. Процесс проектирования и анализа алгоритмов

Понимание задачи

С практической точки зрения, прежде чем заняться проектированием алго­ритма, необходимо полностью понять поставленную перед вами задачу. Прочтите внимательно условие задачи и задайте вопросы, если вы чего-то не поняли. Про­считайте вручную несколько небольших примеров, продумайте частные случаи и при необходимости снова задайте вопросы.

Существует несколько типов задач, которые при создании компьютерных при­ложений встречаются чаще всего. Их обзор будет сделан в следующем разделе. Если перед вами поставлена одна из этих задач, для ее решения можно воспользо­ваться известным алгоритмом. Конечно, в процессе решения необходимо понять, как работает алгоритм, и оценить все его достоинства и недостатки, особенно если существует возможность выбора среди нескольких алгоритмов. Однако чаще все­го вы не сможете найти подходящий готовый алгоритм и должны будете создать собственный. Описанная в этом разделе последовательность этапов поможет вам в этом захватывающем, но не всегда легком деле.

Входные данные, обрабатываемые алгоритмом, определяют экземпляр задачи (problem’s instance), решаемой с помощью выбранного алгоритма. При этом очень важно точно указать границы примеров, которые должны учитываться в алгорит­ме. (Вспомните, насколько отличаются границы примеров для трех алгоритмов поиска НОД, рассмотренных в предыдущем разделе.) Если этого не сделать, то алгоритм будет прекрасно работать для большинства значений входных пара­метров, однако при подстановке отдельных “граничных” значений вы получите некорректные результаты. Позвольте напомнить, что корректным считается такой алгоритм, который выдает абсолютно правильный результат для всех (а не только для большинства!) заранее оговоренных значений входных данных.

Не стоит недооценивать первый этап процесса решения алгоритмической зада­чи, поскольку в противном случае, как правило, приходится многое переделывать.

Определение возможностей вычислительного устройства

Полностью уяснив суть поставленной задачи, необходимо оценить возможно­сти вычислительного устройства, для которого создается алгоритм. Подавляющее большинство современных алгоритмов предназначено для создания на их основе программы, работающей на компьютере, сильно напоминающем по структуре ма­шину фон Неймана[[3]](#footnote-3). Суть этой архитектуры выражена в ее названии — машина с произвольным доступом (random-access machine, или RAM). Предполагалось, что команды должны последовательно выбираться из памяти и выполняться спе­циальным устройством, называемым центральным процессором, причем в каж­дый момент времени в машине Неймана могла выполняться только одна команда. Поэтому алгоритмы, разработанные для выполнения на такой машине, назвали последовательными (sequential algorithms).

Появление машины Неймана не могло остановить прогресс в области вычис­лительной техники. Вскоре были придуманы компьютеры, которые могли одно­временно (т.е. параллельно) выполнять несколько команд. Поэтому алгоритмы, разработанные для таких машин, назвали параллельными (parallel algorithms). Тем не менее изучение классических методов проектирования и анализа алго­ритмов для машины Неймана еще долго будет оставаться краеугольным камнем алгоритмики.

Задумывались ли вы когда-нибудь о том, насколько быстро ваш компьютер выполняет команды и какой у него объем оперативной памяти? Наверняка вы от­ветите “нет”, если до этого проектировали алгоритмы лишь с теоретической точки зрения. Как будет показано в разделе 2.1, большинство кибернетиков предпочита­ют изучать алгоритмы, не привязываясь к параметрам конкретной компьютерной системы. Ответ специалиста-практика наверняка будет зависеть от того, какую задачу перед ним поставили. Даже “медленные” по современным меркам ком­пьютеры на деле оказываются невероятно быстродействующими. Следовательно, проблема “медленного компьютера” не должна возникать для большинства ре­шаемых вами задач. Однако существует целый класс важных задач, которые по своей природе являются очень сложными, должны обрабатывать огромные мас­сивы данных или работать в жестких временных рамках. В подобных ситуациях непременно следует учитывать быстродействие компьютерной системы, на ко­торой будет работать реализация алгоритма, и доступный объем оперативной памяти.

Выбор между точным или приближенным методом решения задачи

Следующий принципиальный вопрос — выбор точного или приближенного метода решения задачи. В первом случае алгоритм называется точным (exact algorithm), а во втором — приближенным (approximation algorithm). Почему для решения задачи иногда выбираются приближенные алгоритмы? Во-первых, суще­ствуют задачи, которые нельзя решить точно. В качестве примера можно привести извлечение квадратного корня, решение нелинейных уравнений или вычисление определенных интегралов. Во-вторых, существующие алгоритмы для точного ре­шения задачи могут быть недопустимо медленными, если ее сложность достаточ­но высока. Наиболее известной из таких задач является задача коммивояжера (traveling salesman problem), которая заключается в поиске кратчайшего маршрута между п городами. В главах 3, 10 и 11 будут приведены и другие примеры подоб­ных задач. В-третьих, приближенный алгоритм может являться частью другого, более сложного алгоритма, с помощью которого задача решается точно.

Выбор подходящих структур данных

В некоторых алгоритмах не требуется, чтобы входные данные были представ­лены в каком-то специфическом формате. Однако так бывает не всегда, более того, для работы многих алгоритмов требуются совершенно определенные структуры данных. Кроме того, некоторые из методов проектировании алгоритмов, о которых пойдет речь в главах 6 и 7, очень тесно связаны со структуризацией и реструкту­ризацией данных, определяющих экземпляры задачи. Много лет назад в уважае­мой всеми книге провозглашалась чрезвычайная важность алгоритмов и структур данных и их влияние на процесс программирования. Причем об этой важности го­ворило само за себя название этой книги: Algorithms + Data Structures = Programs [123]. В современном мире, где властвует объектно-ориентированное программи­рование, структуры данных остаются чрезвычайно важным элементом процесса проектирования и анализа алгоритмов. Обзор основных структур данных будет сделан в разделе 1.4.

Методы проектирования алгоритмов

Теперь, когда все составляющие решения алгоритмической проблемы нахо­дятся на своих местах, остается выяснить, как нужно проектировать алгоритмы решения поставленных перед вами задач. Это основной вопрос данной книги. Для ответа на него вам предлагается изучить несколько общих методов проектирова­ния алгоритмов. Что же такое метод проектирования алгоритма?

Метод проектирования алгоритма (algorithm design technique) (или “стратегия”, или “принцип”) — это универсальный подход, применяе­мый для алгоритмического решения широкого круга задач, относящихся к различным областям вычислительной техники.

Взглянув на оглавление этой книги, вы поймете, что основная часть ее глав посвящена именно отдельным методам проектирования алгоритмов. В них из­лагается несколько проверенных временем ключевых идей, используемых при разработке алгоритмов. Изучение этих методов в высшей степени важно по сле­дующим причинам.

Во-первых, они обеспечивают набор универсальных принципов, руководству­ясь которыми можно разработать алгоритмы решения новых задач, т.е. таких задач, для решения которых еще не существует достаточно хороших алгоритмов. Поэтому, перефразируя известную поговорку, можно сказать, что изучение подоб­ных методов сродни подаренной удочке, а не предложенной рыбе. Конечно, не все из этих универсальных методов подойдут для решения практических задач, с которыми вам предстоит столкнуться. Однако они представляют собой мощный набор средств, которые пригодятся вам в учебе и работе.

Во-вторых, алгоритмы являются краеугольным камнем информатики. Клас­сификация основополагающих понятий важна для любой науки, и информатика не является исключением. Изучение этих методов позволяет классифицировать алгоритмы согласно лежащему в их основе принципу проектирования. Поэтому они как нельзя лучше подходят и для классификации, и для изучения алгоритмов.

Методы представления алгоритмов

После того как алгоритм спроектирован, нужно представить его в каком-либо виде. В разделе 1.1 в качестве примера мы описали алгоритм Евклида — как сло­вами (т.е. в свободной форме в виде последовательности выполняемых действий), так и в виде псевдокода. В наши дни для представления алгоритмов чаще всего используются эти две формы.

Безусловно, использование естественного языка для описания алгоритма име­ет очевидное преимущество. Тем не менее присущая любому такому языку неопре­деленность иногда затрудняет лаконичное и понятное описание алгоритмов. Как бы то ни было, умение словесно описать алгоритм в процессе его изучения нико­гда не будет лишним.

Псевдокод (pseudocode) представляет собой смесь одного из естественных языков[[4]](#footnote-4) и конструкций, характерных для языка программирования. Описание ал­горитма на псевдокоде обычно является более точным по сравнению с естествен­ным языком. Кроме того, в-результате его использования часто получается более компактная запись алгоритма. Неожиданностью является то, что специалисты до сих пор так и не приняли какую-либо форму псевдокода в качестве стандарта. Поэтому в разных книгах можно встретить различные “диалекты” этого языка. К счастью, эти диалекты настолько близки, что любой человек, владеющий каким- либо современным языком программирования, сможет без особого труда в них разобраться.

Используемый в этой книге диалект псевдокода не должен представлять осо­бых затруднений для читателя. Ради простоты изложения мы опустили операторы определения переменных, а для обозначения области действия таких операторов, как for, if и while, в тексте псевдокода использованы отступы. Как вы уже, на­верное, заметили при чтении предыдущего раздела, для обозначения операции присваивания мы используем стрелку, <—, а комментариев — две косые черты подряд; //.

На заре развитии вычислительной техники основным способом представле­ния алгоритмов были блок-схемы (flowchart), т.е. чертежи, состоящие из после­довательности соединенных стрелками геометрических фигур, с помощью кото­рых описывался каждый шаг выполнения алгоритма (примером может служить рис. 1.2). Однако подобный способ представления сочли неудобным, особенно в случае больших и сложных алгоритмов, поэтому в современной литературе он не используется.

Современные средства вычислительной техники еще не достигли такого уров­ня, чтобы описание алгоритма (будь-то его словесное описание или псевдокод) можно было непосредственно ввести в компьютер. Поэтому пока приходится вручную преобразовывать алгоритмы в компьютерные программы и записывать их на одном из доступных языков программирования. Таким образом, подобную программу можно считать еще одним средством представления алгоритма, хотя, строго говоря, программа является реализацией конкретного алгоритма.

Оценка корректности алгоритма

После представления алгоритма в какой-либо форме, необходимо оценить его корректность (correctness). Это означает, что вы должны доказать, что выбран­ный алгоритм за ограниченный промежуток времени выдает требуемый результат для любых корректных значений входных данных. Например, корректность алго­ритма Евклида для вычисления НОД двух чисел, тип, следует из правильности равенства gcd (m, п) = gcd (n, т mod п), которое, в свою очередь, должно быть доказано (см. упражнение 1.1.5), а также из простого наблюдения, что второе чис­ло на каждой итерации будет все время уменьшаться, и по достижении им нуля, выполнение алгоритма прекращается.

Доказать корректность одних алгоритмов очень легко, других — невероятно сложно. Универсальным методом доказательства корректности алгоритма считает­ся метод математической индукции. Дело в том, что алгоритмы по своей природе являются итеративными и описываются в виде последовательности пошаговых инструкций, которые как раз и используются при доказательстве методом индук­ции. Стоит отметить, что, хотя отслеживание быстродействия алгоритма с по­мощью нескольких наборов тщательно подобранных входных данных приводит к очень полезным результатам, подобную проверку нельзя считать убедительной при доказательстве корректности алгоритма. Однако доказательством некоррект­ной работы алгоритма может служить лишь один набор входных данных, при обработке которых получается неправильный результат. Если некорректная ра­бота алгоритма будет доказана, придется либо несколько видоизменить его, не выходя за рамки принятых структур данных, методов проектирования и т.п., ли­бо решать проблему более кардинально, пересмотрев принятые ранее принципы и подходы (см. рис. 1.2).

Оценка корректности приближенных алгоритмов менее тривиальна, чем их точных аналогов. При этом нужно показать, что погрешность получаемых в ре­зультате работы алгоритма выходных данных не превышает заранее установлен­ных пределов. Примеры подобных оценок приведены в главе 11.

Анализ алгоритма

Обычно создатели алгоритмов стараются, чтобы они удовлетворяли несколь­ким требованиям. После проверки корректности алгоритма одной из самых важ­ных характеристик является оценка его эффективности. На практике существует два вида оценки эффективности алгоритма: временная и пространственная. Вре­менная эффективность (time efficiency) является индикатором скорости работы алгоритма. Что касается пространственной эффективности (space efficiency), то эта оценка показывает количество дополнительной оперативной памяти, необ­ходимой для работы алгоритма. Общая идея и конкретные методы анализа эффек­тивности алгоритмов будут рассмотрены в главе 2.

Еще одной важной характеристикой алгоритма является его простота (sim­plicity). В отличие от эффективности, которую можно точно определить и оце­нить с помощью строгих математических выражений, простота алгоритма чем-то напоминает человеческую красоту, понятие которой настолько субъективно, что выработать объективные критерии ее оценки весьма непросто. Например, боль­шинство людей считают, что алгоритм Евклида поиска НОД двух чисел проще, чем тот, которому нас учили в школе, но это вовсе не означает, что он понятнее. В то же время алгоритм Евклида проще алгоритма последовательного перебора целых чисел. Тем не менее простота является важной характеристикой алгорит­ма, и к ней нужно стремиться. Вы спросите, почему? Дело в том, что простые алгоритмы легче понять и запрограммировать. Следовательно, в полученной про­грамме будет содержаться меньше ошибок. Кроме того, в простоте существует неоспоримая эстетическая привлекательность. Ну и наконец, простые алгоритмы зачастую более эффективны, чем их сложные аналоги. К сожалению, последнее утверждение не всегда выполняется. В подобных случаях необходимо руковод­ствоваться здравым смыслом и принимать компромиссные решения.

Следующей важной характеристикой алгоритма является его общность, или универсальность (generality). По сути, здесь можно выделить два момента: общ­ность задачи, для решения которой разработан алгоритм, и диапазон допустимых значений его входных данных. По поводу первого замечания следует сказать, что иногда легче разработать алгоритм для решения общей задачи, чем заниматься поиском решения ее частного случая. В качестве примера рассмотрим такую за­дачу: определить, являются ли два целых числа взаимно простыми. Напомним, что два числа считаются взаимно простыми, если у них нет общих делителей, кроме 1. В данном случае легче разработать алгоритм для решения общей зада­чи вычисления НОД двух целых чисел. Затем, чтобы решить исходную задачу, достаточно подставить заданные числа в функцию gcd и проверить, равно ли ее значение 1. Однако бывают случаи, когда разработка более общего алгоритма нежелательна, затруднена или даже невозможна. Например, излишне выполнять сортировку всего списка, содержащего п чисел, чтобы определить их медиану, поскольку достаточно просто найти значение m-го наименьшего элемента, где т = [п/2]. Еще один пример: известно, что стандартную формулу для поиска корней квадратного уравнения нельзя обобщить для поиска корней полиномов более высоких степеней.

Что касается диапазона допустимых значений входных данных алгоритма, то здесь нужно отметить следующее. При разработке алгоритма нужно учитывать, что диапазон изменения значений входных параметров может колебаться в очень широких пределах и должен естественным образом соответствовать решаемой за­даче. Например, неестественно в алгоритме поиска НОД исключать из рассмотре­ния значения входных параметров, равные 1. С другой стороны, хотя стандартную формулу корней квадратного уравнении можно использовать и в случае комплекс­ных коэффициентов, при принятом уровне обобщения этот случай исключают из рассмотрения, поскольку он должен оговариваться отдельно.

Если вас не устраивает эффективность, простота или общность алгоритма, придется снова взять в руки карандаш и перепроектировать алгоритм. И даже если анализ алгоритма привел к положительным результатам, имеет смысл по­искать другое алгоритмическое решение задачи. Достаточно вспомнить три алго­ритма определения НОД, рассмотренных в предыдущем разделе. Вообще говоря, не стоит надеяться, что вы с первой попытки найдете оптимальное решение за­дачи. Самое меньшее, что можно сделать, попытаться оптимизировать созданный алгоритм. Например, мы внесли несколько улучшений в реализацию алгоритма решета Эратосфена, по сравнению с той, что была описана в разделе 1.1. (Мо­жете ли вы назвать их не задумываясь?) Всегда помните высказывание Антуана де Сент-Экзюпери (Antoine de Saint-Exupery), известного французского писате­ля, летчика и авиаконструктора: “Конструктор достигает совершенства не тогда, когда ему больше нечего добавить к своему детищу, а тогда, когда больше ничего удалить”.[[5]](#footnote-5)

Кодирование алгоритма

Большинство алгоритмов рассчитано на то, что в конечном итоге они будут превращены в компьютерную программу. В процессе написания программы на

основе алгоритма может возникнуть множество подводных камней. Главная опас­ность заключается в том, что при переводе алгоритма на машинный язык могут быть внесены ошибки, либо окажется, что сама программа написана крайне неэф­фективно. Поэтому некоторые авторитетные специалисты считают, что до тех пор, пока корректность компьютерной программы не будет доказана с помощью стро­гих математических выкладок, ее нельзя считать правильной. Они даже разрабо­тали специальные методы для выполнения подобных оценок (см. [47]). Однако пока эти методы формальной верификации удается применить только к очень маленьким программам. На практике корректность компьютерных программ все еще проверяется с помощью тестирования. Процесс тестирования является скорее искусством, чем наукой, но это отнюдь не означает, что здесь нечему поучиться. Тестированию и отладке посвящено довольно много хороших книг, однако их основную мысль можно сформулировать так: в процессе реализации алгоритма программа должна быть тщательно протестирована и отлажена.

Следует также заметить, что в этой книге мы всегда предполагаем, что диа­пазоны значений входных данных алгоритмов не выходят за заранее оговоренные рамки, поэтому их не нужно проверять. Однако при реализации алгоритмов в виде реально работающих прикладных программ такая проверка просто необходима.

Само собой разумеется, что корректная реализация алгоритма в виде про­граммы является необходимым, но недостаточным условием, поскольку мощь алгоритма можно свести на нет неэффективной реализацией. Современные ком­пиляторы до некоторой степени позволяют застраховаться от этого, особенно при использовании режима оптимизации кода. Тем не менее следует знать о таких стандартных вещах, как вынесение операторов вычисления инвариантного выра­жении (т.е. такого выражейия, которое не изменяется в цикле) за пределы цикла, выделение общих подвыражений, замена медленных операторов их более быст­рыми аналогами и т.п. (Хорошее описание методов оптимизации кода программы и других вопросов, связанных с кодированием алгоритмов, можно найти в [15] и [61].) Как правило, применение подобных методов оптимизации может ускорить выполнение программы только на небольшой постоянный множитель, тогда как использование более эффективного алгоритма иногда ускоряет выполнение про­граммы на несколько порядков. Когда алгоритм окончательно выбран, повышение производительности реализующей его программы на 10-50% считается хорошим результатом.

После создании работающей версии программы можно выполнить дополни­тельный эмпирический анализ лежащего в ее основе алгоритма. Для этого нужно зафиксировать время выполнения программы при разных значениях входных дан­ных, а затем проанализировать полученный результат. Достоинства и недостатки данного метода анализа алгоритмов будут описаны в разделе 2.6.

И в заключение позвольте еще раз подчеркнуть основную идею процесса проектирования и анализа алгоритмов, показанного на рис. 1.2.

Хороший алгоритм получается, как правило, в результате кропотливой циклической работы, связанной с возможными переделками.

Даже если вам крупно повезет и вы придумаете алгоритм, который на первый взгляд кажется идеальным, все равно попытайтесь проанализировать, нельзя ли его еще в чем-то улучшить. Как ни странно, эта мысль не такая уж и плохая, поскольку позволяет получить истинное удовольствие от конечного результата. (Да, да! Кстати я собирался назвать эту книгу как The Joy of Algorithms.[[6]](#footnote-6)) С другой стороны, нужно уметь вовремя останавливаться. Что касается реальной жизни, то здесь обычно критерий остановки один — срок сдачи проекта или долготерпение вашего начальника, в зависимости от того, что быстрее закончится. В общем вывод такой: достижение идеала — слишком дорогое удовольствие, которое, к тому же, не всегда нужно. Проектирование алгоритма является достаточно сложной инженерной задачей, связанной с принятием компромиссных решений в условиях ограниченного доступа к ресурсам, одним из которых является время работы проектировщика.

С академической точки зрения затронутый в этом разделе вопрос связан с про­ведением интересного, но, как правило, очень сложного исследования оптималь­ности (optimality) алгоритма. Как ни странно, этот вопрос не относится к эффек­тивности самого алгоритма, а связан со сложностью решаемой с его помощью задачи. То есть необходимо выяснить, какое минимальное количество усилий нужно затратить, чтобы решить стоящую перед вами задачу с помощью произ­вольного алгоритма? Для некоторых задач ответ на этот вопрос найден. Например, в любом алгоритме сортировки элементов массива методом сравнения необходи­мо выполнить порядка п log2 п операций сравнения, где п — размерность массива (см. раздел 10.2). Однако для многих кажущихся простыми задач, типа перемно­жения матриц, ученым до сих пор так и не удалось найти окончательного ответа.

Еще один важный вопрос, возникающий при решении алгоритмической про­блемы, заключается в том, можно ли решить задачу вообще с помощью какого- либо алгоритма? В этой книге мы не будем обсуждать задачи, не имеющие ре­шения, наподобие поиска вещественных корней квадратного уравнения с отри­цательным дискриминантом. В подобных случаях алгоритм должен возвращать специальное значение, являющееся признаком того, что задача не имеет решения. Мы не будем также рассматривать неоднозначно определенные задачи. Речь идет о таких задачах, решение которых нельзя найти с помощью любого алгоритма, хотя у них может быть простой ответ — да или нет. Пример такой задачи будет приведен в разделе 10.3. К счастью, подавляющее большинство подобных задач может быть решено с помощью некоторого алгоритма.

И в заключение этого раздела хотелось бы развеять ваши сомнения по поводу того, что проектирование алгоритмов — довольно скучное занятие. Отчасти они могли появиться при взгляде на рис. 1.2, где все четко разложено по полочкам и на первый взгляд нет никакой свободы для творчества. Это абсолютно не соот­ветствует действительности! Изобретение (или поиск?) алгоритмов — творческий и необычайно захватывающий процесс, и я надеюсь, что книга убедит вас в этом.

Упражнения 1.2

1. Какую из перечисленных ниже формул можно использовать в каче­стве алгоритма для вычисления площади треугольника, длина сторон которого выражена положительными числами а, b и с?

а) S = ***у/р(р -*** *а)****(р- Ъ)(р -*** с), где ***р =*** *(а*+ ***Ъ*** + с)/2;

б) S = TjbcsinA, где А — угол между сторонами b и с;

в) S = \aha, где ha — высота треугольника, опущенная на сторону а.

1. Запишите на псевдокоде алгоритм поиска вещественных корней квад­ратного уравнения ах[[7]](#footnote-7) + Ьх + с = 0, где а, b и с — произвольные вещественные коэффициенты. (Подразумевается, что для поиска квад­ратного корня вы можете воспользоваться функцией sqrt (х).)
2. Опишите стандартный алгоритм преобразовании положительного де­сятичного числа в двоичное:

а) словами;

б) на псевдокоде.

1. Опишите алгоритм работы с банкоматом при получении денег с карточ­ки (если, конечно, она у вас есть). (Опишите алгоритм либо словами, либо на псевдокоде — как вам больше нравится.)
2. а) Может ли быть точно решена задача вычисления числа 7г?

б) Сколько экземпляров имеет данная задача?

в) Поищите алгоритм решения этой задачи в World Wide Web.

1. Приведите пример задачи, отличной от поиска НОД, для решении ко­торой существует несколько алгоритмов. Какой из этих алгоритмов проще? Какой эффективнее?
2. Проанализируйте приведенный ниже алгоритм поиска минимальной разницы между двумя элементами массива чисел.

Алгоритм MinDistance (А [0..п — 1])

// Входные данные: массив чисел А[0..п — 1]

// Выходные данные: минимальная разница между двумя // элементами массива А

dmin <— оо for ъ <— 0 to 77/ — 1 do for j 0 to n — 1 do

if г ф j and |A[i\ — A[j]| < dmin dmin <— | А [г] — A[j] | return dmin

Можете ли вы усовершенствовать алгоритм решения этой задачи, и ес­ли да, то какое количество изменений вы можете в него внести? (При

необходимости можно выбрать другой алгоритм. Если вы не можете улучшить предложенный алгоритм, то можете ли вы усовершенство­вать его реализацию?)

1. Одна из самых известных книг, посвященная алгоритмическому реше­нию задач, озаглавленная How to Solve It, написана американским ма­тематиком венгерского происхождения Джорждем Пойа (George Polya) (1887-1985) [89]. Пойа обобщил выдвинутые им идеи в виде резю­ме, состоящего из четырех пунктов. Найдите это резюме в Web (или, что еще лучше, прочтите саму книгу) и сравните его с предложенным планом решения алгоритмической задачи, который мы рассматривали в разделе 1.2. Что у них общего? В чем различия?
2. Важные типы задач

Среди огромного количества задач, встречающихся в вычислительной тех­нике, можно выделить несколько типов, которым ученые всегда уделяли особое внимание. Подобный интерес вызван либо практическим значением задачи, либо какими-то ценными ее свойствами, представляющими особый интерес для ис­следования. К счастью, в большинстве случаев эти причины взаимосвязаны, что только усиливает интерес ученых.

В этом разделе кратко рассматриваются наиболее важные типы задач:

* сортировка;
* поиск;
* обработка строк;
* задачи из теории графов;
* комбинаторные задачи;
* геометрические задачи;
* численные задачи.

Эти задачи будут использоваться в последующих главах книги для иллюстра­ции различных методов проектирования и анализа алгоритмов.

Сортировка

Задача сортировки (sorting problem) заключается в упорядочении заданного списка каких-либо элементов в возрастающем порядке[[8]](#footnote-8). Само собой разумеется, что для определенности задачи структура этих элементов списка должна поз­волять их упорядочить. (В подобных случаях математики говорят, что между элементами должны существовать отношения, допускающие полное упорядоче­ние.) На практике обычно требуется отсортировать по возрастанию список чисел, расположить символы и строки в алфавитном порядке или, что наиболее важ­но, упорядочить набор записей, содержащих различную информацию, наподобие той, что хранится в папках со сведениями об учащихся школы, в читательских формулярах библиотеки, в отделе кадров о сотрудниках организации. В случае набора записей необходимо выбрать фрагмент записи, содержащий данные, по которым будет осуществляться сортировка. Например, набор записей о студентах можно отсортировать по фамилии, идентификационному номеру или по среднему баллу. Специально отобранный фрагмент данных называется ключом (key). Спе­циалисты по информатике часто говорят о сортировке списка ключей, даже если элементы этого списка являются не записями, а, скажем, целыми числами.

Зачем может понадобиться отсортированный список? Ну, скажем затем, чтобы облегчить поиск ответов на ряд вопросов, связанных со списком. Наибольшую важность при этом имеет быстрота поиска информации. Вот почему словари, телефонные справочники, списки учащихся и т.п. всегда упорядочиваются по алфавиту. В разделе 6.1 будут приведены другие примеры, в которых используется предварительно отсортированный список. Аналогично сортировка используется также как вспомогательный этап в некоторых важных алгоритмах, относящихся к другим предметным областям, например к геометрии.

К настоящему моменту специалистами по вычислительной технике разрабо­таны десятки алгоритмов сортировки. По сути, выдумывание нового алгоритма сортировки можно сравнить с изобретением пресловутого велосипеда. Тем не менее мы с гордостью хотим заявить, что поиск лучших вариантов велосипеда (т.е. алгоритмов сортировки) продолжается. Подобная настойчивость может быть оправданна, если принять во внимание следующие факты. С одной стороны, су­ществует несколько хороших алгоритмов сортировки, в которых для сортировки произвольного массива из п элементов используется nlog2n операций сравне­ния. С другой стороны, нет алгоритма, который бы выполнял сортировку методом сравнения всего значения ключа (а не, скажем, небольшой части ключа) за суще­ственно меньшее количество операций, чем было указано выше.

Существует причина, сдерживающая разработку новых алгоритмов сортиров­ки. Несмотря на то что часть алгоритмов оказывается лучше остальных, пока еще не придуман универсальный алгоритм сортировки, который бы наилучшим образом подходил для всех случаев. Одни из существующих алгоритмов являют­ся простыми, но сравнительно медленными, другие работают быстрее, но более сложны. Некоторые из алгоритмов хорошо работают с неупорядоченными дан­ными, тогда как для других нужно, чтобы списки были частотно отсортированы. Часть алгоритмов пригодна только для сортировки списков данных, расположен­ных в оперативной памяти (т.е. в памяти с быстрым доступом), тогда как другие можно применить для сортировки больших массивов данных, расположенных на внешних носителях данных (магнитном диске, ленте и т.д.).

Два свойства алгоритмов сортировки заслуживают особого внимания. Алго­ритм сортировки называется устойчивым (stable), если в нем сохраняется от­носительный порядок любых двух равных элементов, находящихся во входном списке. Другими словами, если во входном списке есть два одинаковых элемен­та, номера которых равны г и j, причем г < j, то в отсортированном списке, где их номера будут, соответственно, равны г' и f, будет выполняться отноше­ние г1 < f. Это свойство может пригодиться в случае, если, например, требуется отсортировать упорядоченный в алфавитном порядке список учащихся согласно их успеваемости (среднему баллу). В случае применения устойчивого алгоритма будет получен список, в котором учащиеся с одинаковым средним баллом будут упорядочены по алфавиту. Вообще говоря, алгоритмы сортировки методом об­мена значениями ключей, расположенными на значительном расстоянии друг от друга, не являются устойчивыми, однако обычно они работают быстрее. Ниже в этой книге мы покажем, как это общее утверждение применяется к основным алгоритмам сортировки.

Второе важное свойство алгоритмов сортировки связано с количеством допол­нительной оперативной памяти, необходимой для работы алгоритма. Алгоритм называют обменным (in place), если для его работы не требуется дополнитель­ная оперативная память, кроме случаев возможного использования нескольких дополнительных ячеек памяти. Многие важные алгоритмы сортировки относятся к классу обменных, а другие, не менее важные — нет.

Поиск

Задача поиска связана с нахождением заданного значения, называемого клю­чом поиска (search key), среди заданного множества (или мультимножества[[9]](#footnote-9)). Существует огромное количество алгоритмов поиска, так что есть из чего выби­рать. Их сложность варьируется от самых простых алгоритмов поиска методом последовательного сравнения, до чрезвычайно эффективных, но ограниченных алгоритмов бинарного поиска, а также алгоритмов, основанных на представлении базового множества в иной, более подходящей для выполнения поиска форме. По­следние из упомянутых здесь алгоритмов имеют особое практическое значение, поскольку применяются в реально действующих приложениях, выполняющих вы­борку и хранение массивов информации в огромных базах данных.

Для решения задачи поиска также не существует единого алгоритма, который бы наилучшим образом подходил для всех случаев. Некоторые из алгоритмов выполняются быстрее остальных, но для их работы требуется дополнительная оперативная память. Другие выполняются очень быстро, но их можно применять только для предварительно отсортированных массивов, и т.п. В отличие от ал­горитмов сортировки в алгоритмах поиска нет проблемы устойчивости, но при их использовании могут возникать другие сложности. В частности, в тех прило­жениях, где обрабатываемые данные могут часто изменяться, причем количество изменений сравнимо с количеством операций поиска, поиск следует рассматри­вать в неразрывной связи с двумя другими операциями — добавления элемента в набор данных и удаления из него. В подобных ситуациях необходимо видоизме­нить структуры данных и алгоритмы так, чтобы достигалось равновесие между требованиями, выдвигаемыми к каждой операции. Кроме того, организации очень больших наборов данных с целью выполнения в них эффективного поиска (а так­же добавления и удаления элементов) представляет собой чрезвычайно сложную задачу, решение которой особенно важно с точки зрения практического приме­нения.

Обработка строк

В связи с быстрым увеличением в последнее время количества приложений, связанных с обработкой нечисловых данных, интерес ученых и специалистов- практиков все больше привлекают алгоритмы обработки строк. Строкой (string) называется последовательность символов, взятых из заранее определенного алфа­вита. Практический интерес представляют, например, текстовые строки, состоя­щие из букв, цифр и специальных символов; битовые строки, состоящие из нулей и единиц; последовательности генов, которые могут быть смоделированы с по­мощью строк символов, взятых из четырехсимвольного алфавита {А, С, G, Т}. Тем не менее стоит отметить, что алгоритмы обработки строк стали важны для вычислительной техники очень давно — с тех пор, как появились первые языки программирования и соответствующие им программы — компиляторы.

Существует одна специфическая задача, которая привлекла особое внимание специалистов по информатике. Речь идет о поиске заданного слова в строке текста. Ее назвали поиском подстрок (string matching). Для выполнения такого специ­фического поиска разработано несколько алгоритмов. В главе 3 мы опишем один очень простой алгоритм, а в главе 7 обсудим два алгоритма, созданных на основе замечательной идеи Р. Бойера (R. Boyer) и Дж. Мура (J. Moore).

Задачи из теории графов

Одной из самых старых и, пожалуй, наиболее интересных областей алгоритми- ки является обработка графов. Нестрого граф можно определить как набор точек, называемых вершинами, часть из которых соединена отрезками, называемыми ребрами. (Более строгое определение графа будет дано в следующем разделе.)

Графы являются довольно интересным объектом для изучения как с теоретиче­ской, так и с практической точек зрения. С помощью графов можно смоделировать довольно большое количество процессов, происходящих в реальной жизни, на­пример функционирование транспортных и коммуникационных сетей, календар­ное планирование проекта и т.д. Одна из последних интересных задач — оценка “диаметра” Web, заключающаяся в определении максимального количества ссы­лок, которые нужно пройти от одной Web-страницы до другой по оптимальному маршруту[[10]](#footnote-10).

К числу основных алгоритмов теории графов относят следующие:

* алгоритмы обхода графа (этот класс алгоритмов позволяет ответить на такие вопросы, как каким образом можно объехать все узлы железно­дорожной сети);
* алгоритмы определения кратчайшего пути (этот класс алгоритмов поз­воляет ответить на вопросы наподобие следующего: каков кратчайший путь между двумя городами);
* алгоритмы топологической сортировки для ориентированных графов ребрами (этот класс алгоритмов позволяет выяснить, не является ли множество читаемых студентам курсов внутренне противоречивым).

К счастью, все перечисленные алгоритмы можно рассматривать как пример общих методов проектирования. Поэтому вы сможете найти их описание в соот­ветствующих главах книги.

Некоторые из задач теории графов с вычислительной точки зрения очень труд­ны. Это означает, что только небольшое количество экземпляров подобных задач можно решить за приемлемое время с помощью самого быстродействующего ком­пьютера, который только можно себе представить. К наиболее известным задачам теории графов этого типа вероятнее всего относятся задачи коммивояжера и рас­краски графа. Напомним, что задача коммивояжера заключается в нахождении кратчайшего пути между п городами, каждый из которых он должен посетить только один раз. Задача раскраски графа (graph-coloring problem) заключается в том, что нужно с помощью минимального количества красок раскрасить вер­шины графа так, чтобы не было двух смежных вершин одинакового цвета. Эта задача появляется в процессе решения важных практических задач, таких как, например, календарное планирование. Если представить работы в виде вершин графа и соединить их между собой ребрами тогда и только тогда, когда соответ­ствующие им работы не могут выполняться в одно и то же время, решение задачи раскраски такого графа даст оптимальный график выполнения работ.

Комбинаторные задачи

С точки зрения обобщения задачи коммивояжера и раскраски графа являются примерами комбинаторных задач. Суть этих задач в конечном итоге сводится к нахождению такого комбинаторного объекта, как перестановка, сочетание или подмножество, который бы удовлетворял определенным ограничениям и обла­дал заданными свойствами (например, позволял максимизировать прибыль или минимизировать затраты.)

Вообще говоря, комбинаторные задачи относятся к классу самых сложных- как с теоретической, так и с практической точек зрения. Их сложность обуслов­лена следующим. Во-первых, количество комбинаторных объектов обычно очень быстро растет при увеличении масштаба задачи и достигает невообразимых зна­чений уже при весьма скромных ее масштабах. Во-вторых, пока не существует алгоритмов для поиска точного решения подобных задач за приемлемое время. Более того, большинство специалистов в области информатики считают, что та­ких алгоритмов попросту не существует. Это предположение не было ни доказано, ни опровергнуто. Оно до сих пор остается наиболее важным из нерешенных во­просов теоретической информатики. Более подробно этот вопрос мы обсудим в разделе 10.3.

Некоторые комбинаторные задачи можно решить с помощью эффективных алгоритмов, но это скорее счастливое исключение, чем правило.

Геометрические задачи

Геометрические алгоритмы связаны с такими геометрическими объектами, как точки, линии, многоугольники. Древние греки проявляли большой интерес к разработке процедур (естественно, они их еще не называли алгоритмами) реше­ния различных геометрических задач, среди которых можно назвать построение простых геометрических форм (треугольников, окружностей и т.п.) с помощью неградуированной линейки и циркуля. С тех пор прошло более 2000 лет, и в век компьютеров интерес к геометрическим алгоритмам вспыхнул с новой силой. Для решения подобных задач линейки и циркули уже не нужны — только биты, байты и накопленный годами человеческий опыт. Естественно, сейчас человече­ство проявляет совершенно другой интерес к геометрическим алгоритмам. Они нужны для решения современных задач компьютерной графики, робототехники и томографии.

В этой книге мы рассмотрим алгоритмы решения только двух классических задач вычислительной геометрии: поиска пары ближайших точек и определения выпуклой оболочки. Название задачи поиска пары ближайших точек говорит само за себя: среди п точек на плоскости необходимо выбрать пару точек, располо­женных наиболее близко друг к другу. При решении задачи построении выпуклой оболочки необходимо найти наименьший выпуклый многоугольник, который бы охватывал все точки некоторого множества на плоскости. Чтобы больше узнать об этих и других геометрических задачах, обратитесь к специализированным моно­графиям (например, [90]) или к соответствующим главам учебников, построенных по принципу описания конкретных типов задач (например, [102]).

Численные задачи

Численные задачи относятся к еще одной довольно обширной области прак­тического использования алгоритмов. Они имеют дело с математическими объ­ектами, которые по своей сути являются непрерывными. Вот примеры типичных численных задач: решение уравнений и систем уравнений, вычисление определен­ных интегралов и значений функций и т.д. Подавляющее большинство таких задач может быть решено только приблизительно. Еще одна принципиальная трудность заключается в том, что при решении подобных задач обычно нужно выполнять операции с вещественными числами, которые в компьютере могут быть представ­лены только с определенной погрешностью. Более того, выполнение большого количества арифметических операций над представленными приближенно числа­ми может привести к накоплению ошибок округлен™, что в свою очередь может кардинально повлиять на точность получаемых результатов.

За прошедшие годы было придумано большое количество довольно сложных алгоритмов решения численных задач, причем они продолжают играть ключевую роль при выполнении научных и инженерных расчетов. Однако в течение послед­них 25 лет интересы компьютерной индустрии сместились в сторону создания прикладных программ для деловой сферы. Для создания приложений этой кате­гории потребовалось разработать базовые алгоритмы хранения и выборки данных, передачи их по сетям и отображения в удобном для пользователя виде. В резуль­тате таких революционных изменений численный анализ утратил былые позиции как в области промышленного, так и научного использования программ. Тем не менее для любого человека, изучающего компьютерную литературу, важно иметь хотя бы элементарные понятия в области численных алгоритмов. Несколько таких классических алгоритмов мы опишем в разделах 6.2, 10.4 и 11.4.

Упражнения 1.3

1. Проанализируйте приведенный ниже алгоритм сортировки массива ме­тодом подсчета. Вначале для каждого элемента массива подсчитывает­ся количество элементов, меньших, чем он, и на основе этой информа­ции текущий элемент помещается в соответствующее место отсорти­рованного массива.

Алгоритм ComparisonCountingSort (А [0..П — 1])

// Сортировка массива методом подсчета сравнений // Входные данные: массив чисел А[0..п — 1], который нужно

// отсортировать

// Выходные данные: массив чисел £[0..п — 1], состоящий из // элементов массива А, отсортированных

// в неубывающем порядке

for г <— 0 to п — 1 do Count[i] **<— О** for i<— 0 to n — 2 do

for j \*— i+ 1 to n — 1 do

if A[i\ *< A[j]*

Count[j] <— Count[j] + 1 else

Count[i\ <— Count[i\ + 1 for i <— 0 to 7i — 1 do S[Count[г]] A[i\

return S

а) Попробуйте с помощью этого алгоритма отсортировать числа: 60, 35,81,98,14, 47.

б) Является ли этот алгоритм устойчивым?

в) Относится ли он к обменным алгоритмам?

1. Назовите известные вам алгоритмы поиска. Кратко опишите слова­ми каждый алгоритм. (Если вам еще не знаком ни один подобный алгоритм, воспользуйтесь удобной возможностью и разработайте его самостоятельно.)
2. Придумайте простой алгоритм поиска строк.

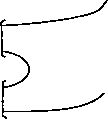
image9

1. Мосты Кенигсберга. Считается, что решение этой головоломки дало толчок развитию теории графов. Задача была решена выдающимся рос­сийским математиком швейцарского происхождения Леонардом Эйле­ром (Leonard Euler) (1707-1783). В головоломке предлагается ответить, можно ли одну прогулку обойти все семь мостов Кенигсберга и вер­нуться в отправную точку? На рис. 1.3 изображен схематический план реки, посередине которой расположены два острова, к которым ведут семь мостов.

а) Сформулируйте эту задачу в терминах теории графов.

б) Имеет ли задача решение? Если вы уверены в том, что имеет, на­чертите схему обхода мостов. Если нет, объясните, почему, и оцени-

flQ



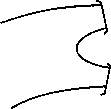
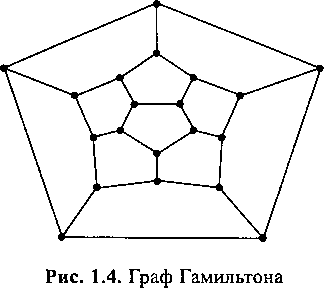


Рис. 1.3. Схематический план мостов Ке­нигсберга

те, какое минимальное количество новых мостов нужно построить, чтобы задача имела решение.

1. Игра в икосаэдр. Через сто лет после исследований Эйлера (см. за­дачу 1.3.4), известный ирландский математик сэр Вильям Гамильтон (Sir William Hamilton) (1805-1865) придумал еще одну головоломку — Icosian Game. На круглой деревянной доске вырезался граф, представ­ленный на рис. 1.4.



Начертите гамильтонов цикл (Hamiltonian circuit) этого графа, т.е. путь однократного обхода всех вершин графа с возвратом в исходную точку.

1. Разработайте алгоритм поиска наилучшего маршрута для пассажира метрополитена, который бы позволял ему перемещаться от одной стан­ции до другой по развитой системе подземных коммуникаций, наподо­бие вашингтонской или лондонской.

а) В условии этой задачи имеется небольшая неопределенность, что в общем-то типично для всех задач, находящих практическое при­менение. В частности, неясно, какой должен быть критерий оценки “наилучшего” маршрута? Можете ли вы его определить?

б) Можете ли вы смоделировать решение этой задачи с помощью графа?

1. а) Сформулируйте задачу коммивояжера в терминах комбинаторики,

б) Сформулируйте задачу раскраски графа в терминах комбинаторики.

1. Проанализируйте карту, изображенную на рис. 1.5.

|  |  |  |  |
| --- | --- | --- | --- |
|  | b | |  |
| а |  |  |  |
|  |  | |
|  |  | 6 | |
|  | с |  | |
| е |  |  | |
|  |  | f | |

Рис. 1.5. Учебная кон­турная карта

а) Поясните, как можно использовать решение задачи раскраски гра­фа для окрашивания этой карты так,, чтобы цвета двух соседних областей не совпадали.

б) Воспользуйтесь решением пункта а упражнения и раскрасьте карту минимально возможным количеством цветов

1. Разработайте алгоритм решения следующей задачи. Предположим, на плоскости находится п точек, заданных их координатами (х,у). Опре­делите, лежат ли все точки на одной и той же окружности?
2. Напишите программу, которая бы считывала из входного потока дан­ных координаты (х, у) конечных точек двух отрезков P\Q\ и P2Q2 и определяла, пересекаются ли эти отрезки (т.е. программа должна най­ти координаты общей точки двух отрезков, если таковая существует).
3. Базовые структуры данных

Поскольку большинство рассматриваемых в этой книге алгоритмов выполня­ют обработку данных, то на процесс их проектирования и анализа существенно влияют конкретные способы организации данных. Понятие структуры данных (data structure) можно определить как особую систему организации взаимосвязан­ных элементов данных. Структура самих элементов данных зависит от задачи, в которой они используются. Они могут быть как очень простыми (например, представлять собой целые числа или строки символов), так и сложными структу­рами данных (например, для представления матриц часто используют одномерный массив, каждый элемент которого, в свою очередь, также является одномерным массивом). Тем не менее существует несколько чрезвычайно важных при проекта- ровании алгоритмов для компьютеров структур данных. Поскольку вы наверняка знакомы с большинством из них (если не со всеми), ниже представлен лишь их краткий обзор.

Линейные структуры данных

Среди простых структур данных можно выделить две самых важных — одно­мерный массив и связанный список. Одномерный массив (array) — это последо­вательность из п однотипных элементов, расположенных подряд в оперативной памяти компьютера, доступ к которым выполняется по значению так называемого индекса массива (см. рис. 1.6).

|  |  |  |  |
| --- | --- | --- | --- |
| Элемент [0] | Элемент [1] | . . . | Элемент [п -1] |

Рис. 1.6. Одномерный массив из п элементов

В большинстве случаев индекс массива, состоящего из п элементов, является целым числом и принимает значения от 0 до п — 1 (как показано на рис. 1.6) или от 1 до п. В некоторых языках программирования допускается, чтобы индекс мас­сива принимал любые целочисленные значения (в том числе и отрицательные). Главное, чтобы он не выходил за пределы заранее установленного диапазона, определяющею нижнюю и верхнюю границы изменения индекса. Иногда в язы­ках программирования для доступа к элементам массива могут использоваться нечисловые индексы. Подобные массивы называются ассоциативными. Напри­мер, можно создать ассоциативный массив из 12 элементов, соответствующий месяцам года, доступ к котором осуществляется по названиям месяцев.

Время обращения к любому элементу массива одинаково и не зависит от его положения в массиве. Эта особенность выгодно отличает массивы от связанных списков, о которых речь пойдет ниже. Однако не стоит забывать, что каждый элемент массива занимает одинаковое количество ячеек памяти компьютера.

На основе одномерных массивов создаются различные структуры данных. Среди них выделяются строки (string), т.е. последовательности символов зара­нее определенного алфавита, заканчивающиеся особым символом. Этот символ служит признаком конца строки. Строки, состоящие только из нулей или единиц, называются двоичными {бинарными) (binary strings) или битовыми (bit strings). Строки являются основным элементом, используемым при обработке текстовых данных, определении синтаксиса языка программирования, компилировании на­писанных на нем программ, а также при изучении абстрактных вычислительных моделей. Выполняемые над массивом операции в первую очередь зависят от его типа. Так, операции обычно выполняемые со строками, отличаются от операций, выполняемых над массивом чисел. В этот перечень входит:

* определение длины строки;
* сравнение двух строк, позволяющее определить, какая из них распола­гается раньше согласно так называемому лексикографическому поряд­ку, т.е. как в словаре;
* объединение (конкатенации) двух строк, позволяющее сформировать из них одну строку, поместив вторую строку в конец первой.

Связанный список — это последовательность (которая может быть пустой) нескольких элементов данных, называемых узлами (nodes). В каждом узле хра­нится информация двух видов: собственно данные узла и одна или несколько ссылок на другие узлы связанного списка, называемых указателями (pointers). Если текущий узел связанного списка является последним, в него помещается т.н. нулевой указатель со специальным значением null; это говорит о том, что указатель ни на что не указывает. В однонаправленном связанном списке (singly linked list) каждый узел содержит только один указатель, в который помещает­ся ссылка на следующий элемент списка, либо “ноль”, если текущий элемент является последним (рис. 1.7).

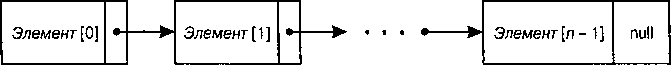


Рис. 1.7. Однонаправленный связанный список, состоящий из п элементов

Для обращения к заданному элементу связанного списка необходимо выбрать первый узел списка, а затем перемещаться по цепочке элементов до достижения нужного узла. Поэтому, в отличие от одномерных массивов, время, затрачива­емое на доступ к произвольному элементу однонаправленного списка зависит от его положения в списке. Это является существенным недостатком связанно­го списка. Однако у него есть и преимущество. Оно заключается в том, что для элементов связанного списка не требуется предварительное резервирование опе­ративной памяти компьютера. Кроме того, вставка и удаление элементов списка не представляют особого труда и выполняются достаточно быстро изменением значений нескольких указателей.

Гибкость структуры связанного списка позволяет использовать его различны­ми способами во множестве приложений. Например, часто бывает очень удобно, чтобы в первом элементе связанного списка находился специальный узел, называ­емый заголовком (header). В него обычно помещают информацию о самом списке, такую как текущее количество элементов в списке и указатель на последний эле­мент списка.

Существует модификация структуры однонаправленного связанного списка, которая называется двунаправленным связанным списком (doubly linked list).

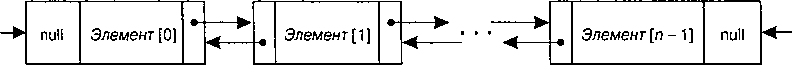


Рис. 1.8. Двунаправленный связанный список, состоящий из п элементов

Отличие между ними в том, что каждый узел двунаправленного списка (кроме первого и последнего) содержит указатели на предыдущий и последующий узлы (рис. 1.8).

Массив и связанный список чаще всего используются для представления еще одной абстрактной структуры данных, называемой линейным списком или просто списком. Список (list) представляет собой конечную последовательность элемен­тов данных, т.е. набор элементов данных, организованных в заданном линейном порядке. К этой структуре данных могут применяться такие базовые операции, как поиск, добавление и удаление элемента.

Среди всего множества списков можно выделить два типа: стеки и очереди. Стеком (stack) называют такой список, в котором можно удалить только по­следний элемент, а новый элемент добавить только в его конец. Последний часто называют вершиной (top) стека, поскольку стеки обычно изображают на рисунках в виде вертикального прямоугольника (по аналогии со стопкой тарелок, которую он напоминает). Кратко сформулировать принцип работы стека можно так: “по­следним вошел, первым вышел” (“last-in-first-out”, или LIFO), поскольку первым из стека всегда удаляется элемент, добавленный в него последним. Этим стек напоминает стопку тарелок, поскольку мы можем взять из нее только верхнюю тарелку, а новую тарелку — положить на верх стопки. Стеки широко применяют­ся на практике, в частности, без них не обойтись при реализации рекурсивных алгоритмов.

Под очередью (queue) понимается такой список, элементы которого удаляют­ся с одной стороны структуры (она называется головой (front)), а добавляются с другой (она называется хвостом (rear)). Первая операция называется удалени­ем элемента из очереди (dequeue), а вторая — постановкой в очередь (enqueue). Таким образом, принцип работы очереди можно сформулировать так: “первым вошел, первым вышел” (“first-in-first-out”, или FIFO). Здесь прослеживается пол­ная аналогия с очередью в магазине, которая образуется, если продавец медленно отпускает товар или наплыв покупателей слишком велик. Очереди также находят широкое практическое применение, в частности, в некоторых алгоритмах решения задач теории графов.

Во многих важных приложениях требуется выбрать среди динамически изме­няемого множества элемент с наивысшим приоритетом. Часто для решения по­добной задачи применяют структуру данных, которая называется очередью с при­оритетами, или приоритетной очередью (priority queue). Эта очередь представ­ляет собой совокупность элементов данных, относящихся к вполне упорядочен­ному универсуму[[11]](#footnote-11) (чаще всего целых или вещественных чисел). К основным операциям над очередью с приоритетами относятся: поиск наибольшего элемен­та, извлечение наибольшего элемента и добавление нового элемента. Само собой разумеется, что последние две операции должны быть реализованы так, чтобы в результате их выполнения получалась новая очередь с приоритетами (либо пе­реупорядочивалась старая). Проще всего реализовать рассматриваемую структуру данных на основе массива или упорядоченного массива, однако ни одно из этих решений не является оптимальным в плане достижения максимально возможной эффективности. В более удачных реализациях используется оригинальная струк­тура данных, называемая пирамидой (heap)[[12]](#footnote-12). Эту структуру данных и основные алгоритмы сортировки на ее основе мы обсудим в разделе 6.4.

Графы

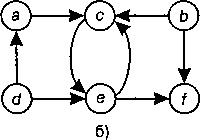
Как уже упоминалось, граф нестрого можно определить как совокупность точек на плоскости, называемых вершинами, или узлами, часть из которых со­единена отрезками, называемыми ребрами, или дугами. Строго граф G = (V, Е) определяется парой множеств: конечного множества элементов V, называемых вершинами, и множества Е, содержащего пары вершин, называемые ребрами. Если пары вершин неупорядочены, т.е. пара вершин (и, г?) означает то же самое, что и пара (v,u), то такой граф называют неориентированным. Если же пары упо­рядочены, то говорят, что ребро (и, v) ориентировано из вершины и к вершине v, при этом сам граф G называют ориентированным (directed). Ориентированные графы называют также диграфами (digraph).

Для удобства вершины графа обозначают латинскими буквами, целыми числа­ми или даже символьными строками, если этого требуют условия задачи (рис. 1.9). Граф, показанный на рис. 1.9а, имеет 6 вершин и семь ребер:

V = {а, Ь, с, d,e,f}, Е = {(а, с), (а, d), (£>, с), (£>, /), (с, е), (d, е), (е, /)} .

Ориентированный граф, изображенный на рис. 1.96, имеет шесть вершин и во­семь ориентированных ребер (дуг):

V {а, 6, с, d, е, f ]•, Е {(&? с), (6, с), (6, f), (с, е), (d, а), (d, е), (е, с), (е, /)} ■



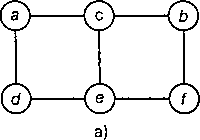


Рис. 1.9. а) Неориентированный граф; б) ориентирован­ный граф

В приведенном выше определении графа не исключены петли (loops), или ребра, начинающиеся и заканчивающиеся в одной вершине. В книге мы будем считать, что у графа не может быть петель, если явно не указано иное. Поскольку по определению не разрешается, чтобы между двумя вершинами неориентиро­ванного графа существовало несколько ребер, то для такого графа, имеющего \V\ вершин и не содержащего петель, можно оценить количество возможных ребер l^l с помощью следующего неравенства:

О ^ |Я|< \V\ (\V\ - 1)/2.

Если каждая из \V\ вершин графа соединяется ребрами с остальными \V\ — 1 вершинами, то количество ребер в нем будет максимальным. Поскольку граф неориентированный, произведение \V\ (\V\ — 1) необходимо разделить на 2, по­скольку между двумя вершинами (и, г?) имеется 2 ребра: от и к v и от v к и.

Граф, в котором каждая пара вершин соединяется ребром, называется пол­ным (complete). Стандартнее обозначение полного графа, состоящего из \V\ вер­шин, — К\у\. Граф, в котором отсутствует относительно небольшое количество ребер, называется плотным (dense), и, напротив, граф, в котором присутствует относительно небольшое количество ребер[[13]](#footnote-13), называется разреженным (sparse). От того, с каким графом мы имеем дело (плотным или разреженным), зависит способ его представления и, следовательно, время выполнения разрабатываемого или используемого алгоритма.

Представление графов

Графы, используемые в компьютерных алгоритмах, могут быть представлены двумя принципиально разными способами: в виде матрицы смежности и в виде связанных списков смежных вершин. Матрица смежности (adjacency matrix) графа, состоящего из п вершин, представляет собой булеву матрицу размером п х 71, в которой каждая строка и каждый столбец соответствуют одной из вершин графа. Элемент этой матрицы, находящийся на пересечении г-ой строки и j-гостолбца, равен 1 в случае, если г-я и j-я вершины графа соединяются ребром, и 0 в противном случае[[14]](#footnote-14). Например, на рис. 1.10а показана матрица смежности для графа, изображенного на рис. 1.9а. Обратите внимание, что матрица смеж­ности для неориентированного графа всегда симметрична (понятно, почему?), т.е. А [г, j] = A [j, г] для всех 0 ^ г, j ^ п — 1.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | a | b | с | d | е | f | |  |  |  |  |  |
| а | "0 | 0 | 1 | 1 | 0 | o' |  | a | -> | с | -> | d |
| Ь | 0 | 0 | 1 | 0 | 0 | 1 |  | b\_ | -> | с | —► | f |
| с | 1 | 1 | 0 | 0 | 1 | 0 |  | с | —► | a | —► | b -> e |
| д | 1 | 0 | 0 | 0 | 1 | 0 |  | d\_ | -> | a | -> | e |
| е | 0 | 0 | 1 | 1 | 0 | 1 |  | e | -> | с | -> | d -> f |
| f | О | 1 | 0 | 0 | 1 | 0\_ |  | \_f\_ | —► | b | -> | e |

а) б)

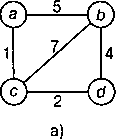
Рис. 1.10. Матрица смежности а) и связанные спис­ки смежных вершин б) для графа, изображенного на рис. 1.9а

Связанные списки смежных вершин (adjacency linked lists) графа представ­ляют собой совокупность связанных списков (по одному для каждой вершины), в которых содержится информация обо всех смежных вершинах текущей верши­ны (т.е. в каждом связанном списке содержится список всех вершин, соединенных ребром с текущей вершиной). Как правило, в начале каждого подобного списка располагается заголовок, содержащий информацию, идентифицирующую верши­ну, для которой этот список составлен. Например, на рис. 1.106 с помощью связан­ных списков смежных вершин представлен граф, изображенный на рис. 1.9а. Если взглянуть на все с другой стороны, то в связанных списках присутствуют вершины графа, которым в матрице смежности соответствуют единичные элементы.

Связанные списки смежных вершин имеет смысл использовать для представ­ления разреженных графов, поскольку при этом требуется меньше оперативной памяти по сравнению с матрицей смежности (которая, кстати, в этом случае будет состоять практически из одних нулей). Естественно, что дополнительную память, которую занимают указатели связанного списка, мы в расчет не принимаем. Од­нако ситуация в корне меняется в случае плотных графов. Вообще говоря, выбор способа представления графа зависит от типа задачи, алгоритма, используемого для ее решения и, возможно, от типа исходного графа (в зависимости от того, разреженный он или плотный).

Взвешенные графы

Под взвешенным графом (weighted graph) подразумевается граф, ребрам ко­торого поставлены в соответствие числа, как показано на рис. 1.11а. Эти числа называются весами (weights) или ценами, или стоимостями (costs). Интерес к подобного типа графам был вызван большим количеством прикладных задач, таких как задача поиска кратчайшего маршрута между двумя пунктами или уже упоминавшаяся выше задача коммивояжера. Интересно, что определение кратчай­шего пути может использоваться как при транспортировке грузов, так и в сетях передачи данных.



Оба рассмотренных выше способа представления графов легко приспособить и для случая взвешенных графов. Если взвешенный граф представляется с помо­щью матрицы смежности А, то ее элемент A [i,j] должен равняться весу ребра, соединяющего г-ю и j-ю вершины. Если же эти вершины не соединены ребром, то вместо веса элемент А [г, j] должен содержать какой-нибудь специальный знак, на­пример знак бесконечности оо. Такая матрица называется матрицей весов (weight matrix) или матрицей стоимости (cost matrix). Подобный подход проиллюстри­рован на рис. 1.116. (В некоторых случаях удобнее, чтобы на главной диагонали матрицы смежности располагались нули, а не знаки бесконечности.) Что касается представления взвешенного графа с помощью связанных списков смежных вер­шин, то каждый узел списка нужно немного расширить, включив в него не только имя смежной вершины, но и весовой коэффициент соответствующего ребра, как показано на рис. 1.11 в.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | а | Ь | с | d | |  |  |  |  |  |  |
| а | 'со | 5 | 1 | оо |  | а | -> | Ь, 5 | —► | с, 1 |  |
| b | 5 | оо | 7 | 4 |  | Ь\_ | -> | а, 5 | —► | с, 7 -> | d, 4 |
| с | 1 | 7 | оо | 2 |  | с | -> | а, 1 | -> | Ь,7 | d, 2 |
| 6 | оо | 4 | 2 | 00 |  | 6\_ | —► | Ь, 4 | -> | с, 2 |  |

б)

В)

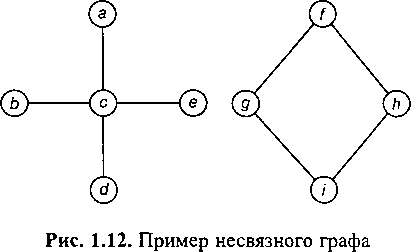
Рис. 1.11. а) Взвешенный граф и его представление б) с помощью матрицы весов и в) связанных списков смежных вершин

Пути и циклы

Среди множества интересных свойств графа можно выделить два особенно важных для решения огромного количества прикладных задач: связность (connec­tivity) и ацикличность (acyclicity). Оба этих свойства основаны на понятии пути. Путь, или маршрут, (path) от вершины и к вершине v можно определить как последовательность смежных (т.е. соединенных ребром) вершин, которая начина­ется в вершине и и заканчивается в вершине v. Если все ребра графа различны, то такой путь называют простым (simple). Под длиной (length) пути понимает­ся общее количество вершин в последовательности, определяющей путь, минус единица. Другими словами, длина пути равна количеству ребер, через которые он проходит. Например, на рис. 1.9а от вершины а до вершины / можно проло­жить простой путь длиной 3: а, с, Ь, /. В то же время между этими вершинами существует путь (не простой) длиной 5: а, с, е, с, 6, /.

В случае ориентированных графов нас обычно будут интересовать ориенти­рованные пути. Под ориентированным маршрутом (directed path) понимается последовательность вершин, в которой каждая следующая друг за другом пара вершин (и, г?) соединена дугой, начинающейся в вершине и и заканчивающейся в вершине v. Например, на рис. 1.96 существует следующий ориентированный путь из вершины а к вершине /: а, с, е, /.

Граф называется связным (connected), если для любой пары его вершин и и v существует путь от и к v. Объяснить это свойство графа, что называется, “на паль­цах” можно так: если создать модель этого графа, связав между собой веревками (которые будут выполнять роль ребер) несколько мячей (они будут выполнять роль вершин), то все предметы можно будет удержать в одной руке. Если граф не является связным, то модель будет состоять из нескольких отдельных участков, которые называются связными компонентами. Строго связный компонент (con­nected component) графа можно определить как максимальный связный подграф[[15]](#footnote-15) данного графа, который нельзя расширить за счет включения дополнительной вершины, смежной с одной из ее вершин. Например, графы, изображенные на рис. 1.9а и 1.11 а, являются связными, тогда как граф, изображенный на рис. 1.12, не является связным, поскольку нельзя проложить путь, скажем, от вершины а к вершине /. Граф, изображенный на рис. 1.12, имеет два связанных компонента с вершинами {а, 6, с, d, е} и {/, р, /г, г} соответственно.



Графы с несколькими компонентами связности успешно применяются при решении реальных прикладных задач. В качестве примера можно привести пред­ставление в виде графа системы магистральных шоссейных дорог (highway sys­tem), проложенных между штатами в США (можете объяснить, почему?).

Во многих прикладных задачах важно знать, содержит ли рассматриваемый граф циклы. Под циклом (cycle) понимается простой путь положительной дли­ны, который начинается и заканчивается в одной и той же вершине. Например, в графе, изображенном на рис. 1.12, циклом является путь: /, h, i, д, /. Граф, не содержащий циклов, называют ациклическим (acyclic). Ациклические графы будут рассмотрены в следующем подразделе.

Деревья

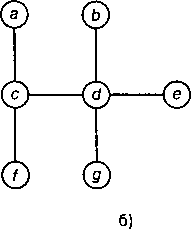
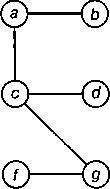


Рис. 1.13. Пример а) дерева и б) леса



а)

Деревом (tree) (точнее, свободным деревом (free tree)) называется связный ациклический граф (рис. 1.13а). Граф, который не содержит циклов, но не обяза­тельно является связным, называется лесом (forest). Каждая компонента связности леса является деревом (рис. 1.136).

©

0

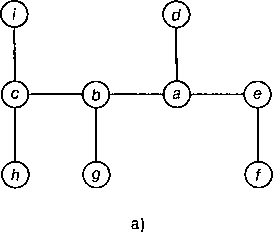
Деревья имеют несколько важных свойств, не присущих другим графам. В частности, число ребер в дереве всегда на единицу меньше, чем число его вершин:

|E| = |F|-1.

На примере графа, показанного на рис. 1.12, можно сделать вывод, что это свойство является необходимым, но не достаточным, чтобы граф был деревом. Тем не менее для связных графов это свойство является достаточным, поэтому с его помощью удобно определять, содержит ли граф цикл.

Корневые деревья

Еще одним важным свойством деревьев является следующее: для любых двух вершин дерева существует только один простой путь от одной вершины к другой. Это свойство позволяет назначить произвольный узел в свободном дереве корнем (root) и преобразовать свободное дерево в так называемое корневое дерево (rootedtree). Последнее обычно изображается так, чтобы его корень был вверху, т.е. рас­полагался на так называемом нулевом уровне дерева. Ниже корня располагаются смежные с ним вершины. Они составляют первый уровень дерева. Вершины, соединенные с корнем двумя ребрами, составляют следующий (т.е. второй) уро­вень дерева, и т.д. На рис. 1.14 показано, как преобразовать свободное дерево в корневое.



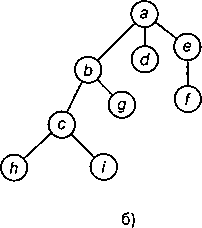


Рис. 1.14. Преобразование а) свободного дерева в б) корневое

Корневые деревья играют в информатике гораздо более важную роль, чем сво­бодные деревья. Поэтому часто для краткости их называют просто “деревьями”. Их основное назначение — описание иерархических структур, таких как струк­тура каталогов файловой системы или организационная структура предприятия. Однако существуют и другие, менее очевидные области применения деревьев, на­пример для организации словарей (об этом пойдет речь в следующем подразделе), для эффективного хранения очень больших массивов данных (см. раздел 7.4), для кодирования данных (см. раздел 9.4). Как вы увидите в главе 2, деревья также находят применение при анализе рекурсивных алгоритмов. И, чтобы закончить этот далеко не полный перечень возможных применений деревьев, следует упо­мянуть так называемые деревья пространств состояний (state-space trees). С их помощью можно объяснить два важных метода проектирования алгоритмов: по­иск с возвратом (backtracking) и метод ветвей и границ (branch-and-bound). О них речь пойдет в разделах 11.1 и 11.2.

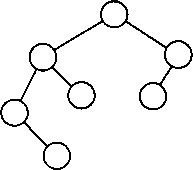
Для любой вершины v в дереве Т верно следующее: все вершины, принадле­жащие простому маршруту от корня дерева до этой вершины, называются предка- ми (ancestors) v. Сама вершина v обычно считается собственным предком. Множе­ство предков, в которое не включен собственный предок, называется множеством истинных предков (proper ancestors), или просто истинными предками. Если (и, v) является последним ребром, принадлежащим простому маршруту, ведуще­му из корня дерева до вершины г? (и иф v), то говорят, что и является родителем (parent) v, a v называют потомком, или дочерней (child) вершиной и. Верши­ны, имеющие общего родителя, называются родственными, или сестринскими(sibling). Вершина, у которой нет потомков, называется листом (leaf). Вершина, у которой есть как минимум один потомок, называется родительской (parental). Совокупность всех вершин, для которых вершина v является предком, называют потомками (descendants) v. Вершина v вместе со всем своими потомками назы­вается поддеревом (subtree) дерева Т с корнем (поддерева) в вершине v. Таким образом, для дерева, изображенного на рис. 1.146, корнем является вершина а; вершины d, g, /, /гиг являются листьями; вершины а, 6, е и с являются родитель­скими. Родительской вершиной для b является вершина а; потомками вершины b являются вершины сир; родственными для Ъ являются вершины d и е; вершинами поддерева с корнем в Ъ являются: {6, с, р, /г, г}.

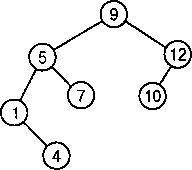
Под глубиной (depth) вершины v понимается длина простого пути от корня до этой вершины. Под высотой (height) дерева понимается длина наибольшего про­стого пути от его корня до одного из листьев. Например, в дереве, изображенном на рис. 1.146, глубина вершины с равна 2, а высота дерева равна 3. Таким образом, если сосчитать уровни дерева сверху вниз, начиная с корня (ему соответствует нулевой уровень), то глубина вершины будет равна уровню этой вершины в де­реве, а высота дерева будет соответствовать максимальному уровню, на котором могут находиться вершины в дереве. (Обратите внимание, что в некоторых кни­гах высота дерева определяется как количество уровней в дереве. При подобном определении высота дерева будет на единицу больше, чем высота, которая соот­ветствует длине наибольшего простого маршрута от корня до одного из листьев.)

Упорядоченные деревья

Упорядоченным (ordered) называется такое корневое дерево, в котором упоря­дочены все потомки каждой вершины. Принято считать, что на диаграмме деревья изображаются так, чтобы все потомки были упорядочены слева направо. Двоич­ное {бинарное) дерево (binary tree) можно определить как упорядоченное дерево, каждая вершина которого имеет не более двух потомков, причем каждый из потом­ков считается либо левым (left child) либо правым (right child) потомком своего родителя. Поддерево, корень которого находится в левом (правом) потомке вер­шины, называется левым {правым) поддеревом этой вершины. Пример бинарного дерева показан на рис. 1.15а.

На рис. 1.156 вершинам бинарного дерева, показанного на рис. 1.15а, назначе­ны числа. Обратите внимание, что число, которое назначено каждой родительской вершине, больше, чем число, назначенное ее левому потомку, и меньше, чем чис­ло, назначенное ее правому потомку. Деревья подобного типа называют бинарным деревом поиска (binary search trees). Бинарные деревья и бинарные деревья поиска находят широкое применение в информатике. С некоторыми из них вы столкне­тесь при чтении этой книги. Бинарные деревья поиска являются частным случаем более общего типа деревьев поиска, которые называются многоканальными де-





а) б)

Рис. 1.15. Пример а) бинарного дерева и б) бинарного дерева поиска

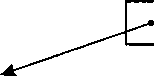
ревьями поиска (multiway search trees). Последние незаменимы при организации эффективного хранения на дисках файлов очень больших размеров.

Ниже в этой книге мы покажем, что эффективность большинства основных алгоритмов, в которых используются бинарные деревья поиска и их вариации, непосредственно зависит от высоты дерева. Поэтому приведенное ниже неравен­ство для высоты бинарного дерева h, содержащего п вершин, особенно важно для анализа подобных алгоритмов:

[log2nJ < h < п — 1.

При использовании в компьютерных программах бинарные деревья обычно реализуют в виде связанного списка, узлы которого соответствуют вершинам де­рева. В каждый узел помещается информация о соответствующей ему вершине дерева (ее имя, присвоенное значение и т.д.), а также два указателя на его левого и правого потомка. На рис. 1.16 показана одна из подобных реализаций бинарного дерева поиска, показанного на рис. 1.156.

Упорядоченное дерево в компьютере можно представить в виде родительской вершины с указателями на всех ее потомков. Однако такое представление не со­всем удобно в том случае, когда количество потомков в процессе работы програм­мы меняется в очень широких пределах. Чтобы избежать подобного неудобства, можно воспользоваться структурой данных, каждый узел которой будет иметь только два указателя (по аналогии с представлением бинарных деревьев). При этом левый указатель будет содержать ссылку на первого потомка текущей вер­шины, а правый — на следующую родственную ей вершину. Поэтому такое пред­ставление произвольного упорядоченного дерева называют “первый потомок — следующий родственник” (first child — next sibling). Таким образом, все родствен­ные вершины будут связаны в один список с помощью правого указателя текущей вершины. При этом ссылка на первый элемент списка будет содержаться в левом указателе ее родителя. На рис. 1.17а показано подобное представление дерева, показанного на рис. 1.146. Нетрудно заметить, что в результате подобного пред­ставления упорядоченное дерево оказалось преобразовано в бинарное. Поэтому



|  |  |  |
| --- | --- | --- |
|  | / 5 N | f 12 null |
| / | ч | / |
| null 1 | \ null 7 null | null 10 null |

null

null

Рис. 1.16. Типовая реализация бинарного дерева поиска, показанного на рис. 1.156

говорят, что данное бинарное дерево ассоциировано с упорядоченным деревом. Чтобы убедиться в этом, достаточно “развернуть” стрелки указателей по часовой стрелке примерно на 45°, как показано на рис 1.176.

a)

Рис. 1.17. а) Представление графа, показанного на рис. 1.146 в виде структуры “первый потомок — следующий родственник”; б) ассоциированное с ним бинарное дерево

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | b |  |
| 1 | r | | |
|  |  | с |  |
|  | r | | |
| null | | h |  |

null

null d

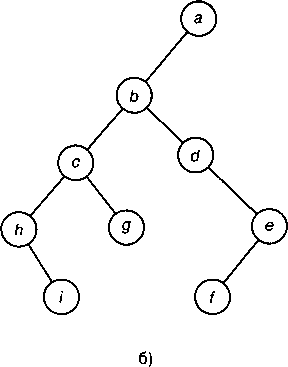
null

null / null

1

**Множества и словари**

Понятие множества играет основную роль в математике. Множество (set) можно охарактеризовать как неупорядоченную совокупность (которая может быть



e null

i

null

null

null

пуста) отдельных элементов, называемых элементами множества. Конкретное множество определяется либо в виде явного списка элементов (например, S = = {2,3,5,7}), либо в виде указания условия, которому должны удовлетворять все элементы множества и только они (например, S = {п, где п — простое число, меньшее 10}). Основными операциями над множествами являются: проверка на членство в данном множестве (т.е. принадлежит ли элемент множеству; другими словами, нужно найти заданный элемент среди элементов множества), поиск объ­единения двух множеств (т.е. формирование нового множества, элементы которо­го принадлежат либо первому, либо второму множеству, либо обоим множествам одновременно), а также поиск пересечения двух множеств (т.е. формирование нового множества, элементы которого принадлежат только обоим множествам одновременно).

При разработке компьютерных приложений множества можно реализовать двумя способами. В первом случае рассматриваются только множества, кото­рые являются подмножествами некоторого большого множества £/, называемого универсальным множеством, или универсумом (universal set). Если множество U состоит из п элементов, то любое его подмножество S можно представить в виде строки из п битов, называемой битовым вектором (bit vector), в ко­торой г-й элемент равен 1 тогда и только тогда, когда г-й элемент множества U включен в подмножество S. Поэтому, возвращаясь к нашему примеру, если U = {1,2,3,4,5, б, 7,8,9}, то подмножество S = {2,3,5,7} можно представить в виде битовой строки 011010100. Описанный способ представления множеств позволяет реализовать стандартный набор операций над множеством в виде очень быстрых процедур. Однако платой за скорость является большой объем оператив­ной памяти, необходимой для представления элементов множества[[16]](#footnote-16).

Чаще всего в компьютерных программах используется второй способ пред­ставления множества. Он заключается в представлении элементов множества в виде связанного списка. Само собой разумеется, речь идет о представлении только конечных множеств. К счастью, в отличие от математики, данный вид множеств наиболее широко используется в компьютерных приложениях. Тем не менее необходимо обратить внимание на два принципиальных различия между множествами и списками. Во-первых, в множестве не может быть одинаковых элементов, а в списке — может. Иногда это требование к уникальности элементов обходят путем введения так называемого мультимножества (multiset), или паке­та (bag), т.е. неупорядоченной совокупности элементов, которые необязательно должны быть различными. Во-вторых, поскольку множество — это неупорядо­ченная совокупность элементов, изменение порядка его элементов не изменяет само множество. В свою очередь, список определяется как упорядоченная со­вокупность элементов, что в корне противоположно понятию множества. Выше мы перечислили важные теоретические различия между множеством и списком, однако, к счастью, они не так важны для большинства приложений. Кроме того, стоит упомянуть, что в том случае, когда множество представляется в виде списка, то для ряда приложений имеет смысл поддерживать его в отсортированном виде.

В прикладных программах над множеством или мультимножеством чаще всего выполняются следующие операции: поиск заданного элемента, добавление нового элемента и удаление элемента из совокупности. Структуру данных, задействован­ную в реализации трех этих операций, называют словарем (dictionary). Заметим, что о взаимосвязи этой структуры данных с задачами поиска говорилось выше в разделе 1.3. Очевидно, что здесь речь идет о выполнении поиска в динами­чески изменяемом контексте. Поэтому эффективная реализация словаря должна быть результатом достижения компромисса между эффективным выполнением поиска и двух других операций над множеством. В действительности существует несколько способов реализации словаря. Они могут быть как очень простыми, с использованием обычных массивов сортированных (или несортированных) эле­ментов, так и довольно сложными. В последнем случае используются методы хеширования и сбалансированные деревья поиска, которые будут описаны в по­следующих главах этой книги.

При решении ряда прикладных задач из области вычислительной техники нужно динамически разбить множество, состоящее из п элементов, на ряд непе- ресекающихся подмножеств. После инициализации множества как набора из п одноэлементных подмножеств задача сводится к последовательности операций поиска и объединения. Эта задача называется задачей объединения множества (set union). Эффективные алгоритмы решения этой задачи будут описаны в разде­ле 9.2 при рассмотрении одного из важных ее применений.

Вы, наверное, уже заметили, что в приведенном выше обзоре основных струк­тур данных мы почти всегда упоминали об особенностях операций, которые обыч­но выполняются над рассматриваемыми структурами данных. Подобную тесную взаимосвязь между структурами данных и выполняемыми над ними операциями специалисты в области информатики заметили уже давно. Это навело их на мысль о существовании так называемых абстрактных типов данных (abstract data type, или/ЮГ), т.е. множества абстрактных объектов, представляющих элементы данных, и определенного на нем набора операций, которые могут быть выпол­нены над элементами этого множества. В качестве иллюстрации этого понятия перечитайте, например, приведенные выше определения приоритетной очереди и словаря. Несмотря на то что абстрактные типы данных можно реализовать с по­мощью старых процедурных языков программирования, таких как Pascal (в каче­стве примера см. [5]), все-таки намного удобнее делать это с помощью объектно- ориентированных языков программирования, таких как C++ или Java, в которых абстрактные типы данных поддерживаются с помощью классов (classes).

1. Поясните, как можно реализовать каждую из перечисленных ниже опе­раций над массивом так, чтобы время ее выполнения не зависело от размера массива п.

а) Удаление г-го элемента массива 1 < г < п).

б) Удаление г-ro элемента отсортированного массива (разумеется, об­разовавшийся после удаления массив также должен остаться отсор­тированным).

1. Предположим, нужно найти число в списке, состоящем из п элемен­тов. Насколько упростится решение, если список будет отсортирован? Рассмотрите отдельные решения для случаев, когда

а) список представлен в виде массива;

б) список представлен в виде связанного списка.

1. а) Каким будет содержимое стека после выполнения каждой коман­

ды приведенной ниже последовательности, считая, что в исходном состоянии стек пуст:

***push*** (a), ***push (b), pop, push*** (с), ***push*** (***d***), ***pop.***

б) Изобразите содержимое очереди после выполнения каждой коман­ды приведенной ниже последовательности, считая, что в исходном состоянии очередь пуста:

***enqueue*** (a), ***enqueue (b), dequeue, enqueue*** (с), ***enqueue*** (***d***),

***dequeue.***

1. a) Предположим, А является матрицей смежности неориентирован­

ного графа. Объясните, какое из свойств матрицы свидетельствует о том, что:

1. граф является полным;
2. граф содержит петли, т.е. какая-либо из вершин соединена реб­ром сама с собой;
3. граф содержит изолированную вершину, т.е. в нем встречается вершина, не соединенная ребрами с другими вершинами.

б) Дайте ответы на те же вопросы для представления в виде связанных списков смежности.

1. Приведите подробное описание алгоритма преобразования свободного дерева в корневое дерево, корень которого находится в заранее опре­деленной вершине свободного дерева.
2. Докажите приведенное ниже неравенство для высоты бинарного дерева h, содержащего п вершин:

|\_log2 п\ ^ h ^ п — 1.

1. Покажите, как можно реализовать абстрактный тип данных очереди с приоритетами в виде:

а) (несортированного) массива;

б) отсортированного массива;

в) бинарного дерева поиска.

1. Как бы вы реализовали словарь, содержащий достаточно малое ко­личество элементов п, при условии, что все его элементы различны (например, представляют собой названия 50 штатов США)? Опишите реализацию каждой операции, выполняемой словарем.
2. Для каждой из перечисленных ниже задач укажите наиболее подходя­щую с вашей точки зрения структуру данных.

а) Система автоматического ответа на телефонные звонки с учетом их приоритетности.

б) Система отправки покупателям счетов за сделанные заказы, которая бы учитывала порядок поступления заказов.

image27

в) Реализации калькулятора для вычисления простых арифметических выражений.

10. Разработайте алгоритм проверки того, являются ли два заданных слова анаграммами, т.е. того, что одно из слов может быть получено путем перестановки букв другого слова. (Например, анаграммами являются слова кот и ток.)

Резюме

* Алгоритм — это последовательность четко определенных инструк­ций, предназначенных для решения некоторой задачи за ограниченный промежуток времени. Входные данные, обрабатываемые алгоритмом, определяют экземпляры задачи, решаемой с помощью выбранного ал­горитма.
* Алгоритмы могут быть описаны на естественном языке либо на псев­докоде. Они также могут быть реализованы в виде компьютерных про­грамм.
* Существует несколько способов классификации алгоритмов, но среди них можно выделить две принципиальные альтернативы:
* группирование алгоритмов в соответствии с типом решаемой с их помощью задачи;
* группирование алгоритмов в соответствии с лежащими в их ос­нове методами проектирования.
* Важными типами задач являются: сортировка, поиск, обработка строк, задачи теории графов, комбинаторные задачи, геометрические задачи и численные задачи.
* Метод проектирования алгоритма — это универсальный подход, при­меняемый для алгоритмического решения широкого круга задач, отно­сящихся к различным областям вычислительной техники.
* Несмотря на то что проектирование алгоритма, безусловно, — творче­ский процесс, в нем тем не менее можно выделить последовательность взаимосвязанных действий, которые необходимо выполнить, как пока­зано на рис. 1.2.
* Хороший алгоритм получается, как правило, в результате кропотливой циклической работы, связанной с возможными переделками.
* Часто одну и ту же задачу можно решить с помощью нескольких алго­ритмов. Например, для вычисления наибольшего общего делителя двух целых чисел в этой главе было описано три алгоритма: алгоритм Ев­клида, алгоритм последовательного перебора целых чисел и алгоритм, который вы изучали в средней школе. Последний алгоритм был на­ми немного усовершенствован: для получения упорядоченного списка простых чисел мы применили алгоритм под названием решето Эра­тосфена.
* Алгоритмы работают с данными. Поэтому для эффективного реше­ния алгоритмической задачи необходимо правильно структурировать данные. Среди простых структур данных самыми важными являют­ся массив и связанный список. Они используются для представления более абстрактных структур данных, таких как список, стек, очередь, граф (представляется с помощью матрицы смежности или связанных списков смежных вершин), бинарное дерево и множество.
* Множество абстрактных объектов, представляющих элементы данных и набор операций, которые могут быть выполнены над элементами это­го множества, называются абстрактным типом данных (ADT). Важ­ными примерами абстрактных типов данных являются список, стек, очередь, очередь с приоритетами и словарь. В современных языках программирования реализация абстрактных типов данных поддержи­вается с помощью классов.

**Глава**

Основы анализа эффективности алгоритмов

Когда вы можете измерить и выразить в цифрах то, о чем говорите, вы что-то знаете об изучаемом предмете. Однако когда вы не можете выразить в цифрах то, о чем говорите, вы имеете весьма скудное пред­ставление об изучаемом предмете: возможно, это начало познания, но вряд ли вам удастся довести ваши идеи до состояния науки, о чем бы ни шла речь.

— Лорд Кельвин (Lord Kelvin) (1824-1907)

Не все, что можно подсчитать, принимают в расчет, и не все, что при­нимают в расчет, можно подсчитать.

— Альберт Эйнштейн (Albert Einstein) (1824-1907)

Э

та глава посвящена анализу алгоритмов. В американском энциклопедическом словаре American Heritage Dictionary слово “анализ” (“analysis”) определя­ется как “разделение материального или духовного целого на составные части с целью их последующего изучения”. Таким образом, все основные характери­стики алгоритмов, упомянутые в разделе 1.2, должны быть изучены на вполне законных основаниях. Однако формально термин “анализ алгоритмов” обычно ис­пользуется в более узком смысле и означает процесс исследования эффективности алгоритмов, которую можно оценить по двум параметрам: времени выполнения алгоритма и требуемому объему оперативной памяти. Важность эффективности очень легко объяснить. Во-первых, в отличие от таких характеристик алгоритма, как простота и универсальность, эффективность можно выразить количественно. Во-вторых, можно доказать, что эффективность является вопросом первостепен­ной важности с практической точки зрения. Однако сделать это почти всегда довольно сложно, учитывая быстродействие современных компьютеров и объ­ем их оперативной памяти. Тем не менее в этой главе речь пойдет именно об эффективности алгоритмов.

Начнем обсуждение с описания в разделе 2.1 общих основ процесса анализа эффективности алгоритмов. Наверное, этот раздел является самым важным в гла­ве. Более того, поскольку в нем рассмотрены фундаментальные понятия, можно сказать что этот раздел является самым важным во всей книге.

В разделе 2.2 мы познакомимся с тремя условными обозначениями: О (про­писная “О”), © (прописная греческая “тета”) и £1 (прописная греческая “омега”). Они были взяты из математики и являются языком, на котором происходит об­суждение эффективности алгоритмов.

В разделе 2.3 мы покажем, как полученные в разделе 2.1 базовые знания можно систематически применять к анализу эффективности нерекурсивных алгоритмов. Основным средством такого анализа является запись суммы, выражающая вре­мя выполнения алгоритма и последующее упрощение этой суммы с помощью стандартных правил суммирования.

В разделе 2.4 мы покажем, как полученные в разделе 2.1 базовые знания применять к анализу эффективности рекурсивных алгоритмов. Здесь основным средством анализа будет уже не сумма, а специальный тип уравнения, называемо­го рекуррентным. В этом разделе будет объяснено, как строить эти рекуррентные уравнения, а также приведен метод их решения.

Несмотря на то что в первых четырех разделах этой главы основы анализа эффективности алгоритмов и методы их применения проиллюстрированы с по­мощью разнообразных примеров, в разделе 2.5 будет рассмотрен еще один при­мер, посвященный числам Фибоначчи. Хотя эта знаменитая последовательность чисел была придумана более 800 лет назад, она по-прежнему широко применя­ется и в информатике, и в других прикладных науках. Благодаря обсуждению последовательности Фибоначчи нам удастся естественным образом познакомить читателя с важным классом рекуррентных соотношений, которые невозможно ре­шить методами, рассмотренными в разделе 2.4. Кроме того, мы также обсудим несколько алгоритмов вычисления последовательности чисел Фибоначчи и на их основе выскажем несколько общих замечаний по поводу эффективности алгорит­мов и методов ее анализа.

Методы, описанные в разделах 2.3. и 2.4, являются достаточно мощным сред­ством анализа эффективности многих алгоритмов, выполняемого с математиче­ской точностью, однако они понятны далеко не всем. В последних двух разделах этой главы будут описаны два подхода (эмпирический анализ и визуализация ал­горитма), которые дополняют чисто математические методики, изложенные в раз­делах 2.3 и 2.4. Хотя эти методики сравнительно новы и еще не так широко рас­пространены, как их математические аналоги, тем не менее им отводится особое место среди средств анализа эффективности алгоритмов.

1. Основы анализа

В этом разделе мы познакомимся с основными понятиями, использующимся для анализа эффективности алгоритмов. Начнем с рассмотрения двух видов эф­фективности: временной и пространственной. Временная эффективность {time efficiency) является индикатором скорости работы алгоритма. Пространственная эффективность {space efficiency) показывает, сколько дополнительной оператив­ной памяти нужно для работы алгоритма. На заре развития электронных вычис­лительных машин (ЭВМ) особое значение имели два ресурса: быстродействие центрального процессора и объем оперативной памяти. Однако более чем полу­вековой процесс непрерывного технического прогресса привел к тому, что быст­родействие и объем оперативной памяти вычислительных устройств увеличились во много раз. Теперь требовании к дополнительному объему оперативной памя­ти, необходимой для работы алгоритма, стали не так важны, как раньше. Однако здесь следует сделать оговорку, что, конечно же, существует определенная разница между быстрой основной памятью, относительно медленной вторичной памятью и кэш-памятью. Что касается временных характеристик алгоритма и быстродей­ствия современных компьютеров, то здесь, к сожалению, нельзя сказать, что эти вопросы совсем сняты с повестки дня. Более того, исходя из опыта научных исследований, можно сказать, что для большинства задач удается добиться суще­ственного выигрыша именно в скорости работы алгоритма, а не в сокращении требуемого объема памяти (зачастую первое происходит за счет второго). Поэто­му по сложившейся во многих учебниках по алгоритмам хорошей традиции, мы в основном сосредоточим внимание на временной эффективности. Тем не менее описанную в этом разделе аналитическую основу вы можете с успехом применить и для анализа пространственной эффективности алгоритмов.

Оценка размера входных данных

Начнем обсуждение с констатации очевидного факта, что время выполнения большинства алгоритмов напрямую зависит от размера вводимых данных (т.е. чем больше размер, тем дольше работает алгоритм). Например, довольно долго длится процесс сортировки больших массивов данных, перемножения больших матриц и т.п. Поэтому вполне логично описать эффективность алгоритма в виде функции от некоторого параметра п, связанного с размером входных данных[[17]](#footnote-17). В большин­стве случаев выбрать такой параметр не представляет большого труда. Например, для задач, связанных с сортировкой, поиском, нахождением наименьшего элемен­та в списке и многих других, связанных с обработкой списков, таким параметром будет размер списка. Для задачи вычисления значения многочлена степени п р (х) = апхп + • • • + ао таким параметром может быть степень многочлена или количество его коэффициентов, которое на единицу больше степени многочлена. Ниже мы увидим, что подобное несущественное отличие не влияет на результаты анализа.

Конечно, бывают случаи, когда выбор параметра, показывающего размер вход­ных данных, имеет особое значение. Один из примеров подобных случаев — пе­ремножение двух матриц размером п х п. Существует две подходящих единицы измерения размера данной задачи. Первая и наиболее часто используемая — это порядок матрицы п. Однако существует и другая, не менее подходящая единица измерения, — количество N перемножаемых элементов в матрицах. Последняя единица к тому же более универсальна, поскольку ее можно применять не только к квадратным матрицам. Поскольку существует простая формула, связывающая эти две единицы измерения, мы легко можем перейти от одной единицы к другой, однако искомая эффективность алгоритма будет в значительной степени отличать­ся в зависимости от того, какая из двух единиц измерения была использована при решении задачи (см. задачу 2 упражнения 2.1).

На выбор подходящей системы измерений размера задачи могут повлиять вы­полняемые рассматриваемым алгоритмом действия. Например, как оценить раз­мер входных данных для алгоритма, выполняющего проверку орфографии? Если в алгоритме проверяется каждый вводимый символ, то оценить размер входных данных мы можем, подсчитав количество символов во входном потоке. Если же в алгоритме происходит обработка текста по словам, то нужно подсчитать коли­чество слов во входном потоке.

Мы должны сделать специальное замечание по поводу оценки размера вход­ных данных для алгоритмов, связанных с нахождением чисел, удовлетворяющих определенным условиям (например, проверяющих, является ли заданное целое число п простым). Для подобных алгоритмов кибернетики предпочитают оцени­вать размер входных данных по количеству битов b в двоичном представлении числа п:

Ь= |\_log2nJ + 1. (2.1)

Подобная система измерений позволяет лучше оценить эффективность рас­сматриваемого алгоритма.

Единицы измерения времени выполнения алгоритма

Мы должны рассмотреть еще один вопрос, касающийся единиц измерения времени выполнения алгоритма. Безусловно, для этой цели можно просто вос­пользоваться общепринятыми единицами измерения времени — секундой, милли­секундой и т.д. и с их помощью оценить время выполнения программы, реали­зующей рассматриваемый алгоритм. Однако у такого подхода существуют явные недостатки, поскольку результаты измерений будут зависеть от:

* быстродействия конкретного компьютера;
* тщательности реализации алгоритма в виде программы;
* типа компилятора, использованного для генерации машинного кода;
* точности хронометрирования реального времени выполнения про­граммы.

Поскольку перед нами стоит задача измерения эффективности алгоритма, а не реализующей его программы, мы должны воспользоваться такой системой измерений, которая бы не зависела от приведенных выше посторонних факторов.

Один из возможных способов решения этой проблемы состоит в подсчете то­го, сколько раз выполняется каждая операция алгоритма. Однако подобный подход слишком сложен и, как мы увидим в дальнейшем, чаще всего не нужен. Поэтому мы должны составить список наиболее важных операций, выполняемых в ал­горитме, называемых основными, или базовыми операциями (basic operation), определить, какие из них вносят наибольший вклад в общее время выполнения алгоритма, и вычислить, сколько раз эти операции выполняются.

Как правило, составить список основных операций алгоритма совсем нетруд­но. Обычно в него включают наиболее длительные по времени операции, выпол­няемые во внутреннем цикле алгоритма. Например, в большинстве алгоритмов сортировки используется метод сравнения двух элементов (ключей) списка, ко­торый сортируется. Для подобного типа алгоритмов основной является операция сравнения ключей. В качестве еще одного примера рассмотрим алгоритмы пере­множения матриц и вычисления значения многочлена. В них используются две основные операции: умножение и сложение. На большинстве компьютеров коман­да умножения двух целых чисел выполняется намного дольше, чем сложение[[18]](#footnote-18). Поэтому она является безусловным кандидатом на включение в список основных операций.

Таким образом, закладывая основу для анализа временной эффективности ал­горитмов, мы предполагаем, что этот показатель будет оцениваться по количеству основных операций, которые должен выполнить алгоритм при обработке вход­ных данных размера п. О том, как определить этот показатель для нерекурсивных и рекурсивных алгоритмов, вы узнаете из разделов 2.3 и 2.4, соответственно.

Рассмотрим один важный пример. Предположим, что сор — время выполнении основной операции алгоритма на конкретном компьютере, а С (п) — количество раз, которые эта операция должна быть выполнена при работе алгоритма. Тогда время выполнения программной реализации этого алгоритма на данном компью­тере Т (п) можно приблизительно определить по следующей формуле:

Т (п) « СорС (п).

Разумеется, эту формулу нужно использовать очень аккуратно. В значении счетчика С (п) не учитывается количество выполняемых алгоритмом операций, не относящихся к основным. Кроме того, обычно значение этого счетчика так­же можно определить только приблизительно. Более того, значение константы Сор также можно определить лишь приблизительно и оценить ее точность весьма непросто. Тем не менее, эта формула дает приемлемую оценку времени выпол­нения алгоритма для не бесконечно больших и не бесконечно малых значений п. Она также позволяет ответить на вопросы наподобие: во сколько раз быстрее будет работать реализация данного алгоритма на компьютере, быстродействие ко­торого больше нашего в 10 раз? Казалось бы, ответ очевиден — в 10 раз. Или пусть, например, С (и) = п (п — 1)/2. Насколько дольше будет выполняться про­грамма, если удвоить размер входных данных? Ответ будет такой: приблизительно в четыре раза медленнее. В самом деле, для достаточно больших п справедлива следующая формула:

С(п) = \п (п — 1) = ^п2 — \п « ^п2. v 1 2 v J 2 2 2

Поэтому

Г(**2**п) \_ ***СорС(2п) \_*** J(**2**n**)2 *Т(п)*** ~ ***СорС(п***) ~ In**2**

Обратите внимание, что для ответа на второй вопрос не нужно знать реальное значение с^, поскольку в приведенной выше дроби оно сокращается. Обратите также внимание, что постоянный множитель 1/2 в формуле для С (п) тоже со­кращается. По этим причинам в процессе анализа эффективности при достаточно больших размерах входных данных не учитывают постоянные множители, а со­средотачиваются на оценке порядка роста {order of growth) количества основных операций с точностью до постоянного множителя.

Порядок роста

Почему выше мы сделали замечание по поводу вычисления порядка роста ко­личества основных операций алгоритма для достаточно больших размеров вход­ных данных? Дело в том, что при малых размерах входных данных невозможно заметить разницу во времени выполнения между эффективным и неэффективным алгоритмом. Например, при вычислении НОД двух небольших чисел, совершен­но непонятно во сколько раз алгоритм Евклида работает быстрее двух других алгоритмов, рассмотренных в разделе 1.1. Непонятным остается также вопрос, почему нас так волнует, какой из алгоритмов быстрее и во сколько раз. И только тогда, когда нужно вычислить НОД двух очень больших чисел, все эти вопросы, связанные с разной эффективностью алгоритмов, выходят на первый план и ста­новятся понятными. Для больших значений п вычисляют порядок роста функции. В табл. 2.1 эти значения приведены для некоторых функций, играющих особую роль в процессе анализа алгоритмов.

Таблица 2.1. Приближенные значения важных для анализа алгоритмов функций

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| п | log2n | п | п log2 п | п2 | п3 | 2" | п\ |
| 10 | 3.3 | 10 | 3.3 • 101 | 102 | 103 | 103 | 3.6 • 10е |
| 102 | 6.6 | 102 | 6.6 • 102 | 104 | 106 | 1.3- ю30 | 9.3 • 10157 |
| 103 | 10 | 103 | 1.0 • 104 | 106 | ю9 |  |  |
| 104 | 13 | 104 | 1.3 • ю5 | 108 | ю12 |  |  |
| ю5 | 17 | 105 | 1.7 -106 | Ю10 | ю15 |  |  |
| 106 | 20 | 106 | 2.0 • 107 | ю12 | ю18 |  |  |

Порядок чисел, приведенных в табл. 2.1, имеет чрезвычайное значение для анализа алгоритмов. Как видно из таблицы, самый малый порядок роста имеет логарифмическая функция. Причем его значение настолько мало, что программы, реализующие алгоритмы с логарифмическим количеством основных операций, будут выполняться практически мгновенно для всех диапазонов входных данных реального размера. Обратите также внимание, что, хотя некоторые значения при таких вычислениях, естественно, будут зависеть от основания логарифма, приве­денная ниже формула позволяет легко переходить от одного основания логарифма к другому, сохраняя при этом логарифмическую зависимость вычислений (при этом используются новые постоянные множители):

loga П = logQ Ь log6n.

Вот почему в случае, когда нужно только определить порядок роста количества основных операций алгоритма с точностью до постоянного множителя, мы будем опускать основание логарифма и записывать просто: log п.

Существует и другая крайность: показательная функция 2П и функция вычис­ления факториала п\. Обе эти функции имеют настолько высокий порядок роста, что его значение становится астрономически большим уже при умеренных зна­чениях п. (По этой причине мы не стали приводить в табл. 2.1 значения порядка роста этих функций при п > 102.) Например, чтобы выполнить 2100 операций компьютеру, имеющему производительность в один триллион операций (1012) в секунду, понадобиться без малого 4 • Ю10 лет! Однако это ничто по сравнению со временем, которое затратит тот же компьютер на выполнение 100! операций.

Его даже нельзя сравнить со временем жизни планеты Земля, которое, по прибли­зительным оценкам, составляет 4.5 миллиарда (4.5 • 109) лет. Несмотря на суще­ственную разницу между порядком роста показательной функции 2П и функции п!, довольно часто говорят, что обе функции имеют экспоненциальный порядок роста. Однако, строго говоря, такое утверждение верно только для первой из этих функций. Подводя итог, можно сформулировать следующий вывод, который всегда важно помнить.

С помощью алгоритмов, в которых количество выполняемых операций растет по экспоненциальному закону, можно решить лишь задачи очень малых размеров.

Существует еще один способ оценки качественного различия в порядке роста функций, приведенном в табл. 2.1. Необходимо рассмотреть реакцию функции на, скажем, двукратное увеличение значения параметра п. Для функции log2 п это приведет к увеличению значения всего на 1, так как log2 2п = log2 2 + log2 п = = 1 + log2 п. Линейная функция увеличит значение в два раза. Значение функции п log2 п увеличится чуть больше, чем в два раза. Квадратичная п2 и кубическая п3 функции увеличатся в четыре и восемь раз, соответственно, поскольку (2п)2 = = 4п2 и (2п)3 = 8п. Значение функции 2п увеличится в квадрате, поскольку 22п = (2П)2. Что касается функции п!, то ее значение увеличится намного больше, чем значение любой из рассмотренных здесь функций.

Эффективность алгоритма в разных случаях

В начале этого раздела мы определили, что имеет смысл оценивать эффектив­ность алгоритма как функцию от некоторого параметра п, связанного с размером входных данных. Однако существует большое количество алгоритмов, время вы­полнения которых зависит не только от размера входных данных, но также и от особенностей конкретных входных данных. В качестве примера рассмотрим зада­чу последовательного поиска. Она решается с помощью довольно простого алго­ритма, который выполняет поиск заданного элемента (ключа поиска К) в списке, состоящем из п элементов, путем последовательного сравнения ключа К с каж­дым из элементов списка. Работа алгоритма завершается, либо когда заданный ключ найден, либо когда весь список исчерпан. Ниже этот алгоритм описан на псевдокоде. В нем для простоты полагается, что список задан в виде массива чисел. (Кроме того, предполагается, что второе условие А [г] ф К не будет про­веряться в случае, если не выполняется первое условие, в котором проверяется, не выходит ли индекс массива за его верхнюю границу.)

**Алгоритм *Sequential Search*** **(*А*** **[0..п** - **1]**, ***К)***

П Входные данные: массив чисел А[0..п — 1] и ключ поиска К

П Выходные данные: возвращается индекс первого элемента // массива А, который равен К, либо —1,

// если заданный элемент не найден

г О

**while** г < **n** **and A[i] ф К** **do i <— г +** **1 if** г < **n return** г **else**

**return —1**

Совершенно очевидно, что время работы этого алгоритма может отличаться в очень широких пределах для одного и того же списка размера п. В наихудшем случае, т.е. когда в списке нет искомого элемента либо когда искомый элемент расположен в списке последним, в алгоритме будет выполнено наибольшее коли­чество операций сравнения ключа со всеми п элементами списка: Cworst (^) = Под эффективностью алгоритма в наихудшем случае (worst-case efficiency) подразумевают его эффективность для наихудшей совокупности входных дан­ных размером п, т.е. для такой совокупности входных данных размером п среди всех возможных, для которой время работы алгоритма будет наибольшим. Спо­соб определения эффективности алгоритма для наихудшего случая в принципе несложен: нужно проанализировать алгоритм и выяснить, для каких из возмож­ных комбинаций входных данных размером п значение числа основных операций алгоритма С (п) будет максимальным, а затем вычислить значение Cworst (п) для наихудшего случая. (Что касается алгоритма последовательного поиска, то здесь ответ очевиден. Методы исследования более сложных ситуаций будут описаны в последующих разделах этой главы.) Очевидно, что анализ эффективности ал­горитма для наихудшего случая позволяет получить очень важную информацию о быстродействии алгоритма в целом, поскольку в данном случае речь идет о мак­симально возможном времени его выполнения. Другими словами, он гарантирует, что для любых исходных данных размером п время выполнения алгоритма не будет превышать максимально возможного значения Cworst (п), получаемого для наихудшей совокупности входных данных.

Под эффективностью алгоритма в наилучшем случае (best-case efficiency) подразумевают его эффективность для наилучшей совокупности входных данных размером п, т.е. для такой совокупности входных данных размером п среди всех возможных, для которой время работы алгоритма будет наименьшим. Соответ­ственно, эффективность алгоритма для наилучшего случая можно проанализиро­вать следующим образом. Сначала нужно определить, для каких из возможных комбинаций входных данных размером п значение числа основных операций ал­горитма С (п) будет наименьшим. (Обратите внимание, что наилучший случай вовсе не означает случай, когда вводится минимальное количество данных. Он означает комбинацию входных данных размером п, при обработке которых алго­ритм будет работать быстрее всего.). Затем мы должны определить само значение Cbest (п) для такого случая. Например, для алгоритма последовательного поис­ка наилучшим случаем входных данных будет такой список размером п, первый элемент которого равен ключу поиска К. Поэтому для него Cbest (п) = 1-

Анализ эффективности алгоритма для наилучшего случая не так важен, как для наихудшего случая, хотя его нельзя назвать совсем уж бесполезным. При ана­лизе алгоритма не стоит полагаться на то, что обрабатываемые им данные будут соответствовать наилучшему случаю. Тем не менее нужно учитывать тот факт, что для некоторых алгоритмов их высокое быстродействие для наилучшего случая со­храняется и для случаев близких к наилучшему. Например, существует алгоритм сортировки методом вставок, для которого наилучший случай входных данных со­ответствует заранее отсортированному массиву. При этом скорость работы такого алгоритма будет наибольшей. Причем она лишь ненамного ухудшается в слу­чае, если входной массив почти отсортирован. Следовательно, такой алгоритм очень хорошо подойдет для случаев, когда приходится иметь дело с почти отсор­тированными массивами. Ну и конечно же, если эффективность алгоритма для наилучшего случая является неудовлетворительной, не имеет смысла продолжать его дальнейший анализ.

Как бы там ни было, из данного описания уже должно быть понятно, что на основании информации, полученной в результате анализа алгоритма для наилуч­шего и наихудшего случаев, нельзя сделать вывод о том, как поведет себя алгоритм при обработке типовых или случайно заданных входных данных. Чтобы получить подобную информацию, нужно выполнить анализ алгоритма для среднего случая {average-case efficiency). Для этого нужно сделать ряд предположений о совокуп­ности входных данных размером п.

Давайте продолжим рассмотрение алгоритма поиска методом последователь­ного перебора. Сделаем два стандартных предположения:

1. вероятность успешного поиска равна р, где 0 ^ р ^ 1;
2. вероятность первого совпадения ключа с г-м элементом списка одина­кова для любого г.

Исходя из этих предположений (несмотря на всю очевидность, их справедли­вость обычно трудно доказать), можно вычислить среднее количество операций сравнения с ключом CaVg (п), как описано ниже. Если совпадение с ключом най­дено, вероятность того, что это произошло именно на г-м элементе списка, равна р/п для любого г. При этом количество операций сравнения, выполняемых в ал­горитме в подобном случае, очевидно и равно г. Если совпадение с ключом не найдено, количество операций сравнения равно п, а вероятность этого события равна (1 — р). Поэтому

Cava {п) — [l • — + 2 • — Н \-i-~ Л bn-— + тг • (1 — p) =

L n n n nJ

= — [1 + 2 H HH n] + n • (1 — p) =

***n***

***pn(n +*1**) ***p{n +*1**) .

= ^ ***'- + n-(l-p) = -±-—J- + n-{l-p).***

***П 2 2***

Эта обобщенная формула позволяет ответить на несколько вполне уместных вопросов. Например, если р = 1 (т.е. искомый элемент, равный ключу, точно при­сутствует в списке и поэтому будет найден), среднее количество операций срав­нения при поиске методом последовательного перебора будет равно (п+ 1)/2. Другими словами, в среднем, в алгоритме проверяется приблизительно полови­на элементов списка. Если р = 0 (т.е. искомого элемента в списке нет), среднее количество операций сравнения будет равно п, поскольку при этом в алгоритме проверяются все п элементов списка.

Как видно из этого очень простого примера, определение эффективности алго­ритма для среднего случая куда более трудная задача, чем для наихудшего и наи­лучшего случаев. При использовании прямого метода для решения этой задачи, необходимо разбить всю совокупность экземпляров входных данных размером п на несколько групп так, чтобы для каждой группы число основных операций, выполняемых алгоритмом, было одинаково. (Сможете ли вы выделить эти груп­пы для алгоритма поиска методом последовательного перебора?) Затем нужно найти или придумать функцию распределения вероятностей для входных данных и с ее помощью вычислить ожидаемое значение количества основных операций алгоритма. Технически реализовать этот план не составит большого труда, хо­тя проверить вероятностные предпосылки, лежащие в его основе, обычно очень тяжело. Поэтому для простоты, когда речь будет заходить об анализе эффективно­сти для среднего случая, мы чаще всего будем приводить уже готовые результаты для обсуждаемого алгоритма. Если вас заинтересует источник этих результатов, рекомендуем обратиться к таким книгам, как [8, 65, 66, 67] и [105].

Нужны ли реально кому-нибудь данные об эффективности алгоритма для сред­него случая? Ответ на этот вопрос не вызывает сомнений — конечно же да! Су­ществует большое число важных алгоритмов, эффективность которых гораздо выше для среднего случая, чем для излишне пессимистичного наихудшего слу­чая, что привлекает к ним практический интерес. Таким образом, информация об эффективности алгоритма для среднего случая позволяет принять правильное ре­шение и не упустить много важных алгоритмов. Наконец, из приведенного выше описания уже должно быть понятно, что эффективность алгоритма для среднего случая нельзя получить, усреднив его эффективности для наихудшего и наилуч­шего случаев. Даже если иногда полученные таким образом результаты совпадут с истинными данными для среднего случая, такой способ анализа не является законным.

Существует еще один вид эффективности, называемый амортизированной эффективностью {amortized efficiency). Она предназначена не столько для ана­лиза общего времени выполнения алгоритма, сколько для анализа последователь­ности операций, выполняемых над одной и той же структурой данных. Бывает так, что одна из операций алгоритма может оказаться слишком продолжительной, но общее время выполнения последовательности из п таких операций будет зна­чительно меньше, чем его оценка, выполненная для наихудшего случая, равная произведению максимального времени выполнения этой операции на число таких операций, п. Поэтому мы можем разделить (или “амортизировать”) высокую сто­имость выполнения алгоритма для наихудшего случая на всю последовательность выполняемых операций по аналогии с тем, как в экономике разносят стоимость дорогостоящего оборудования на весь период его эксплуатации. Этот нетриви­альный подход был предложен американским кибернетиком Робертом Таржаном (Robert Taijan). Он использовал его наряду с другими методами для исследования интересного варианта классического двоичного дерева поиска. (Очень интересное описание этого метода, лишенное технических деталей, приведено в статье Тар- жана [117], а в статье [116] этот же метод рассмотрен со всеми подробностями.) Полезный пример применения амортизированной эффективности будет приведен в разделе 9.2, где мы будем рассматривать алгоритмы объединения непересекаю- щихся множеств.

Повторение пройденного

Давайте в заключение этого раздела еще раз отметим основные моменты, на которых мы останавливались ранее.

* И временная, и пространственная эффективности оцениваются в виде функции от некоторого параметра п, связанного с размером входных данных.
* Временная эффективность измеряется путем подсчета числа основ­ных операций, выполняемых в алгоритме. Пространственная эффек­тивность оценивается в виде количества дополнительных единиц па­мяти, требуемых для работы алгоритма.
* Эффективность отдельных алгоритмов может различаться в очень ши­роких пределах для разных наборов входных данных одинакового раз­мера. Поэтому эффективность таких алгоритмов оценивают для наи­худшего, наилучшего и типичного набора входных данных.
* Основной целью анализа эффективности алгоритмов является опреде­ление порядка роста времени выполнения алгоритма (а также количе-

image28

ства дополнительных единиц памяти) в случае, когда размер входных данных стремится к бесконечности.

В следующем разделе будут рассмотрены строгие методы определения поряд­ка роста. В разделах 2.3 и 2.4 мы обсудим особенности методов анализа нере­курсивных и рекурсивных алгоритмов. Мы увидим, как применить на практике полученные в этом разделе базовые знания для анализа эффективности конкрет­ных алгоритмов. Остальные примеры будут приведены в других главах книги.

Упражнения 2.1

и можете проверить, что именно вы выбрали, только после того, как выбор сделан. Чему равно минимальное количество перчаток, которые надо взять из ящика, чтобы получить как минимум одну пару перчаток одинакового цвета? Дайте ответ на этот вопрос для наилучшего и наихудшего случая. (Задача взята из [80], №18.)

б) Предположим, у вас есть 5 разных пар носков, и вы обнаружили, что два носка потеряны. Естественно, вы заинтересованы в том, чтобы в результате осталось наибольшее количество полных пар, посколь­ку носить разные носки вы не будете. Поэтому в лучшем случае у вас останется 4 полных пары, а в худшем — только 3. Предположим, вы имеете равные шансы потерять любой из 10 носков. Определите вероятности наилучшеш и наихудшего случаев, а также количество пар носков, которые остаются у вас в среднем случае. (Задача взята из [80], №48.)

image29

1. а) Докажите формулу (2.1), определяющую количество битов в двоич­

ном представлении положительного десятичного целого числа.

б) Приведите аналогичную формулу для количества десятичных цифр.

в) Объясните, почему в рамках принятой нами модели основных по­нятий, не имеет значения, какие цифры (двоичные или десятичные) мы используем для указания размера п входных данных.

1. Как, по вашему мнению, следует дополнить любой из алгоритмов сор­тировки, чтобы количество операций сравнения его ключей в лучшем случае составляло п — 1, где п, как обычно, размер списка. Как вы ду­маете, будет ли подобное дополнение уместно для любого алгоритма сортировки?
2. Для решения системы из п линейных уравнений, где п — произвольное целое число, используется классический алгоритм последовательного исключения Гаусса, в котором выполняется п3/3 умножений, являю­щихся основной операцией алгоритма.

а) Определите, во сколько раз дольше будет работать алгоритм Гаусса для решения системы из 1000 уравнений по сравнению со временем решения системы из 500 уравнений.

б) Предположим, вы собираетесь приобрести компьютер, быстродей­ствие которого превосходит в 1000 раз быстродействие того ком­пьютера, на котором вы сейчас работаете. Определите, на сколько порядков большую систему уравнений сможет решить за одно и то же время новый компьютер по сравнению со старым.

1. Для каждой из приведенных ниже функций определите, во сколько раз изменится ее значение при увеличении значения аргумента в четыре раза.
2. log2n б) у/п в) п г) п2

д) п3 е) 2п

1. Для каждой из приведенных ниже пар функций определите, соответ­ствует ли первая функция порядку роста второй функции (с точностью до постоянного множителя), меньше его или больше.

а) п (п + 1) и 2000п2 б) 100п2 и O.Oln3

в) log2 п и In п г) log2 п и log2 п2

image30

д) 2П-1 и 2п е) (п - 1)! и п!

1. Согласно древней легенде, игру в шахматы придумал много столетий тому назад мудрец, проживавший в северо-западной Индии. Он по­казал эту игру своему повелителю. Ему она понравилась настолько, что он предложил мудрецу любую награду, которую тот только по­желает. Мудрец попросил повелителя выдать ему в качестве награды несколько хлебных зерен, количество которых должно определяться по следующему правилу. На первую клетку шахматной доски кладется одно зернышко, на вторую — два, на третью — четыре, на четвертую — восемь, на пятую — 32 и так до тех пор, пока все 64 клетки доски не будут заполнены. Какое окончательное количество зерен должен был получить мудрец согласно описанному алгоритму?
2. Асимптотические обозначения

и основные классы эффективности

Как отмечалось в предыдущем разделе, при изложении основ анализа эффек­тивности алгоритмов основное внимание было сосредоточено на порядке роста количества базовых операций алгоритма, который мы использовали в качестве основного критерия эффективности алгоритма. Для того чтобы можно было срав­нивать между собой эти порядки роста и классифицировать их, ученые ввели три условных обозначения: О (прописная “О”), 0 (прописная греческая “тета”) и О, (прописная греческая “омега”). Для начала дадим нестрогое описание этих поня­тий, а затем, рассмотрев несколько примеров, приведем их строгое определение. В приведенном ниже описании t (п) и д (п) могут быть любыми неотрицательны­ми функциями, определенными на множестве натуральных чисел. Через t (п) мы обозначили время выполнения алгоритма. Обычно оно выражается в виде числа основных операций алгоритма, обозначаемых через С (п). Под д (п) мы будемпонимать некоторую простую функцию, с которой будет проводиться сравнение количества операций.

Нестрогое введение

Говоря нестрого, обозначение О (д (п)) — это множество всех функций, поря- док роста которых при достаточно больших п не превышает (т.е. меньше или ра­вен) некоторую константу, умноженную на значение функции д (п). Вот несколько примеров — все приведенные ниже утверждения справедливы:

п € О (п2) , 100п + 5 € О (п2) , п(п-1)/2еО (п2) .

В самом деле, первые две функции линейны, поэтому их порядок роста заведо­мо меньше, чем д (п) = п2. Последняя функции является квадратичной, поэтому ее порядок роста такой же, как у функции п2. С другой стороны,

п3 £ О (n2) , O.OOOOln3 ф О (п2), п4 + п + 1 ф О (п2) .

В самом деле, обе функции п3 и O.OOOOln3 являются кубическими, поэто­му имеют более высокий порядок роста, чем функция п2. Те же рассуждения справедливы и для полинома четвертой степени пА + п + 1.

Второе обозначение, £) (д (п)), — это множество всех функций, порядок роста которых при достаточно больших п не меньше (т.е. больше или равен) некоторой константы, умноженной на значение функции д (п). Например:

п3 € (n2) , п (п — 1)/2 € (п2) , но 100п + 5 ф ft (п2) .

Наконец, обозначение 0 (д (п)) — это множество всех функций, порядок роста которых при достаточно больших п равен некоторой константе, умноженной на значение функции д (п). Таким образом, любая квадратичная функция ап2 + Ъп + + с при а > 0 принадлежит множеству 0 (п2). Кроме того, среди бесконечного количества других функций множеству © (п2) принадлежат также функции п2 + + sin п и п2 + log п (можете объяснить, почему?).

Надеюсь, что приведенное выше нестрогое описание позволит вам глубже понять идею, лежащую в основе трех асимптотических обозначений. А теперь перейдем к строгим определениям.

О-обозначение

Определение 1. Говорят, что функция t (п) принадлежит множеству О (д (п)), что записывается как t (п) € О (д (п)), если для всех больших значений п функция t (п) ограничена сверху некоторой константой, умноженной на значение функции д (п), т.е. если существует положительная константа с и неотрицательное целое число по такое, что t (п) ^ сд (п) для всех п ^ по- ■

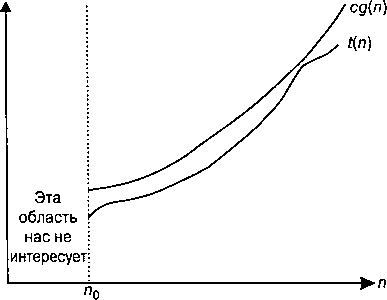


Рис. 2.1. Иллюстрация О-обозначения:

***t{n)eO(g (п))***

Это определение проиллюстрировано на рис. 2.1. Чтобы сделать рисунок на­гляднее, мы использовали вещественный тип числа п.

В качестве примера давайте приведем строгое доказательство одного из утвер­ждений, приведенных во введении к этой главе: 100п + 5 € О {п2). В самом деле, для всех п ^ 5 справедливо следующее выражение:

100п + 5 < 100п + п = 101 п ^ 101п2.

Таким образом, для данного случая в качестве констант с и по, о которых говорится в приведенном выше определении, можно, соответственно, взять 101 и 5.

Заметьте, что в определении ничего не говорится о том, какие именно значения констант с и по мы должны выбирать. Поэтому приведенный выше пример можно доказать и так: для всех п ^ 1 справедливо выражение

100п + 5 < 100п + 5п = 105п.

Из него следует, что в данном случае с = 105, a no = 1.

^-обозначение

Определение 2. Говорят, что функция t(n) принадлежит множеству £l(g(n)), что записывается как t(n)eQ (д (п)), есЛи для всех больших значений п функция t (п) ограничена снизу некоторой константой, умноженной на значение функции д{п), т.е. если существует положительная константа с и неотрицательное целое число по, такое, что t (п) ^ сд (п) для всех n ^ по- ■

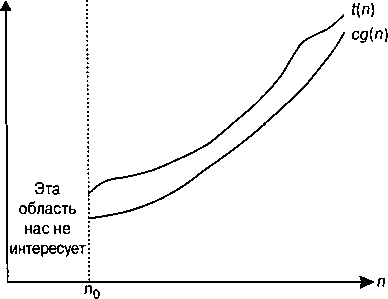


Рис. 2.2. Иллюстрация П-обозначения:

***t{n) £0.(д*** (***п***))

Это определение проиллюстрировано на рис. 2.2.

В качестве примера приведем строгое доказательство того, что n3 G (п[[19]](#footnote-19)). В самом деле, для всех п ^ 0 выполняется неравенство п3 ^ п2. Таким образом, для данного случая можно положить с = 1 и по — 0.

©-обозначение

Определение 3. Говорят, что функция t(n) принадлежит множеству 0(д(п)), что записывается как t (n) Е © (д (п)), если для всех больших значений п функ­ция t (п) ограничена снизу и сверху некоторыми константами, умноженными на значения функции д (п), т.е. если существуют положительные константы с\ и С2, а также неотрицательное целое число по такое, что С2д (п) < t (п) < с\д (п) для всех п ^ по. ■

Это определение проиллюстрировано на рис. 2.3.

В качестве примера приведем строгое доказательство того, что п(п — 1)/2 е е © (п2). Сначала, давайте докажем правую часть неравенства (т.е. определим верхнюю границу). Для всех п ^ 0 справедливо следующее:

1 / 1 2 1 1 2 -п(п- 1) = -п - -п ^ -п .

Затем докажем левую часть неравенства (т.е. определим нижнюю границу). Для всех п ^ 2 справедливо следующее:

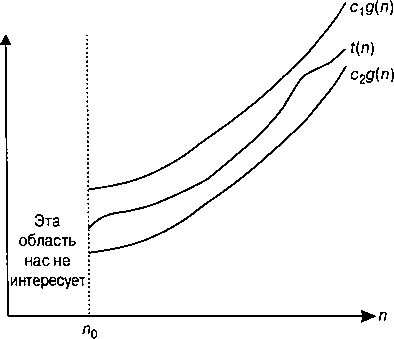


Рис. 2.3. Иллюстрация ©-обозначения: i (га) € © (д (п))

Полезные свойства сделанных асимптотических обозначений

С помощью сделанных выше строгих определений асимптотических обозна­чений можно доказать ряд их основных свойств. (Несколько простых примеров были приведены в задаче 1 упражнения 2.1). В частности, приведенные ниже свойства пригодятся при анализе алгоритмов, состоящих из двух исполняемых частей.

Теорема 1. Если t\ (п) € О (д\ (п)) и £2 (п) € О (<72 (^)), то t\ (п) + £2 {р) € О (max{gi(n),p2(™)})- (Аналогичные утверждения справедливы также для мно­жеств £) и 0).

Доказательство. (Как мы увидим, доказательство этой теоремы представляет собой расширение на порядок роста следующего простого соотношения для про­извольных вещественных чисел аь &ь аг и 62: если а\ < Ъ\ и a<i ^ 62, то а\ + аъ ^ 2 max {61,62}-) Поскольку £1 (п) £ О (п)), существует константа ci и неотрицательное целое число n\ такие, что для всех п ^ п\ справедливо ti (га) < Cigi (п).

По аналогии, поскольку ^ (га) € О (<72 (га)), существует константа сг и неот­рицательное целое число П2 такие, что для всех га ^ пг справедливо <2 (га) ^ ^ С292 (га).

Давайте обозначим сз = max (ci, С2} и рассмотрим га ^ max (rai, гаг}, т.е. ко­гда верны оба неравенства. Сложив приведенные выше неравенства, получимследующее:

h (п) + t2 (п) ^ agi (п) + С2Я2 (п) ^

< ***сз****91* ***(п) + с3д2*** (***п) = с3 [gi*** (***п***) + ***д2*** (п)] <

^ c32max{c/i (п),д2 (п)} .

Отсюда следует, что t\ (n) + t<i (n) Е О (шах {д\ (п), #2 (п)})- Исходя из опре­деления множества О, в качестве констант с и по в данном случае нужно выбрать: с = 2сз = 2max{ci,C2} и по = max{ni,n2}. ■

***h*** (***п)*** € ***О (gi*** (п)) i**2** ***(тг)еО(д2(п))***

Итак, что же означает это свойство с точки зрения анализа эффективности алгоритмов, состоящих из двух исполняемых частей? Оно означает, что общая эффективность алгоритма зависит от той части, которая имеет наибольший поря­док роста, т.е. от наименее эффективной его части:

h (п) + £2 (гг) € О (так {gi (п), д2 (п)}).

Например, проверить, содержит ли массив повторяющиеся элементы, можно с помощью описанного ниже двухэтапного алгоритма. Сначала нужно отсортиро­вать массив, воспользовавшись одним из известных алгоритмов сортировки. За­тем нужно пройтись по всем элементам отсортированного массива и проверить, не дублируют ли соседние элементы друг друга. Если, к примеру, в алгоритме сор­тировки, используемом на первом этапе, выполняется не более чем п(п — 1)/2 операций сравнения (и поэтому он принадлежит множеству О (п,2)), а в алгорит­ме, используемом на втором этапе, выполняется не более чем п — 1 операций сравнения (т.е. он принадлежит множеству О (п)), суммарная эффективность об­щего алгоритма будет принадлежать множеству О (max {п, п2}) — О (п,2).

Использование пределов для сравнения порядка роста двух функций

Несмотря на то что без строгих определений множеств О, £), © нельзя обой­тись при доказательстве их абстрактных свойств, они редко используются при сравнении порядков роста двух конкретных функций. Дело в том, что существу­ет более удобный метод выполнения этой оценки, основанный на вычислении предела отношения двух рассматриваемых функций. Может существовать три

*у* t(n)

hm “ГГ =

***п* ►сх) *д (п)***

основных случая[[20]](#footnote-20):

О если t (п) имеет меньший порядок роста, чем д (п) с если t (п) имеет тот же порядок роста, чем д (п)

^оо если t (п) имеет больший порядок роста, чем д (п).

Обратите внимание, что для двух первых случаев t (п) € О (д (п)), для двух последних случаев t (n) € Cl (д (п)), а для второго случая t (п) € © (д (п)).

Методы, основанные на вычислении пределов, зачастую более удобны для анализа алгоритмов, чем описанные выше методы, основанные на определении множеств. Дело в том, что в первом случае можно воспользоваться преимущества­ми мощного математического аппарата, специально созданного для вычисления пределов, в частности правилом Лопиталя (L’Hopital)

)im Щ = ita ш

п >оо д (п) п-\*оо д' (п) и формулой Стирлинга (Stirling):

/ fi \ п

п\ « у2тт у—J для достаточно больших п.

Ниже приведены примеры использования пределов для сравнения порядка роста двух функций.

Пример 1. Сравните порядки роста функций п (п — 1)/2 и п2. (Это один из тех примеров, который мы использовали выше для иллюстрации определений мно­жеств.)

Um Мп-1У2 = 1 Цт n^-n = 1 Um Л Г) \*

п—\*оо 7lZ 2 71—ЮО п 2 71—► оо Y п J 2

Поскольку предел равен положительной константе, обе функции имеют оди­наковый порядок роста, что записывается как п (п — 1)/2 € © (^2). ■

Пример 2. Сравните порядки роста функций log2 п и у/п. (В отличие от приме­ра 1 решение данной задачи не столь очевидно.)

logon (logon/ (log2e)^ Jn

lim -Ц=- = lim >■■■■?-?-/- = lim Ц-Ln. = 21og2e lim — = 0.

71-Ю0 y/n 71—► OO [у/й) 71—Ю0 71—ЮО П

Поскольку предел равен нулю, функция log2 п имеет меньший порядок роста, чем у/п. (Так как lim ■°ёЯ-п = 0, то для этого случая можно использовать так

п—► оо **У 71**

называемую форму записи “с маленькой о”: log2 пЕо (у/п). В отличие от пропис­ной греческой “О” при анализе алгоритмов строчная “о” используется довольно редко.) ■

Пример 3. Сравните порядки роста функций п! и 2п. (С этим примером мы уже встречались в предыдущем разделе.) Воспользовавшись формулой Стирлинга, получим:

п\ **^/27гn(^)n ^—** Пп я—(П\п

**lim — = lim ^ - = lim v 27Гn- = lim v** 2тгп **— = oo.**

n—\*oo 2n ?!—►ex) 2n n—\*oo 2nen п—юо \2e)

Следовательно, хотя функция 2n растет очень быстро, функция п! растет еще быстрее, что записывается так: n! € f) (2П). Обратите внимание, что при использо­вании этой формы записи читатель, в отличие от случая использования пределов, не может сделать однозначный вывод о том, что порядок роста функции п! выше, чем функции 2П, а не, скажем, равен ей. ■

Основные классы эффективности

Хотя в основах анализа алгоритмов к одному классу относят все функции, чей порядок роста одинаков с точностью до постоянного множителя, существует бес­конечное множество подобных классов. Например, порядок роста показательной функции ап зависит от значения основания а. Поэтому для вас может оказаться неожиданностью, что временную эффективность большого количества алгорит­мов можно отнести всего к нескольким классам. Эти классы, их названия, а также некоторые пояснения, приведены в табл. 2.2 в соответствии с возрастанием их порядка роста.

Таблица 2.2. Основные асимптотические классы эффективности

|  |  |
| --- | --- |
| Класс Название | Примечание |
| 1 Константа | Если не считать эффективности в наилучшем случае, в этот класс попадает очень небольшое количество ал­горитмов. Обычно при бесконечном увеличении разме­ра входных данных время выполнения алгоритма также стремится к бесконечности |

Окончание табл. 2.2

|  |  |  |
| --- | --- | --- |
| Класс | Название | Примечание |
| log п | Логарифмическая | Обычно такая зависимость появляется в результате со­кращения размера задачи на постоянное значение на каждом шаге итерации алгоритма (см. раздел 5.5). Обра­тите внимание, что в логарифмическом алгоритме невоз­можна работа со всеми входными данными (и даже с их некоторой фиксированной частью). В этом случае вре­мя выполнения любого алгоритма будет находиться по меньшей мере в линейной зависимости от размера вход­ных данных |
| п | Линейная | К этому классу относятся алгоритмы, выполняющие ска­нирование списка, состоящего из п элементов, например алгоритм поиска методом последовательного перебора |
| п log и | n-log-n | К этому классу относятся большое количество алгорит­мов декомпозиции (см. главу 4), таких как алгоритмы сортировки слиянием и быстрой сортировки |
| п2 | Квадратичная | Как правило, подобная зависимость характеризует эф­фективность алгоритмов, содержащих два встроенных цикла (см. следующий раздел). В качестве типичных примеров достаточно назвать простой алгоритм сорти­ровки и целый ряд операций, выполняемых над матри­цами размером пх п |
| п3 | Кубическая | Как правило, подобная зависимость характеризует эф­фективность алгоритмов, содержащих три встроенных цикла (см. следующий раздел). К этому классу относят­ся несколько довольно сложных алгоритмов линейной алгебры |
| 2п | Экспоненциальная | Данная зависимость типична для алгоритмов, выполня­ющих обработку всех подмножеств некоторого множе­ства, состоящего из п элементов. Часто термин ‘^экспо­ненциальный” используется в широком смысле и озна­чает очень высокие порядки роста, т.е. включает более быстрые по сравнению с экспонентой порядки роста |
| п! | Факториальная | Данная зависимость типична для алгоритмов, выполня­ющих обработку всех перестановок некоторого множе­ства, состоящего из п элементов |

Вы можете возразить, что классификация алгоритмов в соответствии с основ­ными асимптотическими классами эффективности имеет небольшое практиче­ское применение, поскольку обычно значения постоянных множителей никто не указывает. А это означает, что не исключена возможность, когда для входных дан­ных реального размера, алгоритм, относящийся к худшему классу эффективности, будет работать быстрее, чем алгоритм, относящийся к лучшему классу эффектив­ности. Например, если время выполнения одного алгоритма изменяется по закону п3, а другого — по закону Ю6/!2', кубический алгоритм будет работать быстрее при условии, что п не превышает 106. Известно несколько подобных аномалий. На­пример, существуют алгоритмы перемножения матриц, имеющих лучшую асимп­тотическую эффективность, чем у алгоритма, построенного на определении этой операции (см. раздел 4.5) и имеющего кубическую зависимость эффективности. Однако поскольку для таких алгоритмов значения постоянных множителей на­много большие, они могут представлять собой лишь теоретическую ценность.

К счастью, обычно значения постоянных множителей не отличаются настоль­ко сильно. Поэтому, как правило, можно предполагать, что алгоритм, относящийся к лучшему асимптотическому классу, будет работать быстрее алгоритма, относя­щегося к худшему асимптотическому классу, даже при обработке входных данных среднего размера. Это замечание главным образом справедливо для тех алгорит­мов, время выполнения которых растет не так быстро, как у экспоненциальных или факториальных алгоритмов.

Упражнения 2.2

1. Выберите подходящее обозначение О, © или fi, которое бы наиболее точно указывало на принадлежность к конкретному классу временной эффективности алгоритма поиска методом последовательного перебора (см. раздел 2.1):

а) для худшего случая;

б) для лучшего случая;

в) для среднего случая.

1. Используя нестрогие определения множеств О, © и £), определите,

истинны или ложны перечисленные ниже утверждения.

**ч п(п+1) ^** п(п +**1)** , **2ч**

**а) € О** (п3) **б)** € **О (п2)**

v *п(п+1) ^ ,* оч *п(п +*1) Л / ч

**В) v 2** ) ее (гг3) **г) v2**

1. Для каждой из приведенных ниже функций укажите класс © (д (п)), к которому относится функция. (При ответе используйте максимально простую функцию д (п)). Докажите ваши утверждения.

a) (n2 + l)10 б) \/10п2 + 7п + 3

в) 2п lg (n + 2)2 + (n + 2)2 lg (п/2)

г) 2п+1 + Зп 1 д) [log2 nj

1. а) В табл. 2.1 указаны значения функций, играющих особую роль

в процессе анализа алгоритмов. Их анализ делает очевидным, что функции logn, ?г, nlogn, п2, п3, 2П, п! перечислены согласно воз­растанию их порядка роста. Можно ли считать эти значения строгим математическим доказательством данного факта?

б) Докажите, что приведенный выше список функций в самом деле упорядочен в соответствии с их порядком роста.

1. Расположите перечисленные ниже функции в соответствии с их поряд­ком роста (от самого меньшего к самому большему).

(п — 2)!, 5lg (гг -h 100)10 , 22n, O.OOln4 + Зп3 + 1, 1п2п, у/п, Зп

1. а) Докажите, что любой многочлен вида р(п) = а^пк + аь-\пк~1 +

-I- Ь ао, где > 0, принадлежит множеству 0 (nfc).

б) Докажите, что порядок роста показательной функции ап зависит от значения ее неотрицательного основания а.

1. Используя определения рассматриваемых множеств, докажите или опровергните, приведя соответствующий контрпример, следующие ут­верждения:

а) Если t(n) еО (д (п)), то д (п) € £) (t (n))

б) £) (ад (n)) = Q (д (п)), где а > 0

в) **0** ***(д (п)) = 0(д (п)) ПП(д*** (***п***))

г) Для любых двух неотрицательных функций t (п) и д (п), определен­ных на множестве неотрицательных целых чисел, справедливо либо выражение t (п) € О (д (п)), либо выражение t(n) € £) (д (п)), либо оба эти выражения одновременно.

1. Докажите рассматриваемую в этом разделе теорему для:

а) fi-обозначений; б) ©-обозначений.

1. В этом разделе было сказано, что проверить, содержит ли массив по­вторяющиеся элементы, можно с помощью двухэтапного алгоритма, основанного на предварительной сортировке самого массива.

а) Если известно, что сортировка массива выполняется с помощью алгоритма, временная эффективность которого 0 (nlogn), к како­му классу временной эффективности будет относиться двухэтапный алгоритм в целом?

б) Если для работы алгоритма сортировки требуется дополнительный массив размером п, к какому классу пространственной эффективно­сти будет относиться двухэтапный алгоритм?

1. Вы подошли вплотную к стене, не имеющей ни начала, ни конца. Из­вестно, что в стене есть дверь, но вы не знаете в каком направлении и как долго нужно двигаться, чтобы найти эту дверь. Заметить дверь в стене можно, только подойдя непосредственно к ней и став напротив. Разработайте алгоритм, который бы позволял находить дверь в стене методом обхода не более чем за О (п) шагов, где п — не известное зара­нее число шагов между вашим первоначальным положением и дверью. (Задача взята из [88], №652.)

image34

1. Математический анализ нерекурсивных алгоритмов

В этом разделе мы систематически применим описанные в разделе 2.1 основы анализа эффективности алгоритмов для исследования нерекурсивных алгоритмов. Начнем с очень простого примера, на основе которого продемонстрируем все важные этапы, которые обычно выполняются при анализе подобных алгоритмов.

Пример 1. Рассмотрим задачу поиска наибольшего элемента в списке из п чисел. Для простоты предположим, что этот список реализован в виде массива. Ниже приведен псевдокод алгоритм решения этой задачи.

Алгоритм MaxElement (А [0..п - 1])

// Входные данные: массив вещественных чисел А[0..п — 1]

// Выходные данные: возвращается значение наибольшего // элемента массива А

maxval \*— А[0] for г +— 1 to п — 1 do if А [г] > maxval maxval <— A[i] return maxval

Очевидно, что в этом алгоритме размер входных данных нужно оценивать по количеству элементов в массиве, т.е. числом п. Операции, выполняемые чаще все­го, находятся во внутреннем цикле for алгоритма. Таких операций две: сравнение (А [г] > maxval) и присваивание (maxval <— А [г]). Какую из них считать базо­вой? Поскольку сравнение выполняется на каждом шаге цикла, а присваивание — нет, логично считать, что основной операцией алгоритма является операция при­сваивания. (Обратите внимание, что для любого массива размером п количество операций сравнения в рассматриваемом алгоритме постоянно. Поэтому не имеет смысла отдельно рассматривать эффективность алгоритма для худшего, типично­го и лучшего случаев.)

Обозначим через С (п) количество выполняемых в алгоритме операций срав­нения и попытаемся вывести формулу, выражающую их зависимость от размера входных данных п. Известно, что за один цикл в алгоритме выполняется одна операции сравнения. Этот процесс повторяется для каждого значения перемен­ной цикла г, которое изменяется от 1 до п — 1 включительно. Поэтому для С (п) получаем следующую сумму:

***п—1***

image35

Значение этой суммы очень легко вычислить, поскольку в ней единица суммиру­ется сама с собой п — 1 раз. Поэтому

**п—1**

С (п) = ^ 1 = п — 1 £ © (п).

Ниже предлагается общий план анализа эффективности нерекурсивных алго­ритмов.

**Общий план анализа эффективности нерекурсивных алгоритмов**

1. Выберите параметр (или параметры), по которому будет оцениваться размер входных данных алгоритма.
2. Определите основную операцию алгоритма. (Как правило, она нахо­дится в наиболее глубоко вложенном внутреннем цикле алгоритма.)
3. Проверьте, зависит ли число выполняемых основных операций только от размера входных данных. Если оно зависит и от других факторов, рассмотрите при необходимости, как меняется эффективность алгорит­ма для наихудшего, среднего и наилучшего случаев.
4. Запишите сумму, выражающую количество выполняемых основных операций алгоритма[[21]](#footnote-21).
5. Используя стандартные формулы и правила суммирования, упростите полученную формулу для количества основных операций алгоритма. Если это невозможно, определите хотя бы их порядок роста.

Прежде чем рассмотреть следующие примеры, полезно обратиться к приложе­нию А, в котором приведен список формул суммирования и правила вычислении, которые часто используются в процессе анализа алгоритмов. Особенно часто мы будем использовать два основных правила суммирования:

[и и](#bookmark453)

Y2cai = cY2ai (Rl)

i=l i=l

и и и

***Y2{ai±bi) = Y2ai±Y2bi*** (R2)

***i=l i=l i=l***

и две формулы суммирования:

U

УУ 1 = и — I + 1, где I < и — целые числа, представляющие (S1)

i=l нижнюю и верхнюю границы суммы

[г — г = 1 + 2 Н + п = П ^ « ^n2 G © (n2) (S2)](#bookmark75)

г=0 **г-1**

Обратите внимание, что формула = п~ К0Т0Р°й мы воспользовались

в примере 1, является частным случаем формулы (S1) при 1 = 1ии = п — 1.

Пример 2. Рассмотрим задачу проверки единственности элементов. Другими сло­вами, нужно убедиться, что все элементы массива различны. Эту задачу можно решить с помощью приведенного ниже несложного алгоритма.

Алгоритм UniqueElements (А [0..п — 1])

// Входные данные: массив вещественных чисел А[0..п — 1]

// Выходные данные: возвращается значение “true”, если все // элементы массива А различны, и “false”

// в противном случае

**for** i **<— 0** **to** п — **2** **do**

**for** j **<—** i **+ 1** **to n — 1** **do**

**if** A[i\ = A[j} **return false return true**

В этом алгоритме, как и в предыдущем, размер входных данных вполне есте­ственно оценивать по количеству элементов в массиве, т.е. числом п. Поскольку в наиболее глубоко вложенном внутреннем цикле алгоритма выполнятся только одна операция сравнения двух элементов, ее и будем считать основной опера­цией этого алгоритма. Обратите внимание, что количество операций сравнении будет зависеть не только от общего числа п элементов в массиве, но и от того, есть ли в массиве одинаковые элементы, и если есть, то на каких позициях они расположены. Ограничимся рассмотрением наихудшего случая.

По определению наихуцший случай входных данных соответствует такому массиву элементов, при обработке которого количество операций сравнений Cworst (п) будет максимальным среди всей совокупности входных массивов раз­мером п. После анализа внутреннего цикла алгоритма приходим к выводу, что наихудший случай входных данных (т.е. когда цикл выполняется от начала до конца, а не завершается досрочно) может возникнуть при обработке массивов двух типов: а) в которых нет одинаковых элементов; б) в которых два одинаковых элемента находятся рядом и расположены в самом конце массива. В подобных случаях при каждом повторе внутреннего цикла в нашем алгоритме выполняется одна операция сравнения. При этом переменная цикла j последовательно прини­мает значения от г +1 до п — 1. Внутренний цикл каждый раз повторяется заново при каждом выполнении внешнего цикла. При этом переменная внешнего цикла i последовательно принимает значения от 0 до п — 2. Таким образом, получаем:

п—2 п—1 п—2 п—2

cworst h = 1 = 5Z Kn - - (\*++ ч = X) (n -1 - \*)=

i=0 **j—i**-f 1 г=0 i=0

71—2 7i—2 7i—2 / л\ / \

-Et-D-SX-nE!-^f

i=0 i=0 t=0

/ ,n2 (n —2)(n — 1) (n— l)n 1 о 2\

= (n \_ I)2 \_ 1 >A L = 2 « -n2 € 0 (n2) .

В приведенных выше выкладках сумму (п — 1 — г) можно вычислить

проще, если воспользоваться формулой (S2):

^2 (п - 1 - г) = (гг - 1) + (гг - 2) + • • • + 1 (=}

1=0

Обратите внимание, что этот результат можно было легко предсказать: в рас­сматриваемом алгоритме в самом худшем случае для массива, состоящего из п эле­ментов, нужно сравнить между собой все (гг — 1) гг/2 различных пар элементов. ■

Пример 3. Для двух заданных матриц А и В размером гг х п определите вре­менную эффективность алгоритма их умножения С = АВ, основанного на опре­делении этой операции. По определению С — это матрица размером п х гг, эле­менты которой являются скалярными произведениями соответствующих строки матрицы А ни столбца матрицы Б, как показано ниже.

image36

7

*в*

с

\

/-я строка у-й столбец Элемент С [/, у]

Здесь для каждой пары индексов 0 ^ г, j ^ п — 1 элемент

С [г, j] = ***А [г,*** 0] ***В*** [0, j] + ***А [г, k]B[k,j]-\*** + ***А [г,п*** - 1] ***В [п - l,j].***

АЛГОРИТМ Matrix Multiplication {А [0..п — 1,0 ..п — 1], В [0..п — 1,0..п — 1])

// Выполняется умножение двух квадратных матриц размером // пх п. Используется алгоритм, основанный на определении // этой операции

// Входные данные: две квадратные пх п матрицы А и В

// Выходные данные: матрица С — АВ for i <— 0 to n — 1 do for j \*— 0 to n — 1 do

CM«-o.o

for к <— 0 to n - 1 do

***C[i,j\^C[i,j] + A[i,k]\*B[k,j]*** return ***С***

В этом алгоритме размер входных данных соответствует размеру матрицы п. Во внутреннем цикле алгоритма выполняются две арифметические операции: умножение и сложение. В принципе, в качестве основной операции алгоритма можно выбрать как одну, так и другую операцию. Мы рассмотрим случай, когда в качестве основной выбрана операция умножения (см. раздел 2.1). Заметьте, что для рассматриваемого алгоритма не обязательно отдавать предпочтение одной из этих операций, поскольку на каждом шаге внутреннего цикла каждая из них вы­полняется только один раз. Поэтому, подсчитав, сколько раз выполняется одна из операций, мы автоматически подсчитываем и количество выполнения другой опе­рации. А теперь давайте составим сумму для общего числа операций умножения М (п), выполняемых в алгоритме. Поскольку это число зависит только от разме­ра исходных матриц, не требуется отдельно рассматривать наихудший, типичный и наилучший случаи.

Вполне очевидно, что на каждом шаге внутреннего цикла алгоритма выпол­няется только одна операция умножения. При этом значение переменной цикла к последовательно изменяется от нижней границы 0 до верхней границы п — 1. По­этому количество операций умножения, выполняемых для каждой пары значений переменных i и j, можно записать как 1- Соответственно, общее количествоопераций умножения М (п) выражается следующей тройной суммой:

71—1 71—1 71—1

**"М'ЕЕЕ1-**

г=0 **j=** 0 **к=**0

Для вычисления значения этой суммы воспользуемся формулой (S1) и прави­лом (R1). Принимая во внимание, что значение внутренней суммы Ylk=о 1 — п (почему?), получим:

71 — 1 71— 1 71— 1 71 — 1 П— 1 71 — 1

***M(n)*** = Y,lL,J21 = Y,lL,n = **1>2** = п3-

г=0 **j=**0 **к=**0 г=0 **j=**0 г=0

Рассматриваемый алгоритм достаточно прост. Поэтому мы могли бы получить

тот же самый результат и без всех этих манипуляций с суммами. Хотите знать, как? Для получения результирующей матрицы в алгоритме необходимо вычислить значения п2 ее элементов. Каждый элемент этой матрицы является скалярным произведением n-элементной строки первой матрицы на тг-элементный столбец второй матрицы. При этом выполняется п операций умножения. Таким образом, общее количество операций умножения в рассматриваемом алгоритме составляет п • п2 = п3. Именно эту последовательность логических рассуждений вы долж­ны были применить при выполнении упражнения 2.1.2 (по крайней мере я это подразумевал, когда ставил ее перед вами).

Время выполнения алгоритма на конкретном компьютере можно оценить с по­мощью следующего произведения:

***Т*** (***п***) « ***СтМ (п) = СгпП3,***

где Cm — время выполнения одной команды умножения на рассматриваемом ком­пьютере. Чтобы получить более точную оценку, необходимо также учесть время выполнения команд сложения:

***Т*** (п) « ***СтМ (п)*** + ***СаА (п) = СтП*3** + Сагг**3** = ***(Cm + Са)*** гг3,

где са — время выполнения одной команды сложения. Обратите внимание, что полученная оценка отличается от прежней только постоянным множителем, а не порядком роста. ■ непреодолимой преградой для исследователя. Однако вопреки такому строгому предупреждению наш план можно с успехом применять для анализа большого количества простых нерекурсивных алгоритмов. В этом вы убедитесь сами, про­читав последующие главы этой книги.

В качестве последнего примера рассмотрим алгоритм, в котором значение пе­ременной цикла изменяется несколько иначе, чем в приведенных выше примерах.

Пример 4. С помощью приведенного ниже алгоритма можно определить количе­ство разрядов, которое будет иметь положительное десятичное число в двоичном представлении.

Алгоритм Binary (п)

// Входные данные: целое положительное число п

И Выходные данные: количество разрядов в двоичном // представлении числа п

**count** <— **1 while п** **> 1 do**

***count count*** + **1 *n [п/*2**J **return *count***

Для начала заметим, что в этом алгоритме операция сравнения п > 1, которая выполняется чаще всего, не находится в его внутреннем цикле while. Однако она определяет, будет ли выполняться все тело цикла. Поскольку количество выполня­емых операций сравнения всегда на единицу больше, чем общее число повторов выполнения цикла, выбор основной операции алгоритма не имеет большого зна­чения.

Более важно в этом примере то, что в процессе вычислений переменная цикла принимает всего насколько значений в интервале от ее минимального до мак­симального значения. Поэтому для определения количества раз, которые будет выполняться цикл, мы должны использовать другой метод. Поскольку на каж­дом шаге цикла значение переменной п уменьшается приблизительно вдвое, то искомый результат должен примерно равняться log2 п. На самом деле точная фор­мула для определения количества выполняемых операций сравнения п > 1 равна [log2nJ + 1, что совпадает с приведенной выше формулой (2.1) для количества бит в двоичном представлении числа п. Такой же ответ мы могли бы получить, применив методы, основанные на анализе рекуррентных соотношений. Их мы обсудим в следующем разделе, поскольку они очень удобны для анализа рекур­сивных алгоритмов. ■

1. Вычислите значения приведенных ниже сумм.

а) 1 + 3 + 5 + 7 + • • • + 999

б) 2 + 4 + 8 +16 Ч + 1024

71+1 71+1 71—1

**В) El г)** ±г **Д) Е\*(\* + !)**

г=3 г=3 г=0

71 71 71

**е) Е 3^+1 ж) Е Е У**

**j**=1 i=ij=i

1. Определите порядки роста приведенных ниже сумм.

**а)ПЕ(г2 + 1)2 б)** Е lgi2

г=0 г=2

В) Е (\* + 1) 2i\_1 Г> Ё Е (\* + J)

г=1 г=0 j=0

Используйте обозначения 0 (д (п)) с простейшей функцией д (п).

1. Эмпирическая дисперсия выборки, состоящей из п элементов xi,..., хп вычисляется по формуле

**2** ЕГ=1(\*г-\*)2 - Е?=1\*г

5 = 1 —, где х = - - -—,

?г — 1 п

или

2 ELi А ~ (ELi xi)2/n

***8 - п-*** **1**

Найдите и сравните количество операций деления, умножения, сло­жения и вычитания (последние две операции обычно объединяются и считаются как одна), которые необходимо выполнить для вычисле­ния эмпирической дисперсии по каждой из приведенных выше формул.

1. Рассмотрим следующий алгоритм.

Алгоритм Mystery (п)

// Входные данные: целое положительное число п

***S \*— 0***

**for г** **<— 1 to п** **do**

S \*— S + г \* г return S

а) Что вычисляется с помощью этого алгоритма?

б) Назовите основную операцию алгоритма.

в) Сколько раз выполняется основная операция?

г) К какому классу эффективности относится этот алгоритм?

д) Усовершенствуйте этот алгоритм или предложите другой алгоритм и оцените его класс эффективности. Если вам не удастся этого сде­лать, попытайтесь доказать, что это невозможно.

1. Рассмотрим следующий алгоритм.

Алгоритм Secret (А [0..п - 1])

// Входные данные: массив п вещественных чисел А[0..п — 1]

minval <— А[0] maxval <— А[0]

**for г <— 1 to гг — 1 do** if ***АЩ < minval minval А*** [г]

if ***АЩ > maxval maxval*** <— А[\*] **return *maxval*** — ***minval***

Ответьте на вопросы а)-д) задачи 4.

1. Рассмотрим следующий алгоритм.

Алгоритм Enigma (А [0..п - 1,0..п - 1])

// Входные данные: матрица действительных чисел

//

А[0..п — 1,0..п — 1]

**for** г <— 0 **to** п **—** 2 **do**

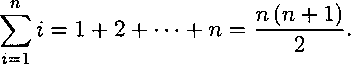
**for j** <— г + **1** **to n — 1 do**

**if *A[i,j] Ф A\j,i]***

**return false return true**

Ответьте на вопросы а)-д) задачи 4.

1. Улучшите реализацию алгоритма умножения матриц (см. пример 3) за счет уменьшения количества выполняемых операций сложения. Как это отразится на эффективности работы алгоритма?
2. К какому классу эффективности относится алгоритм, описанный в при­мере 4, как функция от количества битов в двоичном представлении числа п?
3. Докажите формулу



Для этого воспользуйтесь методом математической индукции либо по­старайтесь применить логику 10-летнего школьника Карла Фридриха Гаусса (Karl Friedrich Gauss) (1777-1855), который впоследствии стал одним из величайших математиков.

1. Рассмотрите один из вариантов важного алгоритма, который мы будем изучать в этой книге позже.

Алгоритм GE (А [0..п - 1,0..п])

// Входные данные: матрица вещественных чисел

// А[0..п — 1,0..п] размером п х п + 1

**for** г **<— 0 to п —** 2 **do for j** **<—** г **-I-1** **to n — 1 do for к** **<—** г **to n do A\j, к)** <- **A\j, к**] - А[г, **к)** \* **A\j, i)/A[i, i]**

а) Определите класс временной эффективности этого алгоритма.

б) Какое место в этом псевдокоде сразу бросается в глаза из-за сво­ей неэффективности? Как можно переписать этот алгоритм, чтобы повысить скорость работы алгоритма?

1. Математический анализ рекурсивных алгоритмов

В этом разделе мы увидим, как применить описанные в разделе 2.1 основы анализа эффективности алгоритмов для исследования рекурсивных алгоритмов. А начнем мы с рассмотрения примера, который часто используется в учебниках для иллюстрации идеи рекурсивных алгоритмов.

Пример 1. Вычислим значение функции факториала F (п) = п! для произволь­ного целого неотрицательного значения п. Так как n! = 1 • ... • (п — 1) • п = = (п — 1)! • п для всех n ^ 1 и по определению 0! = 1, мы можем вычислить F (n) = F (гг — 1) • гг при помощи следующего рекурсивного алгоритма.

Алгоритм F (п)

// Рекурсивное вычисление факториала // Входные данные: Целое неотрицательное число п

**II** **Выходные данные:** Значение п! **if п = 0 return 1 else**

return F(n — 1) \* n

Для простоты будем считать, что размер входных данных алгоритма будет указывать само число гг, а не количество битов в его двоичном представлении. Основной операцией этого алгоритма является умножение, количество выпол­нений которой мы обозначим через М (п).[[22]](#footnote-22) Так как для всех п > 0 значение функции F (гг) вычисляется по следующей формуле:

F (п) = F {п — 1) • гг,

М (гг) = М (гг — 1) +

image38

количество операций умножения М (гг), выполняемых при этом в алгоритме, должно удовлетворять следующему равенству для всех п > 0:

**1**

**Для умножения F(n—1) на п**

В самом деле, для всех п > 0, для вычисления значения функции F (п — 1) необходимо выполнить М (гг — 1) операций умножения. Еще одна операция умно­жения тратится на вычисление значения функции F (п), т.е. умножения получен­ного результата F (гг — 1) на гг.

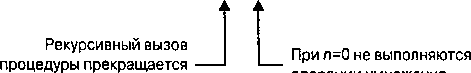
Последнее уравнение определяет последовательность М (гг), которую мы должны найти. Обратите внимание, что значение для М (п) не задано явно, т.е. в виде функции от числа гг. Оно выражено неявно в виде функции, значение которой зависит от значения этой же функции на предыдущем шаге алгоритма, т.е. при п — 1. Подобные равенства называют рекуррентными уравнениями, или рекуррентными соотношениями {recurrence relations), или, для краткости, про­сто рекуррентностями {recurrences). Рекуррентные соотношения играют важ­ную роль не только при анализе алгоритмов, но и в некоторых других областях прикладной математики. Обычно их подробно изучают на лекциях по дискрет­ной математике или дискретным структурам. В приложении Б приведен их очень краткий обзор. Сейчас же наша цель состоит в том, чтобы найти решение рекур­рентного соотношения М (гг) = М (гг — 1) + 1, т.е. определить явную формулу для последовательности М (гг), зависящую только от гг.

Следует обратить ваше внимание, что существует бесконечное множество по­следовательностей, удовлетворяющих решению приведенного выше рекуррентно­го соотношения. (Можете ли вы привести примеры хотя бы двух из них?) Чтобы найти однозначное решение, нужно задать начальные условия {initial condition), т.е. указать значение, с которого начинается последовательность. Чтобы опреде­лить это значение, нужно проанализировать условие, при котором в алгоритме прекращается рекурсивный вызов процедуры:

**if 77, = 0 return 1**

Эта строка кода позволяет сделать два вывода. Во-первых, так как рекурсив­ный вызов процедуры в нашем алгоритме прекращается при п = 0, минимальное значение п, для которого будет выполняться алгоритм и, следовательно, должна быть определена функция М (п), равно 0. Во-вторых, обратившись к строке кода алгоритма, в которой задано условие завершения работы, мы видим, что при п = 0 в алгоритме не выполняются операции умножения. Поэтому начальное условие, которое мы искали, будет выглядеть так:

М(0)= 0.



операции умножения

при л=0

Таким образом, рекуррентное соотношение и начальные условия для количе­ства операций умножения М (п) рассматриваемого нами алгоритма выглядят так:

(**2**.**2**)

М (п) = М (п — 1) + 1 для п > 0, М (0) = о.

Перед тем как приступить к решению этого рекуррентного соотношения, нам необходимо сделать небольшую паузу и напомнить вам одну важную вещь. Выше мы обсуждали две рекурсивные функции. Первая из них — это, собственно, сама функция вычисления факториала F (п), которая определена с помощью следую­щего рекуррентного уравнения:

F (п) = F (п — 1) • п для всех п > 0, F{ 0) = 1.

Вторая — это функция М (п), определяющая количество операций умноже­ния, выполняемых в рекурсивном алгоритме вычисления функции факториала F(n), псевдокод которого приведен в начале этого раздела. Как было показано выше, функция М (п) определяется с помощью рекуррентного соотношения (2.2). И именно его нам нужно решить.

Хотя в данном случае решение нетрудно “предсказать” (вспомните, какая из последовательностей начинается с 0 при п = 0 и увеличивается на 1 на каждом следующем шаге?), тем не менее будет крайне полезно получить его системати­ческим образом. Существует несколько методик решения рекуррентных отноше­ний. Мы воспользуемся одной из них, которая называется методом обратной подстановки (method of backward substitutions). Идея метода и, собственно, его название сразу станут понятны после того, как мы применим его для решениянашего рекуррентного уравнения:

М (n) = М (п — 1) + 1 = подставляем М (п — 1) = М (п — 2) 4- 1

= [М (п - 2) + 1] + 1 =

= М (п — 2) + 2 = подставляем М (п — 2) = М (п — 3) Н- 1

= [А/ (п — 3) + 1] + 2 =

= М (п - 3) + 3.

Взглянув на первые три строки, легко заметить закономерность, которая поз­волит предсказать не только вид следующей строки решения (кстати, какой она должна быть?), но также и вывести для него общую формулу:

М (п) = М (п — г) + г.

Строго говоря, правильность этой формулы должна быть доказана с помо­щью метода математической индукции. Однако гораздо легче сначала получить решение, как показано ниже, а затем проверить его правильность.

Осталось только применить начальные условия. Поскольку они заданы для п = 0, то, чтобы получить последнюю строку (или конечный результат) реше­ния методом обратных подстановок, нужно в приведенной выше общей формуле закономерности выполнить подстановку для г = п:

***М*** (n) ***= М(п — 1) + 1 = -- - = М(п — i) + i = -- - = M(n — п) + п = п.***

Вы не должны разочаровываться из-за того, что на получение этого столь “очевидного” и простого ответа было затрачено довольно много усилий. Преиму­щества описываемого метода, который был проиллюстрирован на этом простом примере, станут понятны чуть ниже, когда придется решать более сложные ре­куррентные уравнения. Обратите также внимание, что для получения произве­дения п последовательных целых чисел можно было воспользоваться простым циклическим алгоритмом, в котором выполнялось бы ровно столько же операций умножения. При этом в нем не тратилось бы время на дополнительный вызов про­цедур и не расходовалась бы дополнительная память для размещения параметров рекурсивной процедуры в стеке.

Следует отметить, что для задачи вычисления п! проблема временной эффек­тивности алгоритма не стоит так уж остро. Как было показано в разделе 2.1, факториал растет настолько быстро, что реально мы можем вычислить ее значе­ния только для очень малых п. Мы привели этот простой и понятный пример только для демонстрации стандартного подхода, используемого при анализе ре­курсивных алгоритмов. ■

Обобщая опыт, полученный при исследовании рекурсивного алгоритма вы­числения п!, можно составить следующий общий план исследования рекурсивных алгоритмов.

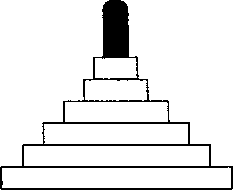
Общий план анализа эффективности рекурсивных алгоритмов

1. Выберите параметр (или параметры), по которому будет оцениваться размер входных данных алгоритма.
2. Определите основную операцию алгоритма.
3. Проверьте, зависит ли число выполняемых основных операций только от размера входных данных. Если оно зависит и от других факторов, рассмотрите при необходимости, как меняется эффективность алгорит­ма для наихудшего, среднего и наилучшего случаев.
4. Составьте рекуррентное уравнение, выражающее количество выпол­няемых основных операций алгоритма, и укажите соответствующие начальные условия.
5. Найдите решение рекуррентного уравнения или, если это невозможно, определите хотя бы его порядок роста.

Пример 2. В качестве следующего примера мы рассмотрим еще одну ставшую уже классической задачу, для решения которой используется рекурсивный алго­ритм. Речь идет о головоломке под названием Ханойские башни. Суть ее состоит в том, что у нас (ну или у некоторого мифического персонажа, если вам не хо­чется перемещать все эти диски) есть п дисков разного диаметра и три колышка. В исходном состоянии все диски надеты на первый колышек и упорядочены по диаметру, т.е. самый большой диск находится внизу стопки, а самый малый — вверху. Цель задачи — перенести все диски на третий колышек, используя при необходимости в качестве вспомогательного второй колышек. Нужно учесть, что за один раз можно перенести только один диск, и кроме того, нельзя помещать диск большего диаметра на диск меньшего диаметра.

Эта задача решается с помощью изящного рекурсивного алгоритма, как пока­зано на рис. 2.4. Чтобы перенести п > 1 дисков с первого колышка на третий (при этом второй колышек используется в качестве вспомогательного), сначала нужно рекурсивно перенести п — 1 дисков с первого колышка на второй (используя при этом третий колышек в качестве вспомогательного). После этого мы должны пере­нести наибольший диск непосредственно с первого колышка на третий и, наконец, рекурсивно перенести п — 1 дисков со второго колышка на третий (используя при этом первый колышек в качестве вспомогательного). Естественно, что при п = 1, мы можем непосредственно перенести единственный диск с первого колышка на третий.

Давайте применим приведенный выше общий план анализа эффективности рекурсивных алгоритмов к задаче о Ханойских башнях. Очевидно, что в данном случае размер входных данных нужно оценивать по количеству дисков п, а основ­ной операцией алгоритма будет перенос одного диска. Понятно, что количество переносов М (п) должно зависеть только от числа п. Поэтому для любого п > 1



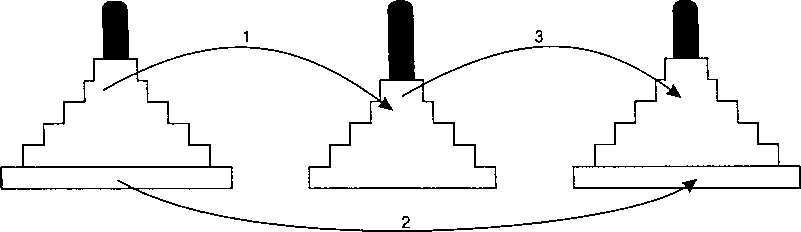


Рис. 2.4. Иллюстрация рекурсивного алгоритма решения головоломки о Ханойских башнях

будет верно следующее рекуррентное уравнение:

М (n) = М (п - 1) + 1 + М (п - 1).

Добавив к нему очевидное начальное условие М (1) = 1, получим следующее рекуррентное соотношение для количества переносов дисков М (п):

(2.3)

М (п) = 2М (п — 1) + 1 для п > 1, М( 1) = 1.

Для решения этого рекуррентного уравнения снова воспользуемся методом обратных подстановок:

М (п) = 2М (п — 1) + 1 =

= 2 [2М (п - 2) + 1] + 1 =

= 22М (п - 2) + 2 + 1 =

= 22 [2М (п - 3) + 1] + 2 + 1 = = 23М (п - 3) + 22 + 2 + 1 подставляем М (п — 1) = 2М (п — 2) + 1 подставляем М (п — 2) = 2М (п — 3) + 1

В первых трех строках этого решения прослеживается очевидная закономер­ность, так что четвертая строка будет выглядеть так:

24М (п- 4) + 23 + 22 + 2 + 1.

Чтобы получить обобщенную формулу решения, подставим вместо номера строки число г. В результате получим следующее:

М (n) = 2iM (гг - г) + 2i\_1 + 2\*~2 + • • • + 2 + 1 = 2{М (гг - г) + 2{ - 1.

Поскольку начальные условия заданы для п = 1, то, чтобы получить со­ответствующую этому числу строку решения рекуррентного отношения, нужно подставить г = п — 1. В результате получим следующее решение рекуррентного уравнения (2.3):

М (n) = 2n~lM (n - (п - 1)) + 2п-1 - 1 =

= 2n-1M (1) + 2п~1 - 1 = 2n\_1 + 2n\_1 - 1 = 2п - 1.

Мы выяснили, что рассматриваемый алгоритм относится к классу экспонен­циальных. А это означает, что он будет работать невообразимо длительное время даже для умеренных значений п (см. упражнение 2.4.5). Причем это отнюдь не означает, что мы выбрали для решения головоломки о ханойских башнях пло­хой алгоритм. На самом деле несложно доказать, что данный алгоритм является самым эффективным среди всех возможных алгоритмов решения нашей задачи. Все дело заключается в сложности, присущей самой задаче, что существенно за­трудняет ее решение численными методами. Тем не менее этот пример позволяет сделать один важный вывод:

Рекурсивные алгоритмы следует использовать очень осторожно, по­скольку часто за их внешней компактностью скрывается крайняя неэф­фективность.

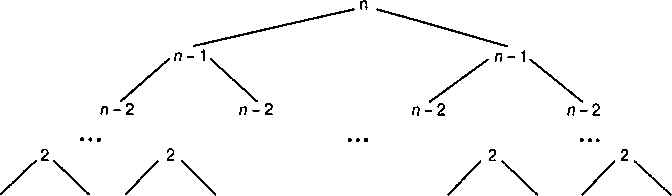
Если в рекурсивном алгоритме выполняется более одного вызова его самого, то для анализа такого алгоритма полезно построить дерево его рекурсивных вызо­вов. Узлы такого дерева будут соответствовать рекурсивным вызовам. Их можно обозначить в соответствии со значением параметра (или, в более общем случае, нескольких параметров), который передается рекурсивной функции в момент вы­зова. Для нашего примера с ханойскими башнями дерево рекурсивных вызовов будет выглядеть так, как показано на рис. 2.5. Сосчитав количество узлов в дереве, мы получим общее число вызовов, выполняемых в этом алгоритме:

п—1

с (n) = **2**\* = 2П - 1,

1=0

где I — номер уровня в дереве, показанном на рис. 2.5.



**1111** **1111**

Рис. 2.5. Дерево рекурсивных вызовов, выполняемых в алгоритме решения головоломки о ханойских башнях.

Как видим, это число соответствует количеству переносов дисков, которое было получено нами ранее. щ

Пример 3. В качестве следующего примера рассмотрим рекурсивный вариант алгоритма, описанного в конце раздела 2.3.

Алгоритм BinRec (п)

// Входные данные: целое положительное число п

II Выходные данные: количество разрядов в двоичном // представлении числа п

if п = 1

return 1 else

return BinRec([n/2J) + 1

Давайте построим рекуррентное уравнение и зададим начальные условия для количества операций сложения А (п), выполняемых в этом алгоритме. Количество операций сложения, выполняющихся при вычислении функции BinRec ([п/2J), равно А (|\_тг/2J). К этому нужно прибавить еще одну операцию сложения, которая выполняется в алгоритме для увеличения на единицу возвращаемого функцией BinRec значения. Поэтому для п > 1 можно записать следующее рекуррентное уравнение:

А(п) = A(|n/2j) + 1. (2.4)

Поскольку процесс рекурсивных вызовов завершается при n = 1 и при этом не выполняется никаких операций сложения, начальное условие для этого алгоритма будет выглядеть так:

А(1) = 0.

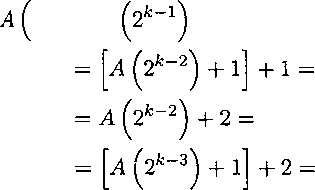
Наличие в параметре функции BinRec выражения \п/2\ приводит к тому, что в методе обратных подстановок сложно использовать значения параметра п, кото­рые не являются степенью 2. Поэтому стандартным подходом при решении такогорекуррентного отношения является поиск решения только для п = 2к с последу­

ющим применением теоремы, называемой правилом сглаживания {smoothness rule), которая описана в приложении Б. В ней утверждается, что сделанную оцен­ку порядка роста функции для п = 2к можно с очень высокой степень прибли­женности считать правильной для всех значений п. (В качестве альтернативы, найдя решение для п = 2к, иногда можно немного видоизменить его так, что­бы получить корректную формулу для произвольного значения п.) Итак, давайте применим описанное выше правило к нашему рекуррентному уравнению, которое при п = 2к примет следующий вид:

image43

А (2°) = 0.

Теперь можно без проблем применить метод обратной подстановки:



= A (2fc~3) + 3 =

2fc) = A + 1 = подставляем A (2fe\_1) = А (2к~2) + 1

подставляем А (2к = А (2к 3 j + 1

image45

= ***А (2к~к^ + к.***

В результате получаем, что

а(**2**fc) ***=А(1) + к = к.***

Поскольку п = 2к, получаем, что к = log2 п. Поэтому

A (n) = log2 n£0 (log п).

На самом деле можно доказать (см. упражнение 2.4.6), что точное решение для произвольного значения п получается, если переписать формулу как А (п) = = [log2nJ. ■

В этом разделе мы рассмотрели только азы анализа рекурсивных алгоритмов. Приведенные в нем методики будут использоваться в последующих главах этой книги, и по мере необходимости мы их будем расширять. В частности, в сле­дующем разделе мы рассмотрим числа Фибоначчи. При анализе алгоритма их вычисления мы применим более сложные рекуррентные отношения, для решения которых воспользуемся методом, отличающимся от метода обратных подстановок.

Упражнения 2.4

1. Найдите решение для следующих рекуррентных отношений:

а) х (п) = х (п — 1) + 5, для п > 1, х (1) = О

б) х (п) = Зх (п — 1), для п > 1, х (1) = 4

**в) х** **(п) = х (п** — **1) + п,** для **п** > **0,** х **(0) = 0**

г)

х (п) = х (п/2) + п, для п > 1, х (1) = 1 (найдите решение для п = 2к)

д) х(п) = х(п/3) + 1, для п > 1, х (1) = 1 (найдите решение для п = 3\*)

1. Постройте рекуррентное уравнение для количества вызовов функции F (п) в рекурсивном алгоритме вычисления п! и найдите его решение.
2. Рассмотрим приведенный ниже рекурсивный алгоритм вычисления суммы кубов первых п целых чисел S (n) = I3 + 23 + п3.

Алгоритм S (п)

И Входные данные: целое положительное число п

II Выходные данные: сумма кубов первых п целых чисел if п = 1 return 1 else

return S(n — l)+n\*n\*n

а) Постройте рекуррентное уравнение для количества выполнений ос­новной операции алгоритма и найдите его решение.

б) Сравните этот алгоритм с простым нерекурсивным алгоритмом.

1. Рассмотрим следующий рекурсивный алгоритм.

Алгоритм Q (п)

II Входные данные: целое положительное число п

if n = 1 return 1

**else**

**return** Q(n — **1)-{-2\*п — 1**

а) Постройте рекуррентное уравнение для значения этой функции, найдите его решение и определите, что вычисляется с помощью этого алгоритма.

б) Постройте рекуррентное уравнение для количества умножений, вы­полняемых в алгоритме, и найдите его решение.

image46

в) Постройте рекуррентное уравнение для количества сложений/вычи­таний, выполняемых в алгоритме, и найдите его решение.

1. а) В оригинальном варианте головоломки о ханойских башнях, кото­

рый был опубликован в 1890 году французским математиком Эду­ардом Лукасом (Edouard Lucas), после перемещения 64 дисков из мифической Башни Брахмы должен был наступить конец света. Оце­ните, сколько лет потребуется служителю культа для перемещения всех дисков, считая, что он выполняет эту работу со скоростью один диск в минуту. (Для простоты предположим, что служитель не будет ни есть, ни спать, т.е. не будет отвлекаться от своей работы и жить бесконечно долго.)

б) Сколько раз в этом алгоритме будет перенесен г-ый наибольший диск (1 ^ г ^ гг)?

в) Придумайте нерекурсивный алгоритм решения задачи о ханойских башнях.

1. а) Докажите, что точное количество операций сложения, выполняемых

в рекурсивном алгоритме BinRec(n) для произвольного положи­тельного целого числа гг, равно [log2 п\.

б) Постройте рекуррентное соотношение для количества сложений, выполняемых нерекурсивной версией этого алгоритма (см. раз­дел 2.3, пример 4), и решите его.

1. а) Разработайте рекурсивный алгоритм вычисления значения 2П для

произвольного неотрицательного целого числа п, который основан на формуле 2n = 2n\_1 + 271-1.

б) Постройте рекуррентное уравнение для количества операций сло­жения, выполняемых в алгоритме, и найдите его решение.

в) Изобразите дерево рекурсивных вызовов для этого алгоритма и под­считайте количество вызовов рекурсивной функции, выполняемых алгоритмом.

г) Можно ли сказать, что это хороший алгоритм для решения постав­ленной задачи?

1. Рассмотрим следующий рекурсивный алгоритм.

Алгоритм Mini (А [0..п - 1])

// Входные данные: массив А[0..п — 1] вещественных чисел

**if п** **= 1 return Д[0] else**

***temp*** <— ***Minl(A[0..n*** — 2]) **if *temp*** ^ ***A[n*** — **1] return *temp* else**

**return A[n** — **1]**

а) Что вычисляется с помощью этого алгоритма?

б) Постройте рекуррентное уравнение для количества выполнений ос­новной операции алгоритма и найдите его решение.

1. Рассмотрите еще один алгоритм решения задачи 8, в котором рекурсив­но выполняется разделение массива на две части. Первый раз функция

вызывается так: Mini (А [0..п — 1]).

Алгоритм Mini {А [/..г]) if I = г return A[i] else

***tempi*** <— ***Min****2****(A[l..[(l*** + r)/2\_|]) ***tempi*** <— ***Min****2****(A\[(l*** + r)/2j + l..r]) **if *tempi*** ^ ***tempi* return *tempi* else return *tempi***

а) Постройте рекуррентное уравнение для количества выполнения ос­новной операции алгоритма и найдите его решение.

б) Какой из алгоритмов — Mini или Mini — быстрее? Можете ли вы предложить алгоритм для решения этой же задачи, который был бы эффективнее двух рассмотренных выше алгоритмов?

1. Детерминант (определитель) матрицы размером п х п

^11 ••• Oin

^21 '‘' 02п

А =

Ол.1

обозначаемый как det А, может быть определен следующим образом. При п = 1 он равен ац. При п > 1 его можно вычислить по приведен­ной ниже рекурсивной формуле

П

det А = SjUij det Aj. j=i

В этой формуле элемент Sj равен +1, если j нечетное, и —1, если j четное; a\j — это элемент первой строки и j-ro столбца, a Aj — детер­минант матрицы размером (п — 1) х (п — 1), полученной из исходной матрицы А путем вычеркивания первой строки и j-ro столбца.

а) Постройте рекуррентное уравнение для количества операций умно­жения, выполняемых при вычислении детерминанта по описанной выше рекурсивной формуле.

б) Не решая это рекуррентное уравнение, что вы можете сказать о его порядке роста в сравнении с факториалом п! ?

1. Пример: числа Фибоначчи

В этом разделе мы рассмотрим числа Фибоначчи, т.е. знаменитую последо­вательность целых чисел:

0,1,1,2,3,5,8,13,21,34,... (2.5)

Ее можно определить с помощью простого рекуррентного соотношения

F (п) = F (п — 1) + F (п — 2) для п > 1 (2.6)

и двух начальных условий

[F (0) =0, F (1) = 1. (2.7)](#bookmark104)

Эта последовательность чисел впервые была получена в 1202 году итальян­ским математиком Леонардо Фибоначчи (Leonardo Fibonacci) при решении задачи о размножении кроликов. С тех времен в естественном мире было обнаружено множество примеров последовательностей чисел наподобие чисел Фибоначчи; они используются даже при решении таких задач, как предсказание биржевых цен на товары и цен на акции. Существуют и другие интересные применения чисел Фибоначчи, в том числе и в области информатики. Например, наихудший случай входных данных для алгоритма Евклида соответствует последовательным числам Фибоначчи (однако это уже выходит за рамки нашего изложения). В этом разделе мы сначала найдем явную формулу для определения n-го элемента по­следовательности чисел Фибоначчи F (п), а затем кратко обсудим алгоритм для его вычисления.

**Явная формула для определения гг-го элемента последовательности чисел Фибоначчи**

При попытке применить метод обратных подстановок для решения рекуррент­ного соотношения (2.6) мы столкнемся с тем, что выделить очевидную закономер­ность для получения его решения будет невозможно. Поэтому в данном случае мы применим результаты теоремы, в которой ищется решение для однородной линейной рекурсии второго порядка с постоянными коэффициентами:

ах (п) +Ъх(п — 1) + сх {п — 2) = 0. (2.8)

Здесь а, Ъ и с — постоянные вещественные числа (причем а ф 0), которые назы­ваются коэффициентами рекурсии, а х (гг) — это неизвестная последовательность, которую нужно найти. Согласно этой теореме (см. теорему 1 приложения Б), рекурсия (2.8) имеет бесконечное количество решений, которые можно получить с помощью одной из трех формул. Какую из трех формул следует применять в каж­дом конкретном случае, зависит от количества вещественных корней квадратного уравнения, коэффициенты которого совпадают с коэффициентами рекурсии (2.8):

аг2 + Ьг + с = 0. (2.9)

Вполне логично, что выражение (2.9) называется характеристическим урав­нением для рекурсии (2.8).

Применим указанную теорему к случаю вычисления последовательности чи­сел Фибоначчи. Для этого перепишем рекуррентное уравнение (2.6) следующим образом:

F (п) - F (п - 1) - F (п - 2) = 0. (2.10)

Его характеристическое уравнение имеет вид:

г2 — г — 1 = 0.

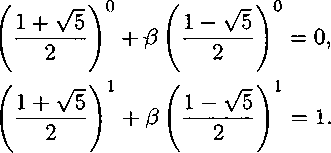
Корни этого уравнения вычисляются по следующей формуле:

1± ^1-4(-1) 1±у/5

Г1,2 - 2 2 ‘

Поскольку это характеристическое уравнение имеет два различных веществен­ных корня, для решения рекуррентного соотношения (2.10) мы должны исполь­зовать формулу, приведенную в первом случае теоремы 1:

Настало время учесть начальные условия (2.7). Воспользуемся ими, чтобы определить конкретные значения параметров а и /3. Для этого подставим числа О и 1 (т.е. значения параметра п, для которого приведены начальные условия) в последнюю формулу и приравняем полученный результат, соответственно, к О и 1 (т.е. к значениям F (0) и F (1) согласно формуле (2.7)):



***F*** (0) = а ***F (****1****) = а***

Проведя ряд несложных алгебраических преобразований, в конечном итоге мы получим приведенную ниже систему из двух линейных уравнений, содержащую неизвестные коэффициенты а и /3:

а + /3 = 0

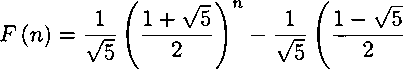
((1 + л/5) /2***)а + ((1-у/5)/2)0 =*** 1

Найдя решение этой системы уравнений (например, подставив во второе урав­нение (3 = —а и решив его для параметра а), получим следующие значения для искомых коэффициентов:

<\* = -4, /?=-4=.

у/5' ^ х/5'

Таким образом,



) =78 (\*\*-\*■) (211)

где

ф = **i+\_v/5 \_ Inigos и** ф = **1 ~** ^ = **-1 ^ —0.61803.[[23]](#footnote-23)**

2 2 ф

На первый взгляд трудно поверить в то, что формула (2.11), в которой в про­извольную целую степень возводятся два иррациональных числа, определяет n-ый элемент последовательности целых чисел Фибоначчи (2.5), но тем не менее это так!

Формула (2.11) имеет одно явное преимущество: при взгляде на нее сразу понятно, что функция F (п) имеет экспоненциальный порядок роста (помните задачу о размножении кроликов?), т.е. F (п) € 0 (фп)- Это следует из констатации того факта, что значение числа ф находится в интервале между — 1 и 0. Поэтому при п, стремящемся к бесконечности, значение члена фп становится бесконечно малым. На самом деле можно доказать, что влияние второго члена формулы (2.11) фп на значение F (п) можно учесть, если округлить значение первого члена этой формулы до ближайшего целого числа. Другим словами, для любого неот­рицательного целого числа п будет справедлива следующая формула:

F (п) = —:= фп, округленное до ближайшего целого числа. (2.12) v5

**Алгоритмы вычисления чисел Фибоначчи**

Несмотря на то что числа Фибоначчи имеют много замечательных свойств, в этом разделе мы ограничимся лишь несколькими замечаниями по поводу су­ществующих алгоритмов для их вычисления. В самом деле, последовательность этих чисел растет настолько быстро, что здесь мы должны рассматривать не эф­фективный по времени метод для их вычисления, а способы работы с большими числами. Кроме того, для простоты в приведенных ниже алгоритмах мы будем учитывать только такие операции, как сложение и умножение. Поскольку после­довательность чисел Фибоначчи очень быстро растет до огромных значений, она должна быть проанализирована более подробно, а не так, как это сделано ниже. Тем не менее, несмотря на сделанное предупреждение, рассмотренные алгорит­мы и методы их анализа являются хорошими примерами для тех, кто учится проектировать и анализировать алгоритмы.

А начнем мы с использования рекурсии (2.6) и ее начальных условий (2.7) для построения очевидного рекурсивного алгоритма вычисления функции F (п).

**Алгоритм F (п)**

II Вычисление n-того элемента последовательности чисел // Фибоначчи с использованием рекурсивного алгоритма,

// соответствующего определению // Входные данные: целое неотрицательное число п

II Выходные данные: n-ый элемент последовательности чисел // Фибоначчи

if п ^ 1 return п else

return F(n — 1) + F(n — 2)

Прежде чем погрузиться в формальный анализ этого алгоритма, можете ли вы сказать, является ли он эффективным? Впрочем, в любом случае мы должны провести его формальный анализ. Очевидно, что основной операцией этого ал­горитма является сложение. Поэтому давайте обозначим через А (п) количество операций сложения, которые выполняются в алгоритме при вычислении функ­ции F (п). Тогда количество операций сложения, выполняемых для вычисления значений функции F (п — 1) и F (п — 2), будет, соответственно, равно А (п — 1) и А (п — 2). Кроме того, в алгоритме нужно выполнить еще одну операцию сло­жения для суммирования этих значений. Таким образом, для А (п) мы получим следующую рекурсию:

А (п) = А (п — 1) + А (п — 2) + 1 для п > 1, (2.13)

Л(0) = 0, Л(1) = 0.

Рекурсия A (n) — А (п — 1) — А (п — 2) = 1 полностью совпадает с рекур­сией (2.10), но ее правая часть не равна нулю. Подобные рекуррентные отно­

шения называют неоднородными рекурсиями (inhomogeneous recurrences). Для их решения существует несколько общих методик (за подробностями обратитесь к приложению Б или к любому учебнику по дискретной математике), но для на­шей конкретной рекурсии можно воспользоваться специальным приемом и тут же получить ее решение. Для этого нужно привести неоднородную рекурсию к однородной, переписав выражение (2.13) следующим образом:

[А (п) + 1] - [А (п - 1) + 1] - [А (п 2) + 1] — 0.

Обозначив В (п) — А (п) -I-1, получим:

В (п) — В (п — 1) — В (п - 2) = 0,

В(0) = 1, В (1) = 1.

Точное решение этой однородной рекурсии можно найти по аналогии с реше­нием рекурсии (2.10), которое было приведено выше при поиске явной формулы для F (п). Но можно этого не делать, поскольку легко заметить, что фактически В (п) — это та же рекурсия, что и F (п), за исключением того, что она начинается с двух единиц, т.е. опережает F (п) на один шаг. Поэтому В (п) = F (п + 1) и

А (п) = В (п) — 1 = F (п + 1) — 1 = -^= (У\*1 - <Г+1) - 1

Отсюда следует, что А (п) G © (фп). Если же оценивать размер задачи количе­ством битов в двоичном представлении числа п, b = [log2 п\ -I- 1, то получится, что найденное решение будет относиться к еще худшему классу эффективности, поскольку представляет собой дважды экспоненту.

Низкую эффективность описанного выше алгоритма можно было предвидеть, взглянув на формулу рекурсии (2.13). В самом деле — в нем содержится два рекур­сивных вызова, которым передаются ненамного меньшие значения параметров, чем само число п. (Вы не сталкивались с подобным случаем ранее?) Кроме того, чтобы понять причину низкой эффективности алгоритма, достаточно взглянуть на его дерево рекурсивных вызовов, построенное по результатам выполнения ал­горитма. Пример подобного дерева, построенного для п = 5 приведен на рис. 2.6. Обратите внимание, что в этом алгоритме функция с одним и тем же значением аргумента вызывается по несколько раз. Поэтому очевидно, что такой алгоритм будет крайне неэффективен.

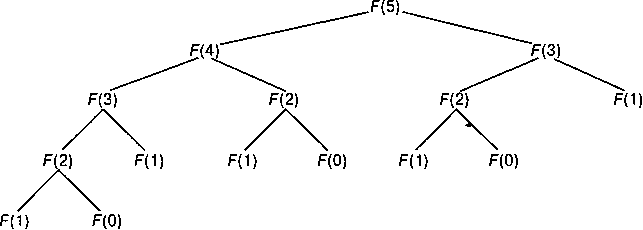


Рис. 2.6. Дерево рекурсивных вызовов, построенное для алгоритма вы­числения последовательности чисел Фибоначчи при п = 5

Получится намного более эффективный алгоритм, если числа Фибоначчи вы­числять последовательно одно за другим в цикле, как показано ниже.

**Алгоритм Fib (п)**

// Вычисление n-го элемента последовательности чисел // Фибоначчи при помощи итеративного алгоритма // с использованием определения чисел Фибоначчи // Входные данные: целое неотрицательное число п

// Выходные данные: n-ый элемент последовательности чисел // Фибоначчи F[0] <- 0 F[l] - 1 for г <— 2 to п do F[i] «- F[i - 1] + F[i - 2] return F[n]

Очевидно, что в этом алгоритме выполняется п—1 операций сложения. Следо­вательно, алгоритм является линейным, если рассматривать зависимость времени его работы от п, и “всего лишь” экспоненциальным, если рассматривать функцию

от количества битов b в двоичном представлении числа п. Заметьте, что можно из­бежать использования дополнительного массива для хранения всех предыдущих элементов последовательности Фибоначчи: для выполнения задачи достаточно сохранять только два значения (см. упражнение 2.5.6).

Третий способ вычисления n-го элемента последовательности чисел Фибо­наччи заключается в использовании формулы (2.12). Очевидно, что эффектив­ность этого алгоритма будет определяться эффективностью экспоненциального алгоритма, используемого для вычисления фп. Если это делается путем простого перемножения ф на себя п — 1 раз, то алгоритм будет принадлежать множеству 0 (п) = 0 (26). Для решения подобных задач имеются и более быстрые алгорит­мы, не относящиеся к экспоненциальному классу Например, в главах 5 и 6 мы рассмотрим алгоритм решения данной задачи со временем работы 0 (log п) = = 0(b). Заметим, что при реализации последнего алгоритма вычисления п-го элемента последовательности чисел Фибоначчи следует быть особенно внима­тельным и осторожным. Поскольку все промежуточные результаты вычислений являются иррациональными числами, мы должны убедиться в том, что их прибли­зительные значения представлены в компьютере с достаточной точностью, чтобы это не повлияло на правильность конечного результата.

Наконец, для вычисления n-го элемента последовательности чисел Фибоначчи существует еще один алгоритм со временем работы 0 (logn), в котором исполь­зуются только целые числа. Он основан на равенстве

|  |  |  |
| --- | --- | --- |
| F(n — 1) F(n) |  | 0 1 |
| F(n) F(n +1) |  | 1 1 |

и использует эффективные способы возведения матриц в степень. Упражнения 2.5

1. Поищите Web-сервер, посвященный применению чисел Фибоначчи, и ознакомьтесь с его содержимым.
2. Проверьте методом непосредственной подстановки, что функция ^g(фп — фп) на самом деле удовлетворяет рекуррентному отношению (2.6) для любого п > 1 и начальным условиям (2.7) для п = 0 и 1.
3. Известно, что максимальное значение переменных простых типов язы­ка Java int и long составляет, соответственно, 231 — 1 и 263 — 1. Найдите наименьшее п, для которого значение n-го элемента после­довательности чисел Фибоначчи уже не будет помещаться в ячейку памяти, выделенную под переменную указанного ниже типа:
4. int б) long
5. Определите количество различных способов подъема по лестнице, со­стоящей из п ступенек, при условии, что на каждом шаге можно поды­маться либо на одну, либо сразу на две ступени. Например, по лестнице, состоящей из трех ступенек, можно подняться тремя способами: 1-1-1, 1-2 и 2-1 [118].

image50

1. Рассмотрим рекурсивный алгоритм вычисления n-го элемента после­довательности чисел Фибоначчи F (п), основанный на их определении. Пусть С (п) и Z (п) — количество вычислений значений F (1) и F (0), соответственно. Докажите, что

а) С (n) = F (п) б) Z(n) = F(n- 1)

1. Усовершенствуйте алгоритм Fib (п), чтобы он требовал при работе только 0(1) памяти.
2. Докажите, что при п ^ 1

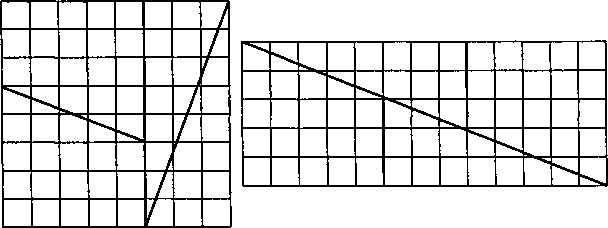
|  |  |  |
| --- | --- | --- |
| F(n — 1) F(n) |  | 0 l" |
| F(n) F(n + 1) |  | 1 1 |

1. Сколько делений по модулю будет выполнено алгоритмом Евклида для входных данных, представляющих собой два последовательных числа Фибоначчи — F (п) и F (п — 1)?
2. а) Докажите тождество Кассини: F (п + 1) F (п — 1)—[F (п)]2 = (—1)п

image51

для всех п ^ 1.

б) Рассмотрим следующий парадокс, основанный на тождестве Кас­сини. Возьмем шахматную доску размером 8x8 (или, в общем случае, доску размером F (п) х F (п), разделенную на [F (п)]2 кле­ток). Разделим ее на две трапеции и два треугольника, как показано в левой части рисунка. Затем соберем эти трапеции и треугольники в новую фигуру, показанную в правой части рисунка. Площадь ле­вого квадрата равна 8 х 8 = 64 клетки, а правого прямоугольника — 13 х 5 = 65 клеток. Поясните этот парадокс.



1. Реализуйте на языке программирования по собственному усмотрению три алгоритма вычисления пяти последних цифр п-го числа Фибонач­чи, основанных на а) рекурсивном алгоритме F (п); б) итеративном алгоритме Fib(n) и в) версии Fib(n) с экономным использованием памяти. Проведите эксперимент по поиску максимального значения п, для которого разработанная вами программа работает на вашем ком­пьютере меньше 1 минуты.
2. Эмпирический анализ алгоритмов

В разделах 2.3 и 2.4 мы видели, как можно выполнить математический анализ рекурсивных и нерекурсивных алгоритмов. Несмотря на успешное применение описанных методов ко многим простым алгоритмам и множество усовершенство­ванных технологий математического анализа алгоритмов ([45, 46, 93] и [105]), считать возможности математики безграничными нельзя. Имеется масса алгорит­мов, причем достаточно простых, которые с трудом поддаются математическому анализу. Как отмечалось в разделе 2.2, это в особенности относится к анализу среднего случая.

Принципиальной альтернативой математическому анализу эффективности ал­горитмов является их эмпирический анализ. Вот примерный план проведения этого вида анализа.

**Общий план эмпирического анализа эффективности алгоритма**

1. Уяснение цели предстоящего эксперимента.
2. Определение измеряемой метрики М и единиц измерения (количество операций или время работы).
3. Определение характеристик входных данных (их диапазон, размер и Т.Д.).
4. Создание программы, реализующей алгоритм (или алгоритмы), для проведения эксперимента.
5. Генерация образца входных данных.
6. Выполнение алгоритма (или алгоритмов) над образцом входных дан­ных и запись наблюдаемых данных.
7. Анализ полученных данных.

Рассмотрим по очереди указанные шаги. Имеется ряд целей, которые мо­гут быть поставлены перед эмпирическим анализом алгоритмов. Они включают проверку точности теоретических выводов об эффективности алгоритма, сравне­ние эффективности нескольких алгоритмов, предназначенных для решения одной и той же проблемы или различных реализаций одного и того же алгоритма, выдви­жение гипотезы о классе эффективности алгоритма, выяснение эффективности программы, реализующей алгоритм, на данной конкретной машине. Очевидно, что разработка эксперимента должна зависеть от того, на какой именно вопрос он должен ответить.

В частности, цель эксперимента должна влиять, если не определять, на то, каким образом будет выполняться измерение эффективности алгоритма. Первый вариант заключается во вставке в программу, реализующую алгоритм, счетчика (или счетчиков), которые будут подсчитывать количество выполнений алгорит­мом базовых операций. Обычно это достаточно просто, и вы должны только не забывать, что базовые операции могут выполняться не в одном месте програм­мы, и учитывать все возможные их выполнения. Само собой, вы всегда должны проверять модифицированную таким образом программу, чтобы убедиться в ее корректности — как в смысле решения поставленной перед алгоритмом задачи, так и в смысле корректности работы внесенных в программу счетчиков.

Второй вариант заключается в определении времени работы программы, ре­ализующей исследуемый алгоритм. Простейший путь выяснения времени рабо­ты — использование системной команды, такой как time в UNIX. Можно также определять время работы фрагмента кода, запрашивая системное время непо­средственно перед началом выполнения фрагмента (tstart) и сразу после его завершения (t/mis/i)> а затем просто вычислять разность полученных значений (tfinish — tstart)-1 В С и C++ для этой цели можно использовать функцию clock, а в Java — метод currentTimeMillis () из класса System.

Однако очень важно не забывать о некоторых вещах. Во-первых, системное время обычно не очень точное, и вы можете получить несколько отличающиеся друг от друга результаты при повторных запусках одной и той же программы с од­ними и теми же входными данными. Очевидным средством противодействия это­му эффекту является запуск программы и выполнение измерений несколько раз, с последующим усреднением полученных результатов. Во-вторых, высокая ско­рость современных компьютеров может привести к тому, что время работы будет невозможно зарегистрировать (будут получаться нулевые значения). Обойти эту неприятность легко, запуская программу в цикле много раз, а затем поделив заре­гистрированное время выполнения на количество итераций цикла. В-третьих, на компьютере, работающем под управлением многозадачной операционной систе­мы (как, например, UNIX), регистрируемое время может включать время, затра­ченное процессором на работу над другими программами, что, понятно, мешает проведению эксперимента. Таким образом, вы должны запросить у операционной системы время, затраченное конкретно на выполнение вашей программы (в UNIX

7Если системное время возвращается в “тиках” **(ticks),** то надо не забыть разделить полученное число на константу, определяющую количество “тиков” в единице времени.

это время называется “пользовательским временем” (user time) и автоматически предоставляется командой time).

Таким образом, измерение физического времени работы имеет ряд недостатков как принципиального характера (наиболее важным их них является зависимость от конкретной машины, на которой проводится эксперимент), так и технических, которых нет у метода подсчета базовых операций. С другой стороны, измерение физического времени дает очень конкретную информацию о производительно­сти алгоритма в данной вычислительной среде, что для экспериментатора может оказаться более важным, чем, скажем, класс асимптотической эффективности ал­горитма. Кроме того, измерения времени, затраченного на различные части про­граммы, могут помочь выявить узкие места в производительности программы, что не позволит сделать абстрактное рассмотрение базовых операций алгоритма. По­лучение таких данных, которое называется профилированием, — важный ресурс для эмпирического анализа времени работы алгоритмов; обычно в большинстве сред имеются специальные системные инструменты для получения данной ин­формации.

Независимо от того, решите ли вы измерять производительность алгоритма при помощи подсчета базовых операций или фиксируя время выполнения, перед вами встанет вопрос о выборе образца исходных данных для проведения экспери­мента. Зачастую требуется использовать образец входных данных, представляю­щий собой “типичные” данные. Для некоторых классов алгоритмов — например, алгоритмов для решения задачи коммивояжера, которая будет рассмотрена поз­же в этой книге, — исследователи разработали набор экземпляров задач, которые используются в качестве тестовых образцов. Однако на практике гораздо чаще приходится сталкиваться с ситуациями, когда входные данные должен выбирать и готовить сам экспериментатор. Обычно приходится самостоятельно решать, ка­ков должен быть размер входных данных (разумно начать с данных относительно небольшого размера и при необходимости постепенно увеличивать их), диапа­зон величин входных данных (чтобы он не был ни слишком малым, ни слишком большим), и разрабатывать процедуру для генерации входных данных в выбран­ном диапазоне. Обычно данные либо следуют некоторому шаблону (например, 1000, 2000, 3000, ..., 10000 или 500, 1000, 2000, 4000, ..., 128 000), либо гене­рируются случайным образом (например, с равномерным распределением между наименьшим и наибольшим значениями).

Главное достоинство следования определенному шаблону в том, что в этом случае легче проанализировать влияние данных на работу и эффективность алго­ритма. Например, если значения входных данных генерируются путем удвоения, можно вычислить отношение М(2п)/М(п) наблюдаемой метрики М (количе­ство операций или время) и увидеть, является ли поведение алгоритма типичным для одного из основных классов эффективности (см. раздел 2.2). Основной недо­статок неслучайных величин — возможность того, что для конкретного набора данных алгоритм продемонстрирует нетипичное поведение. Например, если все значения входных данных четны, а исследуемый алгоритм для нечетных дан­ных работает существенно медленнее, результат эксперимента окажется далек от истины.

Еще один важный вопрос, связанный со входными данными: следует ли ис­пользовать несколько экземпляров данных одного и того же размера. Если вы ожидаете, что наблюдаемая метрика может существенно изменяться даже для данных одного и того же размера, вероятно, будет разумно включить во входные данные несколько разных экземпляров данных. (В математической статистике имеются хорошо разработанные методы, которые могут помочь в выработке вер­ного решения в данной ситуации — стоит лишь немного покопаться в литературе на эту тему.) Само собой, при использовании нескольких экземпляров данных наблюдаемые значения для каждого размера данных должны быть усреднены.

Весьма часто эмпирический анализ эффективности требует генерации случай­ных чисел. Даже используя для входных данных шаблон, мы обычно хотим, чтобы сами экземпляры данных генерировались случайным образом. Генерация случай­ных чисел на цифровом компьютере, как известно, представляет собой сложную задачу, которая в принципе может быть решена только приближенно. В этом за­ключается причина того, что кибернетики предпочитают именовать такие числа псевдослучайными. С практической точки зрения простейший и наиболее есте­ственный способ получения таких чисел состоит в использовании генератора случайных чисел из библиотеки используемого вами языка программирования. Обычно такой генератор дает на выходе (псевдо)случайные значения, равномерно распределенные в интервале от 0 до 1. Если требуются некоторые другие (псев­дослучайные значения, можно применить соответствующие преобразования. На­пример, если х — непрерывная случайная величина, равномерно распределенная в диапазоне 0 ^ х < 1, то величина у = I + \\_х (г — 1)\ будет представлять со­бой целую величину, равномерно распределенную в диапазоне между целыми числами I и г — 1 (Z < г).

Вы можете не пользоваться библиотечным генератором, а реализовать один из множества известных алгоритмов для генерации (псевдо)случайных чисел. Наи­более широко используемым и детально изученным среди таких алгоритмов явля­ется так называемый линейный конгруэнтный метод {linear congmential method).

**АЛГОРИТМ Random** **(n,** га, **seed**, **a, b)**

II Генерирует последовательность из n псевдослучайных чисел // с использованием линейного конгруэнтного метода // Входные данные: Натуральное число п и натуральные

// параметры га, seed, a, b

II Выходные данные: последовательность ... ,гп

II псевдослучайных чисел, равномерно

// распределенных между 0 и т — 1

// Примечание: Псевдослучайные числа между 0 и 1 могут

// быть получены путем рассмотрения

// генерируемых алгоритмом целых чисел как

// цифр после десятичной точки

го <— seed

**for i** 1 **to n** **do**

***Ti <— (a\**** Ti\_i ***+ b***) mod ***m***

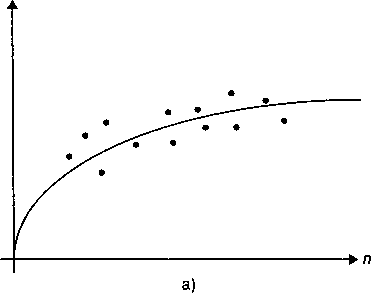
Простота приведенного алгоритма обманчива, поскольку все сложности скры­ты в выборе параметров алгоритма. Вот неполный список рекомендаций, основан­ных на результатах сложного математического анализа (см. [66]): значение seed можно выбрать произвольным образом (зачастую для этого используются текущая дата и время); т должно быть большим (может оказаться удобным выбор для т величины 2W, где w — размер слова компьютера); а должно быть целым числом от 0.01т до 0.99т- без определенной закономерности в его цифрах, но такое, что a mod 8 = 5; значение Ъ может быть выбрано равным 1.

Эмпирические данные, полученные в ходе эксперимента, должны быть запи­саны, а затем представлены для дальнейшего анализа. Данные могут быть пред­ставлены в таблице или графически, точками в декартовой системе координат. Неплохая мысль использовать одновременно оба способа, поскольку для каждого из них характерны свои сильные и слабые стороны.

Основное преимущество табулированных данных заключается в легкости до­ступа и работы с ними. Например, мы можем вычислить отношения М (п)/д (гг), где д (п) — кандидат в представители класса эффективности исследуемого алго­ритма. Если алгоритм действительно принадлежит 0(д(гг)), то, вероятнее все­го, эти отношения будут сходиться к некоторой положительной константе при возрастании п. (Заметим, что некоторые невнимательные экспериментаторы-но­вички полагают, что эта константа должна быть равна единице, что, конечно же, неверно в силу определения © (д (п))). Мы можем также вычислить отношения М (2п) jM (п) и посмотреть, как ведет себя время работы алгоритма при удвоении размера входных данных. Как было показано в разделе 2.2, в случае логарифмиче­ского алгоритма это отношение должно изменяться очень слабо, а для линейного, квадратичного и кубического алгоритмов — сходиться к 2, 4 и 8, соответственно.

С другой стороны, графическое представление данных также может помочь в выдвижении гипотезы о вероятном классе эффективности алгоритма. В слу­чае логарифмического алгоритма график имеет выпуклый вверх вид (рис. 2.1а). Этот вид графика отличает логарифмические алгоритмы от всех прочих алгорит­мов. В случае линейного алгоритма экспериментальные точки имеют тенденцию выстраиваться вдоль обычной прямой линии или, вообще говоря, располагать­ся между двумя прямыми линиями (рис. 2.16). Графики функций из 0 (n lg п)

и 0 (п2) имеют выпуклый вниз вид (рис. 2.1в), что делает их идентификацию бо­лее сложной. График для кубического алгоритма также имеет выпуклый вниз вид, но растет с существенно более высокой скоростью. В случае экспоненциально­го алгоритма вертикальная ось требует использования логарифмической шкалы, т.е. на график следует наносить точки loga М (п) вместо М (п) (наиболее часто применяемые основания логарифмов — 2 и 10). В такой системе координат гра­фик истинной экспоненциальной функции представляет собой прямую линию, поскольку из М (п) « сап следует logb М (n) logb с + п logb а.



Время работы алгоритма Время работы алгоритма

Время работы алгоритма

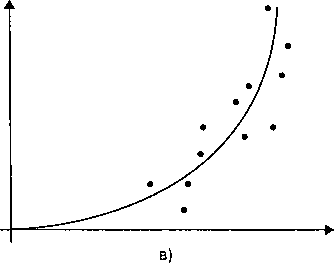
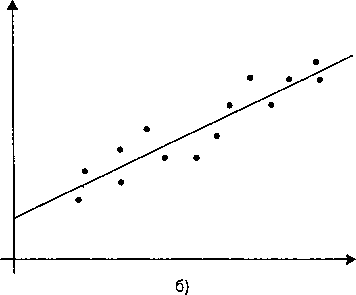


Рис. 2.7. Типичные графики функций: а) логарифмической, б) линейной, в) выпуклой вниз функции



Одно из возможных применений эмпирического анализа — попытка предска­зать производительность алгоритма для экземпляра исходных данных, не вклю­ченного во множество экземпляров исходных данных эксперимента. Например, если мы заметим, что отношение М (п)/д (п) близко к некоторой константе с для экземпляров, использованных в эксперименте, то мы можем аппроксимировать М (п) произведением сд (п) и для других значений п. Хотя этот подход вполнеразумен, его следует использовать с осторожностью, в особенности для значений п, которые находятся вне экспериментально исследованного диапазона. (Матема­тики называют такое предсказание экстраполяцией (<extrapolation) в отличие от интерполяции (interpolation), которая работает со значениями в пределах иссле­дуемого диапазона.) В частности, нельзя ничего сказать о точности таких оценок. Конечно, можно попытаться применить стандартные методы статистического ана­лиза данных, однако заметим, что они в основном базируются на определенных вероятностных предположениях, которые могут оказаться неверны для рассмат­риваемых экспериментальных данных.

Представляется уместным завершить данный подраздел перечислением основ­ных отличий между математическим и эмпирическим анализом алгоритмов. Глав­ное преимущество математического анализа — его независимость от конкретных входных данных, а недостаток — ограниченная применимость, в особенности для исследования эффективности в среднем случае. Эмпирический анализ, напротив, применим к любому алгоритму, но его результаты могут зависеть от конкретных входных данных и использованного для проведения эксперимента компьютера.

Упражнения 2.6

1. Рассмотрим хорошо известный алгоритм сортировки (чуть позже мы более детально познакомимся с ним), в который вставлен счетчик ко­личества сравнений ключей.

Алгоритм Sort Analysis (А [0..п - 1])

// Входные данные: Массив А[0..п — 1] из гг упорядочиваемых

// элементов

// Выходные данные: Общее количество выполненных сравнений // ключей

count <— О

for i <— 1 to тт — 1 do v +— А[г] j <- г - 1

**while *j*** ^ **0 and *A[j\ > v*** **do *count*** <— ***count*** + 1 ***A\j*** + 1] «- ***A[j]*** *j***«-** *3 -* **1**

**A\j** + 1] <- **V**

**return count**

Правильно ли размещен счетчик? Если да, докажите это; если нет, внесите необходимые изменения в код.

1. а) Выполните программу из упражнения 1 с корректно вставленным

счетчиком (или счетчиками) и определите количество сравнений для 20 случайных массивов размером 1000, 1500, 2000, 2500, ..., 9000, 9500.

б) Проанализируйте полученные данные и выдвиньте гипотезу об эф­фективности рассматриваемого алгоритма в среднем случае.

в) Оцените количество сравнений ключей, которые следует ожидать при работе рассматриваемого алгоритма со случайным образом сге­нерированным массивом из 10 000 элементов.

1. Выполните упражнение 2, но на этот раз измеряйте не количество сравнений, а время работы программы в миллисекундах.
2. Выдвиньте гипотезу о классе эффективности алгоритма на основании следующих эмпирических подсчетов количества базовых операций:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Размер | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 | 8000 | 9000 | 10000 |
| Количество | 11966 | 24303 | 39992 | 53010 | 67272 | 78692 | 91274 | 113063 | 129799 | 140538 |

1. Какое масштабирующее преобразование сделает логарифмический гра­фик линейным?
2. Как можно отличить график алгоритма из 0 (lg lg п) от графика алго­ритма 0 (lgn)?
3. а) Найдите эмпирически наибольшее количество делений, которые ал­

горитм Евклида выполняет при вычислении gcd (га, п) при 1 ^ п ^ т ^ 100.

б) Для каждого натурального к эмпирически найдите наименьшую па­ру целых чисел 1 ^ п ^ га ^ 100, для которых алгоритм Евклида требует выполнения к делений при поиске gcd (га, п).

1. Эффективность алгоритма Евклида в среднем случае для входных дан­ных размером п можно измерить как количество делений Davg (п), выполняемых алгоритмом при вычислении gcd (гг, 1), gcd (гг, 2), ..., gcd (гг, гг). Например,

Davg (5) — -(1+ 2 + 3 + 2 +1) = 1.8. о

Постройте график функции DaVg (п) и укажите класс эффективности алгоритма в среднем случае.

1. Выполните эксперимент для выяснения класса эффективности решета Эратосфена (см. раздел 1.1).
2. Выполните эксперименты по исследованию времени работы трех ал­горитмов вычисления gcd (га, п), представленных в разделе 1.1.

**Визуализация алгоритмов**

Помимо математического и эмпирического анализа имеется еще один путь изучения алгоритмов. Он называется визуализацией алгоритма и может быть определен как использование изображений для передачи некоторой полезной ин­формации об алгоритмах. Эта информация может быть визуальной иллюстраци­ей действий, выполняемых алгоритмом, или его производительности для разных входных данных, либо его скорости выполнения по сравнению с другими алго­ритмами для решения той же задачи. Для достижения данной цели визуализация алгоритма использует графические элементы (точки, отрезки, прямоугольники или параллелепипеды и т.д.) для представления некоторых “интересных событий” в работе алгоритма. Имеются два основных варианта визуализации алгоритма:

* статическая визуализация
* динамическая визуализация, именуемая также анимацией алгоритма.

Статическая визуализация алгоритма представляет выполнение алгоритма по­средством серии изображений. Анимация алгоритма, в свою очередь, использует непрерывную презентацию действий алгоритма в стиле мультфильма. Анима­ция — более привлекательный выбор, но, конечно, и существенно сложнее в реа­лизации.

Первые попытки визуализации алгоритмов относятся к 1970-м годам. Пере­ломным моментом стало появление в 1981 году классической визуализации ал­горитма — тридцатиминутного цветного звукового фильма Сортировка за сорти­ровкой. Фильм был создан в университете Торонто Рональдом Беккером (Ronald Baecker) при участии Д. Шермана (D. Sherman) [9, 10]. Фильм содержал визуали­зацию девяти популярных алгоритмов сортировки (более половины из них будут рассмотрены позже в данной книге) и предоставлял убедительную демонстрацию их относительных скоростей.

Успех фильма Сортировка за сортировкой сделал алгоритмы сортировки несомненными фаворитами анимации алгоритмов. В самом деле, проблема сор­тировки естественным образом визуализируется с использованием вертикальных или горизонтальных прямоугольников различной высоты или длины, которые рас­ставляются в соответствии с их размерами (рис. 2.8). Такое представление, одна­ко, удобно лишь для иллюстрации работы алгоритмов сортировки при небольших входных данных. Для входных данных большего размера в фильме Сортировка за сортировкой использована остроумная идея представить данные точками на плоскости, где первая координата точки представляет позицию элемента в мас­сиве данных, а вторая — его значение. При использовании такого представления

данных сортировка представляет собой трансформацию случайного заполнения точками в точки, лежащие на диагонали кадра (рис. 2.9). Кроме того, большин­ство алгоритмов сортировки работает путем поочередного сравнения и обмена двух элементов — события, которое относительно просто визуализировать.

После появления Сортировки за сортировкой было создано множество раз­личных анимаций алгоритмов. Анимации могут содержать как один алгоритм, так и группу алгоритмов для решения одной и той же задачи (например, сорти­ровки) или из одной и той же предметной области (например, геометрические алгоритмы), а также представлять собой анимационные системы общего назначе­ния. Наиболее популярные системы общего назначения включают BALSA [24], TANGO [112] и ZEUS [23]; сравнительный обзор возможностей этих (и еще девя­ти других) пакетов можно найти в [91]. Хорошая анимационная система общего назначения должна позволять пользователю не только просматривать и взаимодей­ствовать с имеющимися анимациями, но и позволять создавать новые анимации. Практика показывает, что создание таких систем — сложная, но не невозможная задача.

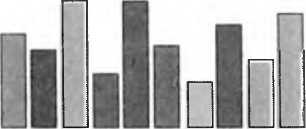
Появление Java и World Wide Web дало новый толчок развитию анимации алгоритмов. Рекомендую обратиться к Internet и познакомиться с образцами ани­мации алгоритмов. Поскольку мир Web очень нестабилен, здесь не приводятся конкретные адреса — попробуйте осуществить поиск по словам “algorithm anima­tion” или “algorithm visualization”. При рассмотрении и оценке различных анима­ций алгоритмов можно пользоваться приведенными ниже “десятью заповедями анимации алгоритмов” — списком желательных возможностей пользовательского интерфейса, предложенным Питером Глуром (Peter Gloor) [42], главным разра­ботчиком Animated Algorithms — еще одной популярной системы визуализации алгоритмов.

1. Последовательность.
2. Интерактивность.
3. Ясность и краткость.
4. Снисходительность к пользователю и прощение его ошибок.
5. Адаптация к уровню знаний пользователя.
6. Упор на визуальную часть.
7. Удержание интереса пользователя.
8. Включение символьного и пиктограммного представления.
9. Включение анализа алгоритма (статистики выполнения) и сравнение с другими алгоритмами для решения той же задачи.
10. Включение истории выполнения.

**\* applet Viewer Son Demn.class**

**Applet**

PROBLEM [Sort **▼]ALGORITHM** SelectionS( I\* **Wj**



Sort Step Back **|Reset[** Exit

**EES!**

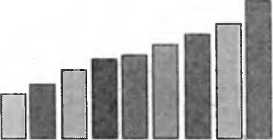
**I Applet**

**let Vlewrr Sort Demo, , »**

PROBLEM [Sort

-inlxf

3 ALGORITHM **1 Selection Sort** ^



**Sort**

**Back**

Reset

**Step**

**Selection Sort has completed - Number of comparisons: 45, Number of exchanges: 8**

Рис. 2.8. Начальный и конечный экраны типичной визуализации алгоритма сортировки с использованием прямоугольников

**j Applet Viewer SortDemp.dass**

**Applet**

**PROBLEM Sort**

▼ ALGORITHM [ Selection Sort

Sort

**Exit**

Step Back **[Resell**

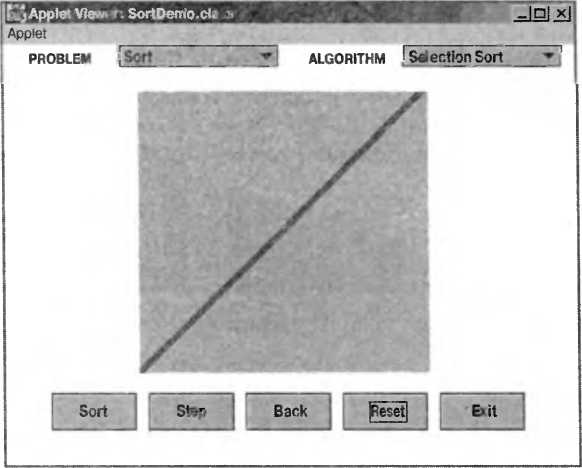


Рис. 2.9. Начальный и конечный экраны типичной визуализации алгоритма сортировки с использованием точек

Основные области применения визуализации алгоритмов — научные исследо­вания и обучение. Применение визуализации алгоритмов в образовании призвано помочь студентам в изучении алгоритмов. Потенциальная польза визуализации в исследовательской работе заключается в возможности открытия некоторых пока что не известных свойств алгоритма. Например, один исследователь использовал визуализацию рекурсивного алгоритма для решения задачи о ханойских башнях, в которой четные и нечетные диски имели разные цвета. Он обратил внимание на то, что два диска одного цвета никогда не находятся в непосредственном контакте в процессе выполнения алгоритма. Это наблюдение помогло ему разработать луч­шую нерекурсивную версию классического алгоритма. Однако, хотя и имеется масса сообщений об успешном применении визуализации в образовании и на­учных исследованиях, эти успехи не столь впечатляющи, как можно было бы ожидать. Опыт показывает, что создания сложной программной системы недо­статочно — требуется глубокое понимание восприятия визуальной информации человеком, чтобы добиться полного раскрытия потенциала анимации алгоритма.

Резюме

* Различают два вида эффективности алгоритмов — временную и про­странственную. Временная эффективность указывает, насколько быст­ро выполняется алгоритм; пространственная эффективность пока­зывает, как много дополнительной памяти требуется алгоритму для работы.
* Временная эффективность алгоритма в основном определяется как функция от размера входных данных, путем подсчета количества вы­полнения базовых операций. Базовая операция — это операция, которая вносит основной вклад в общее время выполнения алгоритма. Обыч­но это операция с наибольшим временем работы в наиболее глубоко вложенном цикле.
* Для ряда алгоритмов время работы существенно отличается для разных входных данных одного и того же размера, что приводит к эффектив­ности для наихудшего, наилучшего и среднего случаев.
* Разработана схема анализа временной эффективности алгоритма, ко­торая основывается на порядке роста времени работы алгоритма при росте размера его входных данных до бесконечности.
* Для указания и сравнения асимптотических порядков роста функций, выражающих эффективность алгоритмов, используются О-, VL- и 0- обозначения.
* Эффективность подавляющего большинства алгоритмов подразделяет­ся на несколько классов — константную, логарифмическую, линейную, п log ?г, квадратичную, кубическую и экспоненциальную.
* Главным инструментом анализа временной эффективности нерекур­сивного алгоритма является построение выражения для суммы коли­чества выполнений его основной операции и выяснение порядка ее роста.
* Главным инструментом анализа временной эффективности рекурсив­ного алгоритма является построение рекуррентного соотношения для количества выполнений его основной операции и выяснение порядка его роста.
* Лаконичность рекурсивного алгоритма может скрывать его неэффек­тивность.
* Числа Фибоначчи представляют собой важную последовательность це­лых чисел, в которой каждый элемент равен сумме двух своих пред­шественников. Имеется несколько алгоритмов вычисления чисел Фи­боначчи с существенно различной эффективностью.
* Эмпирический анализ алгоритма осуществляется путем выполнения программы, реализующей алгоритм, для некоторого образца входных данных и анализа полученных результатов (количества выполненных базовых операций или физического времени работы программы). За­частую при этом используются псевдослучайные числа. Главным пре­имуществом этого подхода является его применимость к любым ал­горитмам, а недостатком — зависимость результатов эксперимента от конкретного компьютера и образца исходных данных.
* Визуализация алгоритма представляет собой использование изображе­ний для вывода полезной информации об алгоритме. Два основных варианта визуализации алгоритма — статическая визуализация и дина­мическая визуализация (именуемая также анимацией алгоритма).

Глава

Метод грубой силы

Наука так же далека от грубой силы, как этот меч от лома.

— Эдуард Литтон (Edward Lytton) (1803-1873) Лейла (Leila), книга II, глава I

Делать что-либо хорошо — зачастую понапрасну терять время.

— Роберт Бирн (Robert Byrne), игрок в бильярд и писатель

О

бсудив схемы и методы анализа алгоритмов в предыдущих главах книги, мы готовы перейти к рассмотрению методов разработки алгоритмов. Каждая из последующих семи глав посвящена некоторой конкретной стратегии разработки. Тема данной главы — простейший метод грубой силы, именуемый также методом решения “в лоб”. Этот метод можно описать следующим образом.

Метод грубой силы представляет собой прямой подход к решению задачи, обычно основанный непосредственно на формулировке задачи и определениях используемых ею концепций.

“Сила” в определении стратегии — сила компьютера, а не сила интеллекта, т.е. сила из пословицы “Сила есть — ума не надо”. Перефразировать определение данной стратегии можно проще: “Нечего думать, надо действовать!”. Зачастую стратегия грубой силы оказывается наиболее простой в применении.

В качестве примера рассмотрим задачу возведения в степень: вычисление ап для некоторого числа а и неотрицательного целого п. Хотя эта задача может показаться тривиальной, она позволяет проиллюстрировать несколько методов разработки алгоритмов, в том числе и подход грубой силы. (Заметим в скобках, что вычисление значения ап mod т для больших целых чисел является основ­ным компонентом ведущих алгоритмов шифрования.) По определению возведе­ния в степень

Отсюда сразу следует простейший алгоритм вычисления ап — путем умноже­ния на а начального значения, равного 1, п раз.

Мы уже встречались в этой книге как минимум с двумя алгоритмами с исполь­зованием грубой силы: последовательная проверка всех целых чисел при поиске gcd (га, п) (раздел 1.1) и алгоритм умножения матриц, основанный на определе­нии данной операции (раздел 2.3). Множество других примеров вы найдете в этой главе. (Можете ли вы указать несколько известных вам алгоритмов, основанных на использовании грубой силы?)

Хотя метод грубой силы редко дает искусные или эффективные алгоритмы, его рассмотрение нельзя опустить, поскольку данный метод представляет собой важную стратегию разработки алгоритмов. Во-первых, в отличие от других стра­тегий, метод грубой силы применим к очень широкому диапазону задач. (Похо­же, это единственный подход, для которого существенно сложнее указать задачу, для решения которой он неприменим.) В частности, метод грубой силы исполь­зуется для многих элементарных, но важных алгоритмических задач, таких как вычисление суммы п чисел, поиск наибольшего элемента в списке и тому по­добных. Во-вторых, для некоторых важных задач (например, сортировки, поиска, умножения матриц, поиска подстрок) метод грубой силы дает вполне рациональ­ные алгоритмы. В-третьих, стоимость разработки более эффективного алгоритма может оказаться неприемлемой, если требуется решить только несколько экзем­пляров задачи, а алгоритм, основанный на грубой силе, позволяет решить их за приемлемое время. В-четвертых, даже будучи неэффективным в общем случае, метод грубой силы может оказаться полезен для решения небольших по разме­ру экземпляров задачи. Наконец, алгоритм, основанный на грубой силе, может служить для важных теоретических или дидактических целей, например мерилом для определения эффективности других алгоритмов для решения данной задачи.

1. Сортировка выбором и пузырьковая сортировка

В этом разделе мы рассмотрим применение метода грубой силы к задаче сор­тировки, которая заключается в следующем: дан список из п упорядочиваемых элементов (например, чисел, символов некоторого алфавита, символьных строк), которые надо разместить в неубывающем порядке. Как упоминалось в разделе 1.3, имеются десятки алгоритмов, разработанных для решения этой очень важной за­дачи. Некоторые из них вам уже, возможно, встречались. В таком случае попы­тайтесь на время забыть о них.

Итак, считая, что вы ничего не знаете о сортировке, задайте себе вопрос: какой метод решения задачи сортировки наиболее прост и непосредственен? Кое- кто может не согласиться, но наиболее простыми представляются два алгоритма, рассматриваемые далее, а именно сортировка выбором и пузырьковая сортировка.

Первый из этих алгоритмов кажется более предпочтительным, хотя бы потому, что более очевидно использует метод грубой силы.

**Сортировка выбором**

Мы начинаем сортировку выбором с поиска наименьшего элемента в списке и обмена его с первым элементом (таким образом, наименьший элемент помеща­ется в окончательную позицию в отсортированном списке). Затем мы сканируем список, начиная со второго элемента, в поисках наименьшего среди оставшихся п— 1 элементов и обмениваем найденный наименьший элемент со вторым, т.е. по­мещаем второй наименьший элемент в окончательную позицию в отсортирован­ном списке. В общем случае, при г-ом проходе по списку (0 ^ г ^ п — 2) алгоритм ищет наименьший элемент среди последних п—г элементов и обменивает его с А^:

i 1

А*0* < Ах< ••• < А{\_{ | Ai,...,Amin,...,An\_l

Элементы в окончательных позициях Последние п - i элементов

После выполнения п — 1 проходов список оказывается отсортирован.

Вот псевдокод данного алгоритма, в котором для простоты предполагается, что список реализован в виде массива.

Алгоритм SelectionSort (А [0..п - 1])

// Сортировка массива методом выбора

// Входные данные: Массив А[0..п — 1] упорядочиваемых

// элементов

// Выходные данные: Массив А[0..п — 1], отсортированный // в неубывающем порядке

for г<— 0 to п — 2 do

min <— г

for j <— г + lton — 1 do

if ***A[j] < A[min] min*** <— ***j***

Обмен ***A[i\*** и ***A[min]***

В качестве примера на рис. 3.1 приведена сортировка выбором следующего списка: 89, 45, 68, 90, 29, 34, 17.

Анализ сортировки выбором выполняется достаточно просто. Размер входных данных определяется количеством п элементов в списке. Базовой операцией алго­ритма является сравнение ключей A[j] < А [тгп]. Общее количество сравнений зависит только от размера массива и определяется следующей суммой:

п—2 п—1 71—2 71—2

с<"> = £ £ 1 = £ к» - и - (• +1) +1] = £ (« -1 - \*)•

г=0 **j—i+l** г=0 г=0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 89 | 45 | 68 | 90 | 29 | 34 | 17 |
| 17 | | | 45 | 68 | 90 | 29 | 34 | 89 |
| 17 | СП  см | | 68 | 90 | 45 | 34 | 89 |
| 17 | 29 | со | со  о | 45 | 68 | 89 |
| 17 | 29 | 34 | 45 | | со  о | 68 | 89 |
| 17 | 29 | 34 | 45 | со  со | со  о | 89 |
| 17 | 29 | 34 | 45 | 68 | 89 | со  о |

Рис. 3.1. Сортировка выбором списка 89,45, 68, 90, 29,

34, 17. Каждая строка соответствует одной итерации алгоритма, т.е. сканированию части списка справа от вертикальной черты. Полужирным шрифтом выделены наименьшие элементы, обнаруживаемые при сканиро­вании. Элементы справа от вертикальной черты нахо­дятся в окончательных позициях и не сканируются

Мы уже встречались с последней суммой при анализе алгоритма из приме­ра 2 в разделе 2.3 (так что вы можете вычислить ее самостоятельно). Каким бы способом вы ни вычисляли данную сумму, ответ, конечно, будет одним и тем же, а именно

гг—2 гг—1 гг—2 ,

[c(n) = Y. Е 1 = =](#bookmark115)

г=0 **j^i+1** г=0

Таким образом, для любых входных данных алгоритм сортировки выбором принадлежит 0 (гг2). Заметим, однако, что количество обменов элементов массива равно 0 (п), точнее, ровно гг — 1 — по одному для каждой итерации цикла г. Это свойство отличает сортировку выбором от многих других алгоритмов сортировки.

**Пузырьковая сортировка**

Еще одно применение метода грубой силы к задаче сортировки состоит в срав­нении соседних элементов и их обмене, если они находятся не в надлежащем по­рядке. Неоднократно выполняя это действие, мы заставляем наибольший элемент “всплывать” к концу списка. Следующий проход приведет к всплыванию второго наибольшего элемента, и так до тех пор, пока после гг — 1 итерации список не бу­дет полностью отсортирован, г-ый проход 0 ^ г ^ гг — 2) пузырьковой сортировки можно представить следующей диаграммой:

**7**

A*0*,...,Aj<^Aj + h...,An\_i\_l | <А„\_{

Элементы в окончательных позициях

Ниже приведен псевдокод данного алгоритма.

**Алгоритм BubbleSort (А** **[0..п - 1])**

// Сортировка массива пузырьковой сортировкой // Входные данные: Массив А[0..п — 1] упорядочиваемых

// элементов

// Выходные данные: Массив А[0..п — 1], отсортированный // в неубывающем порядке

for г <— 0 to п — 2 do

for j <— 0 to n — 2 — г do

if A[j + 1] < A[j]

Обмен A[j] и A[j + 1]

Применение описываемого алгоритма к списку 89, 45, 68, 90, 29, 34, 17 пока­зано на рис. 3.2.

9

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 89 | О | 45 |  | 68 |  | 90 |  | 29 |  | 34 | 17 |
|  |  |  | 9 |  |  |  |  |  |  |  |  |
| 45 |  | 89 | О | 68 |  | 90 |  | 29 |  | 34 | 17 |
|  |  |  |  |  | 9 |  | 9 |  |  |  |  |
| 45 |  | 68 |  | 89 | О | 90 | О | 29 |  | 34 | 17 |
| 45 |  | 68 |  | 68 |  | 29 |  | 90 | 9  О | 34 | 17 |
|  |  |  |  |  |  |  |  |  |  |  | 9 |
| 45 |  | 68 |  | 89 |  | 29 |  | 34 |  | 90 | ^ 17 |
| 45 |  | 68 |  | 89 |  | 29 |  | 34 |  | 17 | 190 |
|  | 9 |  | 9 |  | 9 |  |  |  |  |  |  |
| 45 | <г> | 68 | О | 89 |  | 29 |  | 34 |  | 17 | 190 |
|  |  |  |  |  |  |  | 9 |  |  |  |  |
| 45 |  | 68 |  | 29 |  | 89 | О | 34 |  | 17 | 190 |
|  |  |  |  |  |  |  |  |  | 9 |  |  |
| 45 |  | 68 |  | 29 |  | 34 |  | 89 | О | 17 | 190 |
| 45 |  | 68 |  | 29 |  | 34 |  | 17 |  | 89 | 190 |

Рис. 3.2. Два первых прохода пузырьковой сорти­ровки списка 89,45,68, 90, 29, 34,17. Каждая новая строка представляет собой результат обмена двух элементов. Элементы справа от вертикальной чер­ты находятся в окончательных позициях и в после­дующих итерациях алгоритма не участвуют

Количество сравнений ключей в данной версии пузырьковой сортировки оди­наково для всех массивов размером п. Оно представляется суммой, практически идентичной сумме для сортировки выбором:

**п—2 тг—2—г 71—2**

<?(») = Е Е '“ЕК"-[[24]](#footnote-24)-[[25]](#footnote-25))-^1!”

г=0 j=0 г=0

= g(n-l-!) = i^^€0(n2).

г=0

Количество же обменов элементов зависит от входных данных. В наихудшем случае уменьшающегося массива оно равно количеству сравнений:

***Sworst (п) = С (п) =*** 2^П е © (п2) .

Зачастую при использовании метода грубой силы первая версия алгоритма мо­жет быть усовершенствована ценой достаточно скромных усилий. В частности, приведенную сырую версию пузырьковой сортировки можно улучшить, восполь­зовавшись следующим наблюдением: если при проходе по списку не сделано ни одного обмена, значит, список отсортирован, и выполнение алгоритма можно прекратить (задача 3.1.9а). Однако, хотя новая версия и выполняется быстрее для некоторых входных данных, в наихудшем и среднем случае она все равно принад­лежит © (п2). В действительности даже среди элементарных методов сортировки пузырьковая сортировка — не лучший выбор, и если бы не броское название, вы бы могли никогда о ней не услышать. Тем не менее из ее рассмотрения мы вынесли важный урок:

Первое применение метода грубой силы зачастую дает алгоритм, кото­рый можно улучшить ценой весьма скромных усилий.

Упражнения 3.1

в заданной точке xq и определите его класс эффективности в наи­худшем случае.

б) Если ваш алгоритм принадлежит множеству 0 (п2), постарайтесь разработать алгоритм с линейным временем работы.

в) Можно ли разработать алгоритм с более высокой эффективностью, чем линейная?

1. Отсортируйте список букв Е, X, А, М, Р, L, Е в алфавитном порядке при помощи сортировки выбором.
2. Устойчива ли сортировка выбором? (Определение устойчивого алго­ритма сортировки было дано в разделе 1.3.)
3. Можно ли реализовать сортировку выбором для связанных списков с той же эффективностью © (п2), что и для массивов?
4. Отсортируйте список букв Е, X, А, М, Р, L, Е в алфавитном порядке при помощи пузырьковой сортировки.
5. а) Докажите, что если пузырьковая сортировка не делает ни одного

обмена при проходе по списку, то список отсортирован и алгоритм можно завершать.

б) Напишите псевдокод улучшенного таким образом алгоритма.

в) Докажите, что в худшем случае алгоритм имеет квадратичную эф­фективность.

1. Устойчива ли пузырьковая сортировка?
2. Последовательный поиск и поиск подстрок методом грубой силы

В предыдущем разделе мы познакомились с двумя вариантами применения метода грубой силы к задаче сортировки. Теперь рассмотрим два возможных при­менения этой стратегии к задачам поиска. Первая — каноническая задача поиска элемента с данным значением, вторая — задача поиска подстроки.

**Последовательный поиск**

Мы уже сталкивались с алгоритмом, основанном на методе грубой силы, для решения общей задачи поиска — он называется последовательным поиском (см. раздел 2.1). Этот алгоритм просто по очереди сравнивает элементы заданного списка с ключом поиска до тех пор, пока не будет найден элемент с указанным значением ключа (успешный поиск) или весь список будет проверен, но тре­буемый элемент не найден (неудачный поиск). Зачастую применяется простой дополнительный прием: если добавить ключ поиска в конец списка, то поиск обязательно будет успешным, следовательно, можно убрать проверку завершения списка в каждой итерации алгоритма. Далее приведен псевдокод такой улучшен­ной версии; предполагается, что входные данные имеют вид массива.

**Алгоритм *SequentialSearch****2* **(*А*** **[0..п]**, ***К)***

// Алгоритм реализует последовательный поиск

// с использованием ключа поиска в качестве ограничителя

// Входные данные: Массив Айз п элементов и ключ поиска К

II Выходные данные: Позиция первого элемента массива

// А[0..п — 1], значение которого совпадает

// с К\ если элемент не найден, возвращается

// значение —1

***А[п]*** <- ***К***

г <— О

**while A[i\ ф К** **do г** <— **г** + 1 **if** г **<** гг **return** г **else**

**return —1**

Если исходный список отсортирован, можно воспользоваться еще одним усо­вершенствованием: поиск в таком списке можно прекращать, как только встре­тится элемент, не меньший ключа поиска.

Последовательный поиск представляет превосходную иллюстрацию метода грубой силы, с его характерными сильными (простота) и слабыми (низкая эффек­тивность) сторонами. Эффективность, вычисленная в разделе 2.1 для стандартной версии последовательного поиска, очень мало отличается от эффективности улуч­шенной версии последовательного поиска, так что алгоритм остается линейным как в наихудшем, так и в среднем случае. Позже мы познакомимся с некоторыми алгоритмами, имеющими более высокую эффективность в среднем случае.

**Поиск подстроки**

Вспомним описание задачи поиска подстроки из раздела 1.3: дана символь­ная строка длиной гг, называющаяся текстом, и строка длиной т (га ^ п), именуемая шаблоном (pattern); требуется найти в тексте подстроку, соответству­ющую шаблону. Говоря точнее, мы хотим определить г — индекс крайнего сле­ва символа первой соответствующей шаблону подстроки в тексте — такой, что = РО: • • • j ti+j = Pji • • • ? ti+m— 1 = Pm—1-

**to • • \* • \* • ti+m—l • • • ^71—1 ТеКСТ T**

**l l I**

Po ... Pj ... Pm-1 Шаблон ***P***

Если требуется найти все такие подстроки, алгоритм поиска подстроки может просто продолжать работу до полной проверки текста.

Алгоритм, основанный на грубой силе, очевиден: выровняем шаблон с нача­лом текста и начнем сравнивать соответствующие пары символов слева направо до тех пор, пока не убедимся, что символы во всех т парах равны (в этом случае алгоритм может прекращать работу), либо не встретим пару разных символов. В последнем случае шаблон смещается на одну позицию вправо, и сравнение символов продолжается, начиная с первого символа шаблона и символа в соот­ветствующей позиции в тексте. Заметим, что последняя позиция в тексте, которая еще может выступать в роли начала искомой подстроки — и — т (если позиции в тексте индексируются значениями от 0 до гг — 1). После этой позиции в тек­сте остается слишком мало символов, чтобы они могли соответствовать шаблону. Следовательно, по достижении указанной позиции алгоритм не должен делать никаких сравнений.

АЛГОРИТМ BruteForceStringMatch (T [0..n — 1], P [0..m — 1])

11 Алгоритм реализует поиск подстроки методом грубой силы // Входные данные: массив Т[0..п — 1] из гг символов,

// представляющий текст;

// массив jP [0..7тг — 1] из т символов,

// представляющий шаблон

// Выходные данные: позиция первого символа в тексте,

// с которой начинается первая искомая

// подстрока, соответствующая шаблону; если

// подстрока не найдена, возвращается —1

for г <— 0 to п — т do

***3+-0***

while j < т and P\j] = T[i + j] do

if j = m return i return -1

Работа алгоритма проиллюстрирована на рис. 3.3.

Обратите внимание, что в данном примере алгоритм практически всегда сме­щает шаблон после первого же сравнения символа. Однако наихудший случай намного неприятнее: алгоритм может выполнять все т сравнений перед сдви­гом шаблона, и это происходит в каждой из п — т + 1 попыток (в задаче 3.2.6

N О В О D Y — NOT I CED— HIM NOT NOT NOT NOT NOT NOT NOT NOT

Рис. 3.3. Пример поиска подстроки с применением грубой силы.

Символы шаблона, сравниваемые с символами текста, выделены полужирным шрифтом

требуется привести пример такой ситуации). Таким образом, в наихудшем случае алгоритм имеет время работы 0 (пт). Однако в случае типичного поиска слова в тексте на естественном языке можно ожидать, что большинство сдвигов будет выполняться после небольшого количества сравнений (взгляните на приведенный пример). Таким образом, эффективность в среднем случае должна быть суще­ственно выше эффективности в худшем случае. Это так и есть на самом деле: можно показать, что при поиске в случайных текстах эффективность оказывается линейной, т.е. равной 0 (п + т) = 0 (гг). Для поиска подстрок имеется масса бо­лее интеллектуальных, и эффективных алгоритмов. Наиболее широко известный из них — алгоритм Р. Бойера (R. Воуег) и Дж. Мура (J. Мооге) — будет рассмотрен в разделе 7.2 вместе с упрощением, предложенным Р. Хорспулом (R. Horspool).

Упражнения 3.2

1. Найдите количество сравнений, выполняемых версией последователь­ного поиска с ограничителем

а) в наихудшем случае;

б) в среднем случае при вероятности успешного поиска р (0 ^ р ^ 1).

1. Как было показано в разделе 2.1, среднее количество сравнений клю­чей, выполняемых при последовательном поиске (без ограничителя, при стандартных предположениях о входных данных), определяется формулой

Cavg (п) = ***Р* ^2+ ^ +** П **(1 -** р)

где р — вероятность успешного поиска. Для фиксированного п опре­делите значения р 0 ^ р ^ 1), для которых формула дает наибольшее и наименьшее значения CaVg (п).

1. Разработайте и выполните эксперимент по сравнению времени работы двух версий последовательного поиска, описанных в разделах 2.1 и 3.2.
2. Определите количество сравнений символов, которые будут выпол­нены алгоритмом, основанном на грубой силе, при поиске шаблона GANDHI в тексте

THERE\_IS\_MORE\_TO\_LIFE\_THAN\_INCREASING\_ITS\_SPEED

(Будем считать, что длина текста — 47 символов — известна до начала поиска.)

1. Сколько сравнений (успешных и неудачных) сделает алгоритм поис­ка подстроки, основанный на грубой силе, для следующих шаблонов в тексте, состоящем из 1000 нулей?

а) 00001 б) 10000 в) 01010

1. Приведите пример текста длиной п и шаблона длиной т, которые пред­ставляют собой входные данные для алгоритма поиска подстроки на основе грубой силы, приводящие к наихудшему случаю. Какое именно количество сравнений выполняется для таких входных данных?
2. Напишите псевдокод алгоритма поиска подстрок, основанного на гру­бой силе, который для заданного шаблона возвращает общее количе­ство соответствующих ему подстрок в данном тексте.

image59

1. Если вы ищете шаблон, содержащий редко встречающиеся символы (например, Q или Z в английском тексте), то как вы модифицируете алгоритм, основанный на грубой силе, чтобы воспользоваться этой информацией?

image60

1. Популярная в Америке игра в слова состоит в том, что игрок должен найти каждое из заданного множества слов в квадратной таблице, за­полненной отдельными буквами. Слова можно читать по горизонтали, вертикали или диагонали в любом направлении. Разработайте алгоритм для этой игры, основанный на использовании грубой силы, и реали­зуйте его в виде компьютерной программы.
2. Напишите программу для игры “морской бой”, которая основана на версии поиска шаблона с использованием грубой силы. Правила игры очень просты. В игре участвуют два игрока (в нашем случае — человек и компьютер). Игра происходит на двух одинаковых полях размером 10 х 10 квадратов, где игроки размещают свои корабли, невидимые для противника. У каждого игрока по 5 кораблей, каждый из которых зани­мает определенное количество квадратов поля: эсминец (2 квадрата), подводная лодка (3 квадрата), крейсер (3 квадрата), линкор (4 квадра-

та) и авианосец (5 квадратов).[[26]](#footnote-26) Корабли размещаются горизонтально и вертикально, причем не должны касаться друг друга. Игроки пооче­редно “стреляют”, называя координаты выстрела. Результат каждого выстрела может быть “попал” или “промазал”. При попадании попав­ший игрок делает внеочередной ход, и так до первого промаха. Цель игры — первым потопить все корабли противника (корабль считается потопленным, если попадания были во все составляющие его клетки).

1. Задачи поиска пары ближайших точек и вычисления выпуклой оболочки с использованием грубой силы

В этом разделе мы рассмотрим методы грубой силы для решения двух по­пулярных задач, оперирующих с конечным множеством точек на плоскости. Эти задачи, помимо теоретического интереса, используются в таких важных приклад­ных областях, как вычислительная геометрия и анализ операций.

**Поиск пары ближайших точек**

Задача поиска пары ближайших точек заключается в том, чтобы во множестве из п точек найти две, расположенные друг к другу ближе других. Для просто­ты мы будем рассматривать только двумерный случай, хотя задача может быть поставлена и для большего числа измерений. Точки задаются стандартным спо­собом с помощью декартовых координат (х, у), а расстояние между двумя точками Pi — (pCii Vi) и Pj — (xj)Vj) представляет собой стандартное расстояние геомет­рии Евклида (евклидово расстояние)

***d(PuPj) = \J*** *{Xi* ***~~*** *Xj)* + (уг ***~ У*** *j)* •

Подход с применением грубой силы для решения этой задачи приводит нас к следующему очевидному алгоритму: вычислить расстояния между каждой парой точек и найти пару с наименьшим расстоянием. Естественно, мы хотим избежать повторного вычисления расстояния между одними и теми же точками, поэтому рассматриваем только пары точек (Р\*, Pj), для которых г < j.

Алгоритм *BruteForceClosestPoints* (*Р)*

// Входные данные: список Р, состоящий из гг ^ 2 точек

Ч Р\ = (xl j Уг): • • • ? Рп = Уп)

// Выходные данные: индексы index 1 и index2 пары

// ближайших точек

drain <— оо

**for** г 1 **to п —** 1 **do**

for j <— г + 1 to п do

d <— sqrt((xi — Xj) \* \*2+ 11 sqrt — функция вычислен™

+(Уг — У j) \* \*2) 11 квадратного корня,

if d < drain 11 \*\* — возведение в степень

*drain* <— *d\ index* 1 <— г; *index*2 <— *j;* return *index* 1, *index*2

Базовой операцией алгоритма является вычисление расстояния между двумя точками. В век электронных калькуляторов с кнопкой вычисления квадратного корня может показаться, что это очень простая операция, как, скажем, сложение или умножение. Однако это не так. Начнем с того, что даже для большинства це­лых чисел квадратный корень является иррациональным числом, так что он может быть вычислен только приближенно. Более того, такие приближенные вычисле­ния — отнюдь не тривиальная задача. Однако все оказывается гораздо интереснее: вычисления квадратного корня в алгоритме вполне можно избежать! (Можете ли вы придумать, каким образом?) Фокус в том, что можно просто проигнорировать функцию для вычисления квадратного корня и сравнивать значения (х\* ~ xj)2 + + {уг — г/j)2. Мы можем так поступить, поскольку чем меньше число, для которого мы вычисляем квадратный корень, тем меньше и вычисленный квадратный ко­рень. Говоря строгим математическим языком, функция квадратного корня строго возрастающая.

Так что, если заменить d <- sqrt((xi - xj) \* \*2 + (yi - Vj) \* \*2) на

d<-(xi- Xj) \* \*2 + (Уг - Vj) \* \*2

то базовой операцией алгоритма станет возведение чисел в квадрат. Общее коли­чество таких операций можно вычислить следующим образом:

гг—1 гг п—1

С (п) = ^2 2 — 2 ^ (гг — г) = 2 [(гг — 1) + (п — 2) Н Ь 1] =

г=1 *j=i+l i*—1

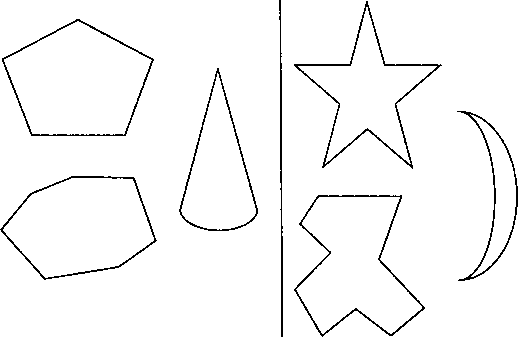
= (n — 1) п е 0 (гг2) .

В главе 4 мы рассмотрим алгоритм для решения этой задачи, класс эффектив­ности которого — п log п.

**Поиск выпуклой оболочки**

Теперь рассмотри другую задачу — вычисление выпуклой оболочки. Начнем с определения того, что такое выпуклая оболочка.

Определение 1. Множество точек (конечное или бесконечное) на плоскости на­зывается выпуклым, если для любых двух точек Р и Q, принадлежащих данному множеству, отрезок с конечными точками Р и Q будет полностью принадлежать этому же множеству. щ



а)

б)

Рис. 3.4. а) Выпуклые множества, б) Множества, не явля­ющиеся выпуклыми

Все множества на рис. ЗАа — выпуклые; выпуклыми являются обычная пря­мая, треугольник, прямоугольник (говоря более обобщенно — любой выпуклый многоугольник[[27]](#footnote-27)), круг, вся плоскость. Множества, показанные на рис. 3.46, любое множество, состоящее из двух различных точек, граница выпуклого многоуголь­ника, окружность -- все это примеры множеств, не являющихся выпуклыми.

Теперь можно перейти к понятию выпуклой оболочки. Интуитивно выпуклая оболочка множества из п точек на плоскости — это наименьший выпуклый мно­гоугольник, который содержит все точки множества — внутри или на границе. Если такое определение не вызывает у вас энтузиазма, можно воспользоваться интерпретацией Д. Харела (D. Harel) [48]: как обнести п спящих тигров забором минимальной длины? Проблема в такой формулировке только в том, как вбить столб для забора в том месте, где спит тигр?... Есть и еще одна, не такая страш­ная интерпретация. Предположим, что каждая точка представлена гвоздем, вби­тым в деревянную поверхность, представляющую плоскость. Возьмем резиновую ленту, растянем ее так, чтобы она охватывала все гвозди, и отпустим. Выпук­лая оболочка представляет собой область, ограниченную отпущенной резиновой лентой (рис. 3.5).

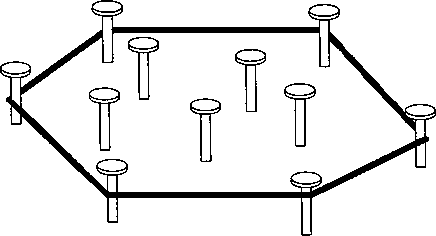


Рис. 3.5. Интерпретация выпуклой оболочки с помощью резиновой ленты

Формальное определение выпуклой оболочки применимо к произвольным множествам, включающим множества точек, лежащих на одной прямой, и другие подобные экстремальные случаи.

Определение 2. Выпуклая оболочка множества точек S представляет собой наи­меньшее выпуклое множество, содержащее S. (“Наименьшее” означает, что вы­пуклая оболочка S должна быть подмножеством любого выпуклого множества, содержащего S.) я

Если множество S выпукло, то очевидно, что выпуклая оболочка S совпадает с S. Если S — множество, состоящее из двух точек, то его выпуклая оболоч­ка представляет собой отрезок, соединяющий эти точки. Для множества из трех точек, не лежащих на одной прямой, выпуклая оболочка представляет собой тре­угольник с вершинами в этих трех точках; если же три точки лежат на одной прямой, то выпуклая оболочка является отрезком, соединяющим две, наиболее удаленные из них. На рис. 3.6 приведен пример выпуклой оболочки множества большего размера.

Изучение примеров позволяет сформулировать следующую теорему.

Теорема 1. Выпуклая оболочка произвольного множества S, состоящего из п > 2 точек (не лежащих на одной прямой), представляет собой выпуклый многоуголь­ник с вершинами в некоторых точках S. (Если все точки лежат на одной прямой, то многоугольник вырождается в отрезок прямой, две конечные точки которого все равно принадлежат множеству S.) ■

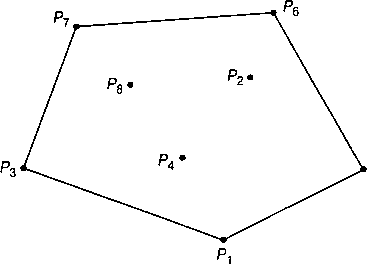


Рис. 3.6. Выпуклая оболочка для множе­ства из 8 точек, представляющая собой вы­пуклый многоугольник

Вычисление выпуклой оболочки представляет собой задачу построения вы­пуклой оболочки для множества S, состоящего из п точек. Для ее решения необ­ходимо найти точки, которые служат вершинами искомого многоугольника. Мате­матики называют вершины такого многоугольника “угловыми точками” (extreme points). По определению, угловая точка выпуклого множества, которая не явля­ется срединной точкой никакого отрезка с конечными точками из данного мно­жества. Например, угловыми точками треугольника являются три его вершины; угловыми точками круга — все точки его окружности; угловыми точками выпук­лой оболочки восьми точек, показанной на рис. 3.6, — точки Р\, Р5, Pq, Р7 и Р3.

Угловые точки имеют ряд особых свойств, отсутствующих у других точек выпуклого множества. Одно из таких свойств используется очень важным алго­ритмом — так называемым симплекс-методом (simplex method). Этот алгоритм предназначен для решения задач линейного программирования, т.е. задач поис­ка минимума или максимума линейной функции от п переменных, на которые накладываются линейные ограничения (см. в качестве примера задачу 3.3.9 или обсуждение в разделе 6.6). Сейчас же угловые точки нас интересуют постоль­ку, поскольку их идентификация решает задачу вычисления выпуклой оболочки. В действительности для решения этой задачи надо знать не только угловые точки, но и то, какие пары этих точек должны быть соединены для получения границы выпуклой оболочки. Заметим, что ответ на этот вопрос может быть получен при перечислении угловых точек в порядке обхода по или против часовой стрелки.

Так как же решить задачу вычисления выпуклой оболочки с применением гру­бой силы? Если у вас нет плана немедленной фронтальной атаки, не отчаивайтесь: задача вычисления выпуклой оболочки не имеет очевидного алгоритмического ре­шения. Тем не менее имеется простой, но неэффективный алгоритм, основанный на следующем свойстве отрезков, образующих границу выпуклой оболочки: от­резок, соединяющий точки Р\* и Pj множества, состоящего из п точек, является частью границы выпуклой оболочки тогда и только тогда, когда все прочие точки лежат по одну сторону от прямой, проходящей через эти две точки.[[28]](#footnote-28) (Проверь­те это свойство для множества, показанного на рис. 3.6.) Проверка каждой пары точек дает список отрезков, которые составляют границу выпуклой оболочки.

Для реализации этого алгоритма требуется вспомнить несколько элементар­ных фактов из аналитической геометрии. Во-первых, прямая линия, проходящая через две точки (х\, yi) и (х2, у2) на координатной плоскости, описывается урав­нением ах Л-by = с, где а = у2 - уь Ь = х\ - х2 и с = х\у2 - У\х2.

Во-вторых, такая прямая делит плоскость на две полуплоскости: для всех точек одной из полуплоскостей ах + by > с, а для точек второй — ах -\-Ъу < с (естественно, для точек на прямой справедливо соотношение ах + Ьу = с). Таким образом, для того, чтобы проверить, лежат ли точки по одну сторону прямой, достаточно вычислить для них значения ах by — с и выяснить, во всех ли точках эти значения имеют одинаковый знак. Реализация деталей этого алгоритма остается читателям в качестве самостоятельного упражнения.

Эффективность данного алгоритма равна О (гг3): для каждой из п (гг — 1)/2 различных пар точек понадобится найти знаки выражений ах + by — с для гг — 2 точек. Для решения этой задачи существуют более эффективные алгоритмы; ко­торые будут рассмотрены позже.

Упражнения 3.3

1. Можете ли вы разработать более быстрый алгоритм, основанный на методе грубой силы, с помощью которого выполнялся бы поиск па­ры ближайших точек для п точек xi,..., хп на действительной оси координат?
2. а) Имеется ряд альтернативных определений расстояний между двумя

точками Pi = (xi,yi) и Р2 = (х2, у2). В частности, так называемое манхэттенское расстояние определяется как

***dM (Р****11****Р****2****) =*** *\Х1* ***- х****2****\*** + |yi - У**2**1 •

Докажите, что dM удовлетворяет следующим аксиомам, которым должна удовлетворять любая функция расстояния:

1. **йм (Pi,P**2**)** ^ 0 для любых двух точек Pi и Р2, и **dM** (Pi, Р**2**) = О тогда и только тогда, когда Pi = Р2;
2. **dM** (Pi, Р**2**) = **dM** (Р**2**, Pi);
3. Для любых точек Pi, Р2 и Рз справедливо неравенство

Лм (Р\,Рг) **^** dM (Pi, Рз) **+** dM (Рз, Pi)-

б) Изобразите на координатной плоскости все точки, манхэттенское расстояние которых до начала координат (0,0) равно 1. Сделайте то же для евклидова расстояния.

в) Истинно или ложно следующее утверждение: результат поиска пары ближайших точек не зависит от того, какая из двух метрик исполь­зуется — евклидова, cLe, или манхэттенская, с£м?

1. Задача поиска пары ближайших точек может быть поставлена в к- мерном пространстве, где евклидово расстояние между точками Р' = = (x'i,..., х'к) и Р" = (х",..., х") определяется как

***к***

image64

Каков класс эффективности алгоритма поиска пары ближайших точек в fc-мерном пространстве, основанного на методе грубой силы?

1. Найдите выпуклые оболочки следующих множеств и укажите их угло­вые точки (если таковые имеются):

а) отрезок прямой;

б) квадрат;

в) граница квадрата;

г) прямая линия.

1. Какое минимальное и максимальное количество угловых точек может иметь выпуклая оболочка множества из п различных точек?
2. Разработайте алгоритм с линейным временем работы для поиска одной угловой точки выпуклой оболочки множества из п точек на плоскости.
3. Как изменить алгоритм, вычисляющий выпуклую оболочку с помощью грубой силы, чтобы он подходил для случая, когда на одной прямой могут располагаться больше двух точек?
4. Напишите программу, реализующую алгоритм для вычисления выпук­лой оболочки с использованием метода грубой силы.
5. Рассмотрим следующий небольшой экземпляр задачи линейного про­граммирования:

максимизировать Зх + 5 у

при условиях х + у < 4

х + Зу ^ 6 х ^ 0, у ^ 0.

а) Изобразите на координатной плоскости допустимую область (fea­sible region) задачи, т.е. множество точек, удовлетворяющих всем ограничениям из условия задачи.

б) Определите угловые точки области.

в) Решите задачу оптимизации с использованием следующей теоремы: задача линейного программирования с непустой ограниченной до­пустимой областью всегда имеет решение, причем оно находится в одной из угловых точек допустимой области.

1. Исчерпывающий перебор

Многие важные задачи требуют поиска элемента со специальными свойствами в области, экспоненциально (или еще быстрее) растущей с увеличением размера экземпляра задачи. Обычно такие задачи возникают в случаях, когда присутству­ют — явно или неявно — комбинаторные объекты, такие как перестановки, соче­тания или подмножества данного множества. Многие подобные задачи являются задачами оптимизации: в них требуется найти элемент, который максимизирует ил и минимизирует некоторые интересующие нас характеристики, как, например, длина пути или назначенная стоимость.

Исчерпывающий перебор (exhaustive search) представляет собой подход к ком­бинаторным задачам с позиции грубой силы. Он предполагает генерацию всех возможных элементов из области определения задачи, выбор тех из них, которые удовлетворяют ограничениям, накладываемым условием задачи, и последующий поиск нужного элемента (например, оптимизирующего значение целевой функ­ции задачи). Заметим, что, хотя идея исчерпывающего перебора весьма проста, ее реализация обычно требует алгоритма для генерации определенных комбинатор­ных объектов. Отложим подробное рассмотрение таких алгоритмов до главы 5, а пока что будем просто считать, что таковые существуют. Исчерпывающий пе­ребор будет проиллюстрирован применением его к трем важным задачам: задаче коммивояжера, задаче о рюкзаке и задаче о назначениях.

**Задача коммивояжера**

Задача коммивояжера беспокоит умы исследователей уже более 100 лет про­стотой формулировки, важными приложениями и интересными связями с другими комбинаторными задачами. В переводе для любителя задача формулируется так: надо найти такой кратчайший путь по заданным п городам, чтобы каждый го­род посещался только один раз и конечным пунктом оказался город, с которого начиналось путешествие. Проблему удобно смоделировать при помощи взвешен­ного графа, вершины которого представляют города, а веса ребер определяют расстояния. После этого задача может быть сформулирована как задача поис­ка кратчайшего гамильтонового цикла (Hamiltonian circuit) неориентированного графа. (Гамильтоновым именуют цикл, который проходит по всем вершинам гра­фа ровно по одному разу. Он назван так в честь ирландского математика Вильяма Ровена Гамильтона (William Rowan Hamilton) (1805-1865), который интересовался такими циклами в качестве применения своих алгебраических открытий.)

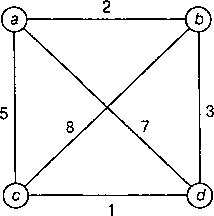
Легко видеть, что гамильтонов цикл можно также определить как последо­вательность п + 1 смежных вершин ... , Uin\_15Ui0, где первая вершина

в последовательности совпадает с последней, в то время как все остальные п — 1 вершин различны. Далее, без потери общности можно предположить, что все циклы начинаются и заканчиваются в одной конкретной вершине (в конце кон­цов, все они циклы, верно?) Значит, можно получйть все возможные маршруты, генерируя все перестановки гг — 1 промежуточных городов, вычисляя длину соот­ветствующих путей и находя кратчайший из них. На рис. 3.7 показан небольшой экземпляр данной задачи и его решение описанным методом.

При внимательном рассмотрении рис. 3.7 можно выявить три пары обходов, которые отличаются друг от друга только направлением. Таким образом, можно снизить количество перестановок вершин вдвое. Можно, например, выбрать две промежуточные вершины, скажем, В и С, и рассматривать только те переста­новки, в которых В предшествует С (такой трюк неявно определяет направление обхода). Однако это улучшение алгоритма не дает заметного эффекта. Общее ко­личество перестановок остается равным (п — 1)!/2, что делает исчерпывающий перебор неприемлемым для всех значений гг, кроме самых малых. С другой сто­роны, если вы — оптимист и видите стакан наполовину полным, а не наполовину пустым, то скажете, что нельзя недооценивать снижения количества работы вдвое, в особенности если она делается вручную. Заметим также, что если бы не по­ставленное ограничение, чтобы все пути начинались в одной и той же вершине, то количество перестановок было бы в п раз больше.

**Задача о рюкзаке**

Вот еще одна известная алгоритмическая задача. Дано п предметов весом wi,..., wn и ценой гц,..., г>п, а также рюкзак, выдерживающий вес W. Требуется найти подмножество предметов, которое можно разместить в рюкзаке, и которое имеет при этом максимальную стоимость. Если вам не нравится представлять себя вором, пытающимся запихнуть в свой рюкзак самые ценные вещи, представьте



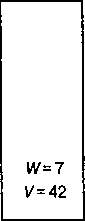
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Путь | |  | Длина |  |
| —> | Ь -> | с —> d —> | a | / = 2 + 8 + 1+ 7=18 |  |
| —> | b -> | d —> с —> | a | / = 2 + 3+1+5=11 | Оптимален |
| —> | с —> | b —> d —> | a | / = 5 + 8 + 3 + 7 = 23 |  |
| —> | с —> | Q.  I   * СГ   I  V | a | /=5+1+3 + 2=11 | Оптимален |
| —> | d -> | b —> с —> | a | / = 7 + 3 + 8 + 5 = 23 |  |
| —> | d —> | 0  1   * Cr   I  V | a | / = 7+1+8 + 2=18 |  |

Рис. 3.7. Решение небольшого экземпляра задачи коммивояжера методом исчерпывающего перебора

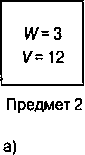
себя транспортным чиновником, которому надо перевезти максимально ценный груз, не превышающий определенного веса. На рис. ЗЯа приведен небольшой экземпляр задачи о рюкзаке.

Исчерпывающий перебор в этой задаче приводит к рассмотрению всех под­множеств данного множества из п предметов, вычислению общего веса каждого из них для того, чтобы выяснить, допустим ли такой набор предметов (т.е. не превосходит ли его общий вес возможности рюкзака), и выбору из допустимых подмножества с максимальным весом. В качестве примера решение экземпля­ра задачи, представленного на рис. ЗЯа, показано на рис. 3.86. Поскольку общее количество подмножеств n-элементного множества равно 2П, исчерпывающий пе­ребор приводит к алгоритму со временем работы Q (2П), вне зависимости от того, насколько эффективным методом генерируются рассматриваемые подмножества.

Таким образом, применение метода исчерпывающего перебора к задачам ком­мивояжера и о рюкзаке приводит к исключительно неэффективным алгоритмам для любых входных данных. На самом деле эти две задачи представляют собой наиболее известные примеры так называемых NP-сложных задач (NP-hard prob­lems). Ни для одной из TVP-сложных задач не известен алгоритм, решающий их за полиномиальное время. Более того, большинство ученых-кибернетиков сходятся



Предмет 1



Рюкзак

|  |  |  |
| --- | --- | --- |
|  | |  |
| 1У=4  V=40 |  | W=5  У=25 |

Предмет 3

Предмет 4

Подмножество Общий вес Общая стоимость

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 0 |
| {1} | 7 | 42 |
| {2} | 3 | 12 |
| {3} | 4 | 40 |
| {4} | 5 | 25 |
| {1,2} | 10 | 36 |
| {1,3} | 11 | Недопустим |
| {1,4} | 12 | Недопустим |
| {2,3} | 7 | 52 |
| {2,4} | 8 | 37 |
| {3,4} | 9 | 65 |
| {1,2,3} | 14 | Недопустим |
| {1,2,4} | 15 | Недопустим |
| {1,3,4} | 16 | Недопустим |
| {2,3,4} | 12 | Недопустим |
| {1,2,3,4} | 19 | Недопустим |

б)

Рис. 3.8. а) Экземпляр задачи о рюкзаке, б) Решение путем исчер­пывающего перебора (оптимальное решение выделено полужирным шрифтом)

во мнении, что такие алгоритмы не существуют вообще, хотя это важное предпо­ложение никем не доказано. Более интеллектуальные подходы, рассматриваемые в разделах 11.2 и 11.2, позволяют решить некоторые (но не все) экземпляры

этих (и подобных) задач за время, меньшее экспоненциального. Можно также воспользоваться одним из приближенных алгоритмов, подобных рассмотренным в разделе 11.3.

**Задача о назначениях**

В третьем примере задачи, которая может быть решена исчерпывающим пе­ребором, имеется п работников, которые должны выполнить п заданий, по од­ному заданию каждый (т.е. каждому работнику назначается только одно задание, и каждое задание назначается лишь одному человеку). Стоимость выполнения г-ым работником j-ro задания известна и равна С [i,j] для всех пар i,j = 1,... п. Задача заключается в следующем: надо распределить задания между работниками таким образом, чтобы они были выполнены с наименьшей общей стоимостью.

Небольшой экземпляр данной задачи приведен в таблице стоимости выполне­ния заданий С [г, j]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Задание 1 | Задание 2 | Задание 3 | Задание 4 |
| Работник 1 | 9 | 2 | 7 | 8 |
| Работник 2 | 6 | 4 | 3 | 7 |
| Работник 3 | 5 | 8 | 1 | 8 |
| Работник 4 | 7 | 6 | 9 | 4 |

Легко видеть, что экземпляр задачи с назначением заданий полностью опре­деляется матрицей стоимости С. В терминах данной матрицы в задаче требуется выбрать по одному элементу из каждой строки матрицы так, чтобы выбранные элементы находились в разных столбцах, а их общая сумма имела наименьшее возможное значение. Заметим, что очевидной стратегии решения данной зада­чи не существует. Например, нельзя выбирать наименьшие элементы в каждой строке, поскольку они могут оказаться в одном и том же столбце. В действитель­ности, наименьшие элементы матрицы могут вообще не входить в оптимальное решение. Так что исчерпывающий перебор для решения данной задачи может оказаться неизбежен.

Допустимое решение задачи о назначениях можно описать в виде кортежа из п значений (Д,... ,jk), в котором г-ый компонент г = 1,..., гг) указывает столбец, где находится выбранный в г-ой строке компонент (т.е. номер задания, назна­ченного г-му работнику). Например, для приведенной выше матрицы стоимости (2,3,4,1) указывает допустимое назначение второго задания первому работнику, третьего задания — второму работнику, четвертого задания — третьему работнику, и первого задания — четвертому работнику. Из условий задачи вытекает, что име­ется однозначное соответствие между допустимыми назначениями и перестанов­ками первых п натуральных чисел. Следовательно, исчерпывающий перебор длязадачи о назначениях может потребовать генерации всех перестановок натураль­ных чисел 1,2,... ,п, вычисления общей стоимости каждого назначения путем суммирования соответствующих элементов матрицы стоимости и окончательного выбора назначения с минимальной стоимостью. Несколько первых шагов приме­нения данного алгоритма к приведенному ранее экземпляру задачи показано на рис. 3.9. Читателям предлагается самостоятельно завершить это упражнение.

9 2 7 8

1. 4 3 7

5 8 18

1. 6 9 4

<1, 2, 3, 4> стоимость = 9 + 4 + 1 +4= 18 <1, 2, 4, 3> стоимость = 9 + 4 + 8 + 9 = 30

и т.д.

<1, 3, 2, 4> стоимость = 9 + 3 + 8 + 4 = 24

<1, 3, 4, 2> стоимость = 9 +3 + 8+ 6 = 26

< 1, 4, 2, 3> стоимость = 9 + 7 + 8 + 9 = 33

<1, 4, 3, 2> стоимость = 9 + 7 + 1 +6 = 23

Рис. 3.9. Несколько первых итераций решения экземпляра задачи о на­значениях путем исчерпывающего перебора

Поскольку в задаче о назначениях в общем случае количество рассматри­ваемых перестановок равно гг!, исчерпывающий перебор непрактичен для всех, кроме очень небольших, значений п. К счастью, для данной задачи имеется суще­ственно более эффективный алгоритм, именуемый венгерским методом в честь открывших его венгерских математиков Кёнига (Konig) и Эгервари (Egervary) (см., например, [69]).

Это хорошая новость. Оказывается, экспоненциальный (или более быстрый) рост области определения задачи не обязательно влечет отсутствие эффективного алгоритма для ее решения. Позже в этой книге вы познакомитесь с другими подобными задачами. Однако такие примеры — скорее исключение, чем правило. Чаще всего для задач, область определения которых растет экспоненциально (при условии, что мы хотим получить точное решение), полиномиальный алгоритм решения неизвестен. И, как было сказано, очень может быть, что такие алгоритмы просто не существуют.

Упражнения 3.4

1. а) К какому классу эффективности относится описанный в тексте алго­ритм исчерпывающего перебора для решения задачи коммивояжера, если считать, что каждый путь может быть сгенерирован за посто­янное время?

б) Пусть такой алгоритм, реализованный в виде компьютерной про­граммы, делает 1 миллиард сложений в секунду. Оцените макси­мальное количество городов, для которых задача коммивояжера мо­жет быть решена за

1. час;
2. сутки;
3. год;
4. столетие.
5. Набросайте схему алгоритма для поиска гамильтонова цикла методом исчерпывающего перебора.
6. Опишите алгоритм, с помощью которого можно определить, содержит ли граф, представленный в виде матрицы смежности, Эйлеров цикл. К какому классу эффективности относится данный алгоритм?
7. Продолжите применение алгоритма исчерпывающего перебора к эк­земпляру задачи о назначениях, приведенному в тексте раздела.
8. Приведите пример задачи о назначениях, оптимальное решение кото­рой не включает наименьшие элементы матрицы стоимости.
9. Напишите реферат о венгерском методе.
10. Рассмотрим задачу разбиения (partition problem): даны п натуральных чисел, которые надо разбить на два непересекающихся подмножества с одинаковой суммой их элементов (понятно, что задача не всегда имеет решение). Разработайте алгоритм решения данной задачи на основе исчерпывающего перебора. Постарайтесь минимизировать количество подмножеств, которые должен сгенерировать алгоритм.
11. Рассмотрим задачу о клике (clique problem): для графа G и натураль­ного числа к надо определить, содержит ли граф клику размера /с, т.е. полный подграф из к вершин. Разработайте алгоритм на основе исчерпывающего перебора для решения данной задачи.

image68

1. Поясните, как применить метод исчерпывающего перебора к задаче сортировки и определите класс эффективности такого алгоритма.
2. Магический квадрат порядка п представляет собой набор чисел от 1 до гг2, размещенных в матрице размером п х гг, причем каждое число встречается в матрице ровна один раз, при этом суммы чисел в каждой строке, каждом столбце и каждой диагонали одинаковы.

а) Докажите, что если магический квадрат порядка п существует, то интересующая нас сумма должна быть равна п (гг2 + 1) /2.

б) Разработайте алгоритм исчерпывающего перебора для генерации всех магических квадратов порядка п.

в) Поищите в Internet или библиотеке более эффективный алгоритм генерации магических квадратов.

г) Реализуйте алгоритм исчерпывающего перебора и найденный вами более эффективный алгоритм и экспериментальным путем опреде­лите максимальный размер магического квадрата, который может быть сгенерирован каждой из реализаций за одну минуту.

Резюме

* Подход к решению задачи с использованием грубой силы обычно ос­нован непосредственно на формулировке задачи и определениях при­меняемых в ней концепций.
* Главные достоинства подхода с использованием грубой силы — ши­рокая применимость и простота; главный недостаток — очень низкая эффективность таких алгоритмов.
* Первое применение подхода с использованием грубой силы для ре­шения задачи обычно дает алгоритм, который можно улучшить ценой достаточно скромных усилий.
* Примером подхода с использованием грубой силы могут служить сле­дующие известные алгоритмы:
* алгоритм умножения матриц, основанный на определении этой операции;
* сортировка выбором;
* последовательный поиск,
* простой алгоритм поиска подстрок.
* Исчерпывающий перебор представляет собой подход с использовани­ем грубой силы для решения комбинаторных задач. Он предполагает генерацию всех комбинаторных объектов задачи, выбор среди них тех, которые удовлетворяют ограничениям из условий задачи, и последую­щий поиск нужного объекта.
* Задача коммивояжера, задача о рюкзаке и задача о назначениях пред­ставляют собой типичные примеры задач, которые могут быть реше­ны — по крайней мере теоретически — при помощи алгоритма исчер­пывающего перебора.
* Исчерпывающий перебор непрактичен для всех, кроме очень неболь­ших, экземпляров задач, к которым он может быть применен.

**Глава**

Метод декомпозиции

О чем бы не молился человек, он молится о чуде. Каждый молящийся приходит к этому — Господи Боже, сделай, чтобы дважды два не было равно четырем.

- И. Тургенев (1818-1883)

М

етод декомпозиции (он же метод “разделяй и властвуй”), вероятно, наи­более популярный метод разработки алгоритмов. Ряд очень эффективных алгоритмов представляют собой реализации этой общей стратегии. Алгоритмы, основанные на методе декомпозиции, работают в соответствии со следующим планом.

1. Экземпляр задачи разбивается на несколько меньших экземпляров той же задачи, в идеале — одинакового размера.
2. Решаются меньшие экземпляры задачи (обычно рекурсивно, хотя ино­гда для небольших экземпляров применяется какой-нибудь другой ал­горитм).
3. При необходимости решение исходной задачи находится путем комби­нации решений меньших экземпляров.

Схема метода декомпозиции показана на рис. 4.1, где приведен случай разде­ления задачи на две равные по размеру подзадачи, что является, пожалуй, самой распространенной ситуацией (по крайней мере для алгоритмов декомпозиции, разработанных для выполнения на однопроцессорном компьютере).

В качестве примера рассмотрим задачу вычисления суммы чисел ао,..., ап~ь Если п > 1, задачу можно разделить на два экземпляра той же задачи: вычисле­ние суммы первых [п/2\ чисел и вычисление суммы оставшихся [гг/2] чисел (понятно, что при п = 1 в качестве ответа просто возвращается значение ао). Как только каждая из этих двух сумм будет вычислена (с применением описан­ного метода, т.е. рекурсивно), мы сможем сложить их значения, чтобы получить окончательный ответ:

а0 Н + О'П-1 — (&0 + \* \* \* + CL[n/2J — l) + (a|n/2j + • \* \* + О'П-1) •

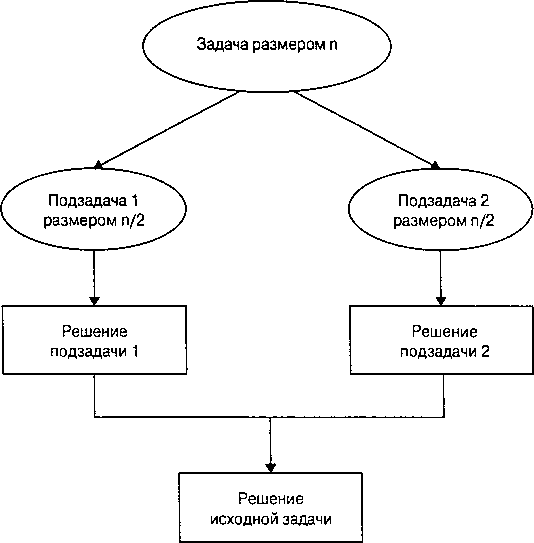


Рис. 4.1. Метод декомпозиции

Является ли описанный алгоритм эффективным способом суммирования п чи­сел? Небольшие размышления по поводу сравнения этого алгоритма с алгоритмом на основе грубой силы, конкретный пример суммирования четырех чисел, фор­мальный анализ (приведенный далее), да и просто здравый смысл — все приводит к отрицательному ответу на этот вопрос.

Таким образом, не каждый алгоритм на основе декомпозиции эффективнее алгоритма, основанного на грубой силе. Но очень часто Бог алгоритмов — вспом­ните эпиграф — прислушивается к молитвам, и время, затраченное на выполне­ние алгоритма на основе декомпозиции, оказывается меньше, чем время реше­ния задачи каким-то другим методом. Метод декомпозиции дал кибернетике ряд очень важных и эффективных алгоритмов. Мы рассмотрим несколько классиче­ских примеров таких алгоритмов в данной главе. Хотя здесь рассматриваются только последовательные алгоритмы, стоит помнить о том, что метод декомпози­ции идеально подходит для параллельных вычислений, когда каждая подзадача может одновременно решаться собственным процессором.

Пример суммирования иллюстрирует наиболее типичный случай декомпо­зиции: экземпляр задачи размером п делится на два экземпляра размером п/2. В общем случае экземпляр задачи размера п может быть разделен на несколько экземпляров размером п/b, а из которых требуется решить (здесь а и Ъ — кон­станты; а ^ 1 и b > 1). Полагая для упрощения анализа, что размер п равен степени Ь, получаем следующее рекуррентное соотношение для времени работы алгоритма Т (п):

***T{n) = aT(n/b) + f(n),*** (4.1)

где / (п) — функция, учитывающая затраты времени на разделение задачи на меньшие подзадачи и комбинирование решений подзадач. (В примере с сумми­рованием чисел а = b = 2 и / (n) = 1.) Рекуррентное соотношение (4.1) называ­ется обобщенным рекуррентным уравнением декомпозиции (general divide-and- conquer recurrence). Очевидно, что порядок роста его решения Т (п) зависит от значений констант а и b и порядка роста функции / (п). Анализ эффективности множества алгоритмов, основанных на декомпозиции, существенно упрощается следующей теоремой (см. приложение Б).

ОСНОВНАЯ ТЕОРЕМА. Если в рекуррентном уравнении (4.1) / (n) е © (nd), где 0, то

{

0 (nd) если a <bd

© (nd log п) если a = bd

0 (nlogba) если a > bd

(Аналогичные результаты выполняются и для обозначений О и П.) ■

Например, рекуррентное соотношение для количества сложений А (п) в опи­санном выше алгоритме суммирования декомпозицией для входных данных раз­мера п = 2к равно

А(/г) = 2А (п/2) + 1.

Таким образом, в данном примере а = 2, b = 2 a d = 0. Следовательно, поскольку а > bd,

А (п) е © (п1обьа) = 0 (nlog22) = © (n).

Обратите внимание, что мы можем указать класс эффективности алгоритма, не решая само рекуррентное уравнение. Конечно же, этот подход позволяет уста­новить порядок роста решения, не определяя неизвестные множители, которые можно найти, только решив рекуррентное уравнение.

1. Сортировка слиянием

Сортировка слиянием (mergesort) представляет собой превосходный пример успешного применения метода декомпозиции. Она сортирует заданный массив А [0..п — 1] путем разделения его на две половины, А [0.. |\_n/2j — 1] и A[|\_n/2J.. п — 1], рекурсивной сортировки каждой половины и слияния двух отсортирован­ных половин в один отсортированный массив.

Алгоритм Mergesort (А [0..п - 1])

// Рекурсивно сортирует массив А[0..п — 1]

// Входные данные: Массив А[0..п — 1] упорядочиваемых

// элементов

// Выходные данные: Отсортированный в неубывающем порядке

// массив А[0..п — 1]

if п > 1

Копировать А[0..|\_n/2j — 1] в В[0..[п/2\ — 1]

Копировать А[\п/2\..п — 1] в C[0..|\_n/2J — 1]

***Mergesort(B[0..[n/2\*** — 1])

***Mergesort(C[0..\n/2\*** — lj)

***Merge{B,C***, ***A)***

Слияние двух отсортированных массивов можно выполнить следующим обра­зом. Два указателя (индекса массивов) после инициализации указывают на первые элементы сливаемых массивов. Затем элементы, на которые указывают указатели, сравниваются, и меньший из них добавляется в новый массив. После этого индекс меньшего элемента увеличивается, и он указывает на элемент, непосредственно следующий за только что скопированным. Эта операция повторяется до тех пор, пока не будет исчерпан один из сливаемых массивов, после чего оставшиеся эле­менты второго массива просто добавляются в конец нового массива.

Алгоритм Merge (В [0..р -\].С [0..? - 1]. А [0..р + q - 1])

// Слияние двух отсортированных массивов в один // Входные данные: Сортированные массивы В[0..р — 1]

// и C[0..q — i]

// Выходные данные: Сортированный массив А[0..р + q — 1],

// состоящий из элементов В а С

г <— 0; j \*— 0; к <— 0 while i < р and j < q do if B[i\ sS C\j]

A[/u] <— i <— i -|-1

**else**

***A[k***] <- ***C\j)\ j \*- j + 1 к* <— *к* + 1** if ***i*** = ***p***

Копировать C[j..q — 1] в A[k..p + q — 1] else

Копировать ***B[i..p*** — 1] в ***A[k..p + q —*** *1****]***

Работа алгоритма со списком 8,3,2,9,7,1,5,4 показана на рис. 4.2.

Насколько эффективна сортировка слиянием? Положим для простоты, что п является степенью 2. Рекуррентное соотношение для количества выполняемых

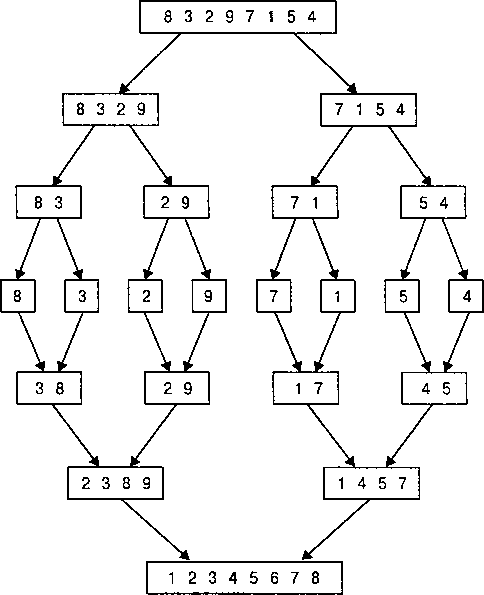


Рис. 4.2. Пример работы алгоритма сортировки слиянием

сравнений ключей С (п) выглядит следующим образом:

**С** (**П**) = 2**С** (п/2) + **Стегде** (п) **ДЛЯ** П > 1, **С** (1) = 0.

Давайте проанализируем величину Стегде (п), равную количеству сравнений ключей в процессе слияния. На каждом шаге выполняется в точности одно срав­нение, после которого общее количество элементов в двух сливаемых массивах уменьшается на 1. В худшем случае ни один из массивов не исчерпывается до самого конца, пока в другом массиве не останется только один элемент (например, наименьшие элементы поступают из двух массивов поочередно). Следовательно, в наихудшем случае Стегде (п) = п—1, и мы получаем рекуррентное соотношение

**Cworst** (n) — **^Cijjorst** **(^/^) + П 1 ДЛЯ 71 I> 1, Cworst (1) ~ 0[[29]](#footnote-29)**

Согласно основной теореме, CWOrst (п) € 0 (n log п) (почему?). Несложно най­ти и точное решение приведенного рекуррентного соотношения для п = 2к:

Cworst (п) = п log2 n-n + 1.

Количество сравнений ключей, выполняемых сортировкой слиянием, в худ­шем случае весьма близко к теоретическому минимуму1 количества сравнений для любого алгоритма сортировки, основанного на сравнениях. Основной недостаток сортировки слиянием — необходимость дополнительной памяти, количество ко­торой линейно пропорционально размеру входных данных. Сортировка слиянием возможна и без привлечения дополнительной памяти, но такая экономия памяти существенно ее усложняет и дает в результате значительно больший постоян­ный множитель в формуле для времени работы, так что эта версия сортировки слиянием представляет сугубо теоретический интерес.

Упражнения 4.1

1. а) Напишите псевдокод декомпозиционного алгоритма для поиска по­

зиции наибольшего элемента в массиве из п чисел.

б) Какими будут выходные данные алгоритма, если наибольшее значе­ние имеют несколько элементов?

в) Напишите и решите рекуррентное соотношение для количества срав­нений ключей, выполняемых алгоритмом.

г) Сравните созданный вами алгоритм с алгоритмом для решения ука­занной задачи, основанным на грубой силе.

1. а) Напишите псевдокод декомпозиционного алгоритма для поиска наи­

большего и наименьшего элементов в массиве из п чисел.

б) Напишите и решите рекуррентное соотношение для количества срав­нений ключей, выполняемых этим алгоритмом.

в) Сравните созданный вами алгоритм с алгоритмом для решения ука­занной задачи, основанным на грубой силе.

1. а) Напишите псевдокод декомпозиционного алгоритма для вычисле­

ния ап, где а > 0, а п — натуральное число.

б) Напишите и решите (для п — 2к) рекуррентное соотношение для количества умножений, выполняемых алгоритмом.

в) Сравните созданный вами алгоритм с алгоритмом для решения ука­занной задачи, основанным на грубой силе.

1. Рассматривая схему разработки и анализа алгоритмов в главе 2, мы упоминали, что в большинстве задач, возникающих при анализе алго­ритмов, основание логарифма не играет роли. Насколько это верно для основной теоремы, в которой имеются логарифмы?
2. Найдите порядок роста решений следующих рекуррентных соотно­шений.

а) Т (п) = 4Т (п/2) +п,Т (1) = 1.

б) Т (п) = 4Т (п/2) + п2, Т (1) = 1.

в) Т (п) = 4Т (п/2) + п3, Г (1) = 1.

1. Примените сортировку слиянием для упорядочения букв Е, X, А, М,

Р, L, Ев алфавитном порядке.

1. Устойчива ли сортировка слиянием?
2. а) Решите рекуррентное соотношение для количества сравнений клю­

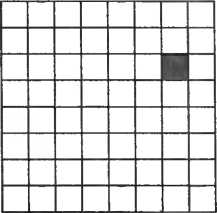
чей, выполняемых сортировкой слиянием в наихудшем случае. (Мож­но считать, что п = 2к.)

б) Напишите рекуррентное соотношение для количества сравнений ключей, выполняемых алгоритмом сортировки слиянием в наилуч­шем случае, и решите его при п — 2к.

в) Напишите рекуррентное соотношение для количества перемещений ключей, выполняемых описанной в разделе 4.1 версией алгоритма сортировки слиянием. Изменится ли класс эффективности алгорит­ма, если учесть количество перемещений ключей?

image71

1. Можно реализовать сортировку слиянием без рекурсии, начав со сли­яния соседних элементов данного массива, затем — отсортированных пар и т.д. Реализуйте такую восходящую версию алгоритма на своем любимом языке программирования.



1. Триомино. Триомино — элемент мозаичного заполнения в форме L, об­разованный тремя квадратами шахматной доски. Задача состоит в по­крытии триомино шахматной доски размером 2П х 2П с одной вырезан­ной в произвольном месте клеткой. Триомино должны покрывать все клетки, за исключением вырезанной, без пропусков и перекрытий.

ь

Разработайте декомпозиционный алгоритм для решения этой задачи.

1. Быстрая сортировка

Быстрая сортировка (quicksort) — еще один важный алгоритм сортировки, ос­нованный на методе декомпозиции. В отличие от сортировки слиянием, которая разделяет элементы массива в соответствии с их положением в массиве, быстрая сортировка разделяет элементы массива в соответствии с их значениями. Кон­кретно она выполняет перестановку элементов данного массива А [0..п — 1] для получения разбиения (partition), когда все элементы до некоторой позиции s не превышают элемента A [s], а элементы после позиции s не меньше A [s]:

А [0]... A [s — 1] A [s] A [s + 1]... А [п — 1].

' V ' ' V '

**Все элементы ^A[s] Все элементы ^A[s]**

Очевидно, что после разбиения A [s] находится в окончательной позиции в от­сортированном массиве, и мы можем сортировать два подмассива элементов до и после A [s] независимо (например, тем же самым методом).

Алгоритм Quicksort (A [L.r])

// Сортирует массив методом быстрой сортировки // Входные данные: Подмассив А[1..г] массива А[0..п — 1],

// определяемый начальным и конечным

И индексами I и г

// Выходные данные: Подмассив А[1..г], отсортированный // в неубывающем порядке

if *I < г*

***s \*— Partition(A[l..r]***) ***II s —*** позиция разбиения ***Quicksort(A[l..s*** — 1])

***Quicksort(A[s*** + l..r])

Разбиение массива А[0..п—1] и, в общем случае, его подмассива А[1..г] (0 ^ I < г ^ п — 1) можно выполнить следующим образом. Сначала выбира­ется элемент, относительно которого будет выполняться разбиение. В силу важ­ности роли этого элемента он называется опорным (pivot). Имеется ряд различ­ных стратегий для выбора опорного элемента (мы вернемся к этому вопросу при рассмотрении эффективности алгоритма). А пока что воспользуемся простейшей стратегией — выберем в качестве опорного первый элемент подмассива: р = А[1].

Имеется также ряд различных процедур перестановки элементов для разбие­ния. Здесь нами будет использоваться эффективный метод, основанный на двух проходах подмассива — слева направо и справа налево, и при каждом проходе эле­менты массива будут сравниваться с опорным элементом. Проход слева направо начинается со второго элемента. Поскольку мы хотим, чтобы элементы, меньшие опорного, находились в первой части подмассива, первый проход пропускает эле­менты, меньшие опорного, и останавливается, встретив первый элемент, который

не меньше опорного. Проход справа налево начинается с последнего элемента подмассива. Поскольку надо, чтобы все элементы, большие опорного, находились во второй части подмассива, при этом проходе опускаются все элементы, большие опорного, и проход останавливается, встретив первый элемент, не превышающий опорный.

В зависимости от того, пересеклись ли индексы сканирования, возможны три ситуации. Если индексы сканирования i и j не пересеклись, т.е. г < j, то мы просто обмениваем элементы А [г] и А [)] и продолжаем сканирование путем увеличения i и уменьшения j:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | ->/ |  | J<r- |  |
| р | все элементы й р |  |  |  | все элементы £ р |

Если при сканировании произошло пересечение индексов, т.е. i > j, то мы выполняем разбиение, обменивая опорный элемент с A [j]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | У<- | ->/ |  |
| Р | все элементы < р | \*Р | ^Р | все элементы £ р |

И, наконец, если при сканировании индексы остановились на одном элементе, т.е. г = j, то значение этого элемента должно быть равно р (почему?). Следова­тельно, мы имеем следующее разбиение массива:

|  |  |  |  |
| --- | --- | --- | --- |
| Р | все элементы й р | = Р | все элементы > р |

Последний случай можно объединить со случаем пересечения индексов (г > j), обменивая опорный элемент с A [j] при г ^ j.

Вот псевдокод, реализующий описанную процедуру разбиения.

**Алгоритм Partition (A** **[i..r])**

// Разбивает подмассив с использованием первого элемента // в качестве опорного

// Входные данные: подмассив A[L.r] массива А[0..п — 1],

// определяемый левым и правым

// индексами / и г (/ < г)

// Выходные данные: разбиение А[1..г]\ при этом позиция // разбиения возвращается как значение

// функции

Р \*- Щ i <— (; j <— г + 1 repeat repeat

г +— г + 1 until A[i) ^ р repeat **3 3 ~** 1

until A[j] ^ р вгшр(А[г], A[j]) until г ^ **j**

***swap(A[i\, A[j}) 11*** Отмена последнего обмена при ***г*** ^ ***j swap(A[l), A\j])* return** *j*

Заметим, что в таком псевдокоде возможен выход индекса г за пределы границ подмассива. Вместо того чтобы при каждом увеличении г проверять, не вышел ли индекс за пределы границы, можно добавить к массиву А [0..п — 1] “ограни­читель”, который не позволит индексу г выйти за пределы п. Заметим также, что выбор опорного элемента в конце подмассива делает ограничитель ненужным.

Пример сортировки массива при помощи алгоритма быстрой сортировки по­казан на рис. 4.3.

Обсуждение эффективности быстрой сортировки мы начнем с замечания о том, что количество сравнений ключей, выполненных до разбиения, достигает вели­чины п + 1, если индексы пересекаются, и гг, если совпадают (почему?). Если все разбиения оказываются посредине соответствующих подмассивов, реализуется наилучший случай. Количество сравнений ключей в наилучшем случае, Cbest (п)> удовлетворяет следующему рекуррентному соотношению:

Cbest (^0 = ^Cfoest (тг/2) + тг при ТЬ > 1, Cbest (1) — 0.

Согласно основной теореме, Cbest (п)е® (п log2 п). Точное решение для п = 2к дает Cbest Ы = п log2 п.

В наихудшем случае все разбиения оказываются такими, что один из подмас­сивов пуст, а размер второго на 1 меньше размера разбиваемого массива. Такая ситуация возникает, в частности, в возрастающем массиве, т.е. для входных дан­ных, для которых задача сортировки уже решена! В самом деле, если А [0..тг — 1] — строго возрастающий массив и мы используем в качестве опорного элемента А [0], то сканирование слева направо остановится на элементе А [1], в то время как ска­нирование справа налево дойдет до элемента А [0], что указывает на разбиение в позиции 0:

->/ у<-

|  |  |  |  |
| --- | --- | --- | --- |
| А[0] | А[ 1] |  | Д[п-1] |

Итак, после выполнения п + 1 сравнений и обмена опорного элемента А [0] с самим собой, выясняется, что теперь следует сортировать строго возрастаю­щий массив А [1..71 — 1]. Такая сортировка строго возрастающих массивов будет

**)**

7

7

7

7

7

7

2

2

2

У

2

3

3

3

3

3

3

3

/

3

/

3

5

5

5

5

5

5

2

2

2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 1  1 | 1  3 | 4 | /=0,г=3 |  | /=5,г=7 |
|  | У | / |  | S-1 |  | s=6 |

4

4

У

4

**/=0, г=7 s=4**

9

/

9

/

4

4

4

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| /=0,г=0 |  | /=2,г=3 |  | /=5, г=5 |  | 1=7, г-7 |
|  |  | s=2 |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 4 | /=2,г=1 |  | /=3,г=3 |
|  | 4 |  |  |  |

б)

|  |  |  |
| --- | --- | --- |
| 8 | /  9 | У  7 |
| 8 | /  7 | У  9 |
| 8 | У  7 | /  9 |
| 7 | 8 | 9 |
| 7 |  | 9 |

а)

Рис. 4.3. Пример работы алгоритма Quicksort, а) Преобразования массива (опорный эле­мент выделен полужирным шрифтом), б) Дерево рекурсивных вызовов Quicksort с указа­нием входных значений I и г и полученной позиции 5

продолжаться до последнего массива А [п — 2..п — 1]. Общее количество выпол­ненных сравнений ключей составляет

Cwarst (п) = (n + 1) + п н 1- 3 = + + ^ - 3 е 0 (п2) .

Таким образом, встает вопрос об эффективности алгоритма в среднем случае. Обозначим среднее количество сравнений ключей, выполняемых при быстрой сортировке случайного массива размером п, через Cavg (п). Считая, что разбиение

может выполняться в каждой позиции s (О ^ s ^ п—1) с одинаковой вероятностью 1 /п, получаем следующее рекуррентное соотношение:

**^ гг—1**

Cavg

[(гг + 1) + **Cavg** (s) + **Cavg** (n — 1 — в)] При **П >** 1,

П s=0 Cavg (0) = 0)

**Cavg** (1) = 0.

Хотя решение этого соотношения оказывается проще, чем можно было бы ожидать, тем не менее оно существенно сложнее, чем в наихудшем и наилучшем случае. Оставляем решение этого соотношения читателям в качестве упражнения, а здесь приведем только готовый ответ:

**Cavg (п)** « 2nlnn « 1.38nlog2 **П.**

Таким образом, алгоритм быстрой сортировки в среднем случае выполняет сравнений ключей всего на 38% больше, чем в наилучшем случае. Кроме того, внутренний цикл данного алгоритма настолько эффективен, что для случайных массивов он работает быстрее, чем сортировка слиянием (и пирамидальная сор­тировка — еще один алгоритм класса п log п, который будет рассмотрен в главе 6), оправдывая тем самым название, данное его разработчиком — выдающимся бри­танским кибернетиком Хоаром (С. A. R. Ноаге)[[30]](#footnote-30).

Исходя из важности быстрой сортировки, многие годы делались попытки улучшить базовый алгоритм. Среди прочих усовершенствований, открытых раз­личными исследователями, — улучшенные методы выбора опорного элемента (на­пример, разбиение на основе медианы трех элементов, когда в качестве опор­ного элемента используется медиана крайнего слева, справа и среднего элементов массива), переключение на более простую сортировку для малых подмассивов, удаление рекурсии (так называемая нерекурсивная быстрая сортировка). Согласно ведущему эксперту в области быстрой сортировки Седжвику (R. Sedgewick) [103], все вместе эти улучшения могут снизить время работы алгоритма на 20-25%.

Следует также отметить, что идея разбиения может оказаться полезной и в дру­гих приложениях, не только для сортировки. В частности, она лежит в основе быстрого алгоритма для важной задачи выбора, рассматриваемой в разделе 5.6.

1. Примените быструю сортировку для сортировки списка Е, X, А, М, Р, L, Е в алфавитном порядке. Изобразите дерево выполненных ре­курсивных вызовов.
2. Для процедуры разбиения, описанной в разделе 4.2:

а) Докажите, что если при сканировании индексы остановились на одном элементе, т.е. г = j, то значение этого элемента должно быть равно р.

б) Докажите, что при прекращении сканирования индексов j не может указывать на элемент, находящийся более чем в одной позиции слева от элемента, на который указывает г.

в) Почему наихудшим случаем является прекращение сканирования после того, как встретится элемент, равный опорному?

1. Устойчива ли быстрая сортировка?
2. Приведите пример массива из гг элементов, для которого необходим ограничитель, упомянутый в разделе. Каким должно быть его значе­ние? Поясните также, почему одного ограничителя достаточно для лю­бых входных данных.
3. Для описанной версии быстрой сортировки:

а) что собой представляет массив из одинаковых элементов — наихуд­ший случай, наилучший случай или ни то, ни другое?

б) что собой представляет массив из строго убывающих элементов — наихудший случай, наилучший случай или ни то, ни другое?

1. Предположим, что в алгоритме используется упомянутое в тексте раз­биение на основе медианы трех элементов.

а) Что в этом случае представляет собой массив из возрастающих элементов — наихудший случай, наилучший случай или ни то, ни другое?

б) Дайте ответ на тот же вопрос для массива из убывающих элементов.

1. Решите рекуррентное соотношение для среднего случая алгоритма быстрой сортировки.
2. Разработайте такой алгоритм перестановки элементов в заданном мас­сиве из п действительных чисел, чтобы все отрицательные числа пред­шествовали положительным. Алгоритм должен быть эффективен как в смысле времени работы, так и в смысле используемой памяти.
3. Реализуйте быструю сортировку на любом языке программирования, по вашему выбору. Выполните программу для различных входных дан-

ных, чтобы убедиться в справедливости теоретических предположений об эффективности алгоритма.

image73

10. Задача о гайках и болтах. У вас имеется п болтов различного размера и п соответствующих гаек. Вы можете сравнивать гайку и болт и опре­делять, подходят они друг к другу или гайка больше (или меньше) болта, но вы лишены возможности выполнить сравнение двух болтов или двух гаек между собой. Ваша задача состоит в том, чтобы разде­лить все болты и гайки по парам, в которых гайка по размеру будет соответствовать болту. Разработайте алгоритм для решения этой задачи за время 0 (nlogn) [94].

1. Бинарный поиск

Бинарный поиск представляет собой в высшей степени эффективный алгоритм для поиска в отсортированном массиве. Он работает путем сравнения искомого ключа К со средним элементом массива А [т]. Если они равны, алгоритм пре­кращает работу. В противном случае та же операция рекурсивно повторяется для первой половины массива, если К < А [т], и для второй, если К > А[т]:

***к***

г

Л[0]...Л[т-1] А[т] Л[т+1]...Л[л-1]

' V ' ' V '

Поиск в этой части, если Поиск в этой части, если

К<А[т) К>А[т]

0 1 23456789 10 11 12

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 14 | 27 | 31 | 39 | 42 | 55 | 70 | 74 | 81 | 85 | 93 | 98 |

В качестве примера найдем ключ К = 70, применяя алгоритм бинарного поиска к массиву

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 14 | 27 | 31 | 39 | 42 | 55 | 70 | 74 | 81 | 85 | 93 | 98 |

Итерации алгоритма показаны в следующей таблице:

Индекс Значение

Итерация 1 / т г

Итерация 2 I т г

Итерация 3 1,т г

Хотя ясно, что бинарный поиск основан на рекурсии, его очень легко реали­зовать как нерекурсивный алгоритм. Вот псевдокод нерекурсивной версии.

АЛГОРИТМ Binary Search (А [0..тг — 1], К)

Н Реализует нерекурсивный бинарный поиск // Входные данные: Массив А[0..п — 1], отсортированный

// в возрастающем порядке, и искомый ключ К

П Выходные данные: Индекс элемента массива, равного К,

// или —1, если такого элемента нет

I <— 0; г <— п — 1; while I ^ г do га <— [(/ + г)/2J if К = А[т\ return тп else if К < А[т\ г <— га — 1 else

I <— m + 1 return —1

Стандартный способ анализа эффективности бинарного поиска состоит в под­счете количества сравнений искомого ключа с элементами массива. Кроме того, из соображений простоты, мы будем считать, что используются тройные сравнения, т.е. что одно сравнение К и А [тп] позволяет определить, меньше ли АТ, больше или равно значению А [га].

Сколько же сравнений должен выполнить алгоритм при поиске в массиве из п элементов? Ответ, очевидно, зависит не только от гг, но и от конкретного экземпляра задачи. Найдем количество сравнений Cw (и) в наихудшем случае. В наихудший случай входят все массивы, которые не содержат искомого ключа (кроме них в этот разряд попадают и некоторые массивы, поиск в которых завер­шается успешно). Поскольку после одного сравнения алгоритм попадает в ту же ситуацию, что и до сравнения, только массив, в котором ведется поиск, стано­вится вдвое меньше, можно записать следующее рекуррентное соотношение для Cw (гг):

Cw (гг) = Cw (|n/2j) + 1 для гг > 1, Cw( 1) = 1. (4.2)

(Подумайте немного над тем, что значение п/2 следует округлять в меньшую сторону и что начальное условие именно таково, как указано в (4.2)).

Как уже говорилось в разделе 2.4, стандартный способ решения рекуррент­ных соотношений наподобие соотношения (4.2) состоит в том, что мы полагаем п — 2к и решаем получившееся соотношение путем обратной подстановки или при помощи какого-то другого метода. Читателю в качестве достаточно простого упражнения предлагается самостоятельно найти решение

(4.3)

Cw (2k^j = к + 1 = log2 п + 1.

Можно доказать, что решение (4.3) для п = 2к можно применить для получе­ния корректного решения для любого положительного целого п:

Cw (п) = [log2 nj + 1 = [log2 (п + 1)] . (4.4)

Давайте убедимся с помощью непосредственной подстановки, что функция Cw (п) = |\_1°§2 nJ + 1 действительно удовлетворяет уравнению (4.2) для любого положительного четного числа п. (Сделать то же самое для нечетных значений п предлагается в упражнении 4.3.3.) Если п положительно и четно, то п = 2г, где г > 0. Левая часть уравнения (4.2) при п = 2г имеет вид

Cw (п) = |bg2nj + 1 = [log2 2zJ + 1 = |k)g2 2 + log2zj + 1 =

= (1 + |\_log2 zj) + 1 = [log2 zj + 2.

Правая часть уравнения (4.2) при п = 2i имеет вид Cw (Ln/2J) + 1 — Cw (|\_2z/2J) + 1 = Cw (г) + 1 = ([log2 ij + 1) + 1 = U°S2 + 2-

Оба выражения одинаковы, что и доказывает наше утверждение.

Формула (4.4) заслуживает особого внимания. Во-первых, из нее следует, что эффективность бинарного поиска в наихудшем случае равна 0 (log п) (кстати го­воря, этот вывод можно сделать, воспользовавшись Основной теоремой, но в этом случае мы не получили бы значения постоянного множителя). Во-вторых, полу­ченный ответ именно таков, как и следовало ожидать: поскольку алгоритм на каж­дом шаге просто уменьшает размер оставшейся части массива вдвое, количество таких итераций для сведения массива из п элементов к единственному элементу должно быть примерно равно log2 п. В-третьих, как уже говорилось в разделе 2.1, логарифмическая функция растет очень медленно, так что ее значение остается небольшим даже при очень больших значениях п. В частности, согласно форму­ле (4.4), потребуется не более чем [log2 103J + 1 = 10 тройных сравнений для поиска элемента с заданным значением (или выяснения того, что такой элемент отсутствует) в массиве из 1000 элементов, и не более [log2 106J + 1 = 20 сравне­ний для поиска в массиве в миллион элементов!

Что можно сказать об эффективности бинарного поиска в среднем? Сложный анализ показывает, что среднее количество сравнений ключей в случае бинарного поиска только немного меньше такового в наихудшем случае:

Cavg (п) ^ k)g2 П.

(Более точные формулы для среднего количества сравнений в случае успешного и неудачного поиска — Cavg (п) « log2 п — 1 и С™д (п) « log2 (гг + 1), соответ­ственно.)

Если ограничиться только операцией сравнения ключей (см. раздел 10.2), то бинарный поиск является оптимальным алгоритмом, однако имеются алгоритмы поиска (см. интерполяционный поиск в разделе 5.6 и хеширование в разделе 7.3) с лучшим временем работы в среднем, а один из этих алгоритмов (хеширование) даже не требует, чтобы массив был отсортирован! Однако эти алгоритмы, кроме сравнения ключей, требуют некоторых специальных вычислений. И последнее: идея, лежащая в основе бинарного поиска, имеет ряд применений помимо поиска (см., например, [15]). Кроме того, она может быть использована для решения нелинейных уравнений (об этом мы поговорим в разделе 11.4).

Перед тем как завершить раздел, следует сделать еще одно замечание по по­воду бинарного поиска. Иногда бинарный поиск представляют квинтэссенцией алгоритма, основанного на декомпозиции. Такая интерпретация некорректна, по­скольку бинарный поиск на самом деле — весьма нетипичный случай метода декомпозиции. В самом деле, в соответствии с определением, данным в начале главы, метод декомпозиции делит задачу на несколько подзадач, каждая из кото­рых требует решения. В случае же бинарного поиска необходимо решить только одну из двух подзадач. Таким образом, если рассматривать бинарный поиск как алгоритм декомпозиции, то его следует трактовать как вырожденный случай это­го метода. В действительности бинарный поиск в большей степени соответствует классу алгоритмов деления пополам, которые мы рассмотрим в разделе 5.5. По­чему же в таком случае мы решили описать бинарный поиск в этой главе? Ча­стично — по традиции, частично — потому что плохой пример иногда в состоянии показать то, что не в состоянии показать хороший пример.

Упражнения 4.3

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 14 | 27 | 31 | 39 | 42 | 55 | 70 | 74 | 81 | 85 | 93 | 98 |

а) Чему равно максимальное количество сравнений при бинарном по­иске в этом массиве?

б) Перечислите ключи массива, при поиске которых будет выполняться максимальное количество сравнений.

в) Найдите среднее количество сравнений, выполняемых при успеш­ном бинарном поиске в этом массиве; предполагается, что вероят­ности поиска каждого ключа одинаковы.

г) Найдите среднее количество сравнений, выполняемых при неудач­ном бинарном поиске в этом массиве; предполагается, что вероят­ность поиска для каждого из 14 интервалов, образуемых элементами массива, одинакова.

1. Решите рекуррентное уравнение Cw (n) = Cw ([п/2J) + 1 при п > 1, Cw (1) = 1 для п — 2к методом обратной подстановки.
2. а) Докажите равенство

[log 2 n\+l= [log2 (П + I)] При П ^ 1.

б) Докажите, что функция Cw (n) = [log2 nj + 1 является решением рекуррентного уравнения (4.2) для положительных нечетных п.

1. Оцените, во сколько раз быстрее в среднем будет выполняться успеш­ный бинарный поиск в отсортированном массиве из 100000 элементов по сравнению с последовательным поиском.
2. Последовательный поиск дает практически одинаковую эффективность при поиске в массиве и в связанном списке. Справедливо ли аналогич­ное утверждение для бинарного поиска? (Естественно, мы предпола­гаем, что список отсортирован.)
3. Как с помощью бинарного поиска найти диапазон, т.е. все элементы отсортированного массива, которые находятся между заданными зна­чениями L и U включительно (L < £/)? Какова эффективность этого алгоритма в наихудшем случае?
4. Напишите псевдокод рекурсивной версии бинарного поиска.
5. Разработайте версию бинарного поиска, где бы использовались обыч­ные сравнения, такие как < и =. Реализуйте алгоритм на любом языке программирования и тщательно отладьте его (обычно при написании таких программ образуется повышенное количество ошибок).

image74

1. Проанализируйте временную эффективность версии бинарного поиска с обычными сравнениями, предложенной в упражнении 8.
2. Перед вами разложены 42 рисунка в семь рядов, по 6 рисунков в каж­дом. Необходимо определить загаданный партнером рисунок, задавая вопросы, на которые можно получить ответ “да” или “нет”. Ваша за­дача — предложить максимально эффективный алгоритм решения этой задачи и указать максимальное количество вопросов, которое потребу­ется задать, чтобы гарантированно определить рисунок.
3. Обход бинарного дерева

В этом разделе мы рассмотрим, каким образом метод декомпозиции может быть применен к бинарным деревьям. Бинарное дерево Т определяется как ко­нечное множество узлов, которое может быть либо пустым, либо состоит из корня

и двух непересекающихся бинарных деревьев Ть и Tr, именуемых, соответствен­но, левым и правым поддеревьями корня. Обычно мы говорим о бинарном дереве как о частном случае упорядоченного дерева (рис. 4.4). (Соответствующее опре­деление бинарного дерева было дано в разделе 1.4.)

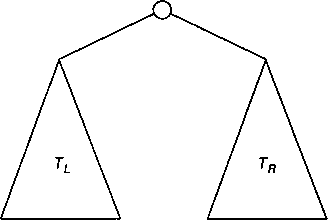


Рис. 4.4. Стандартное представле­ние бинарного дерева

Поскольку по определению бинарное дерево делится на две меньшие структу­ры такого же типа — левое поддерево и правое поддерево, многие задачи, связан­ные с бинарными деревьями, могут решаться при помощи метода декомпозиции. В качестве примера рассмотрим рекурсивный алгоритм определения высоты би­нарного дерева. Вспомним, что высота дерева определяется как длина самого длинного пути от корня к листу. Следовательно, ее можно вычислить, выбрав максимальное значение среди высот левого и правого поддеревьев корня и при­бавив к нему 1 (чтобы учесть дополнительный уровень корня). Заметим также, что удобно определить высоту пустого дерева равной —1. Итак, мы получили следующий рекурсивный алгоритм:

Алгоритм Height (Г)

// Рекурсивное вычисление высоты бинарного дерева // Входные данные: Бинарное дерево Т

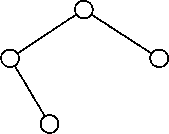
II Выходные данные: Высота бинарного дерева Т ИТ = 0 return —1 else

**return та*x{Height{TL), НегдЫ(Тя)}*** *+* **1**

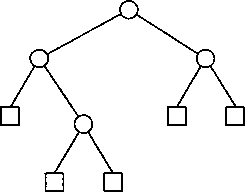
Размер экземпляра данной задачи измеряется количеством узлов п (Т) бинар­ного дерева Т. Очевидно, что количество сравнений, выполняемых для вычисле­ния максимального из двух чисел, и количество сложений А (п (Т)), выполняемое алгоритмом, одинаково. Для А (п (Г)) можно записать следующее рекуррентное уравнение:

А (п (Г)) = А (п (TL)) + А(п {TR)) + 1 при п (Г) > О, А (0) = 0.

Прежде чем решать уравнение (можете ли вы сказать, каким будет его реше­ние?), заметим, что сложение — не самая часто выполняемая операция в данном алгоритме. А какая же? Проверка, не является ли бинарное дерево пустым (кста­ти, это весьма типично для алгоритмов, работающих с бинарными деревьями). Например, для пустого дерева сравнение Т = 0 выполняется один раз, в то время как сложение в этом случае не выполняется вовсе. Для дерева из одного узла количество сравнений и сложений равны, соответственно, трем и одному.



При анализе алгоритмов, работающих с бинарными деревьями, зачастую по­могает изображение расширения дерева, в котором пустые поддеревья заменены специальными узлами. Такие дополнительные узлы, изображенные на рис. 4.5 квадратами, называются внешними (external); исходные узлы (изображенные в ви­де кругов) называются внутренними (internal). По определению расширение пу­стого бинарного дерева представляет собой единственный внешний узел.



а) б)

Рис. 4.5. а) Бинарное дерево, б) Его расширение. Внут­ренние узлы представлены кругами, внешние — квад­ратами

Легко увидеть, что алгоритм поиска высоты бинарного дерева выполняет ров­но по одному сложению для каждого внутреннего узла расширенного дерева, и по одной проверке на пустоту дерева для каждого внутреннего и внешнего узла. Таким образом, для определения эффективности алгоритма надо знать, сколько внешних узлов может иметь расширенное дерево с п внутренними узлами. Рас­смотрев рис. 4.5 и несколько аналогичных примеров расширенных деревьев, легко выдвинуть гипотезу (которую несложно доказать), что количество внешних узлов х всегда на один больше количества внутренних узлов п:

х = п + 1. (4.5)

Докажем это равенство по индукции по количеству внутренних узлов п > 0. Базис индукции справедлив, поскольку при п = 0 мы имеем пустое дерево с одним внешним узлом по определению. Положим в общем случае, что х = к + 1 для любого расширенного дерева с 0 ^ к < п внутренними узлами. Пусть Т —

расширенное бинарное дерево с п внутренними и х внешними узлами, пь и хь — количество, соответственно, внутренних и внешних узлов в левом поддереве Т, а пд и xr — количество внутренних и внешних узлов в правом поддереве Т, соответственно. Поскольку п > 0, бинарное дерево Т имеет корень, который является внутренним узлом, и, следовательно, п = hl + пд +1. Полагая гипотезу индукции справедливой для левого и правого поддеревьев, находим количество внешних узлов дерева Т:

**X = XL + Xr = (71l + 1) + (TlR + 1) = (til + nR + 1) + 1 = 71 + I,**

что и завершает доказательство.

Возвращаясь к алгоритму Height, находим, что количество сравнений, выпол­няемых для проверки пустоты дерева, равно

С (п) = п + ж = 2п + 1, а количество сложений равно

А (п) = п.

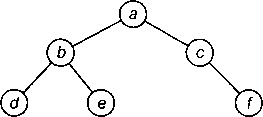
Наиболее важные алгоритмы бинарных деревьев — это три классических обхо­да бинарных деревьев — в прямом порядке (preorder traversal), симметричный, или центрированный (inorder traversal), и в обратном порядке (postorder traversal). При всех указанных обходах узлы дерева посещаются рекурсивно, т.е. посещаются корень и его левое и правое поддеревья, и отличие обходов только в последова­тельности посещений.

* При обходе в прямом порядке сначала посещается корень дерева, а за­тем левое и правое поддеревья (в указанном порядке).
* При симметричном обходе корень посещается после левого поддере­ва, но перед посещением правого.
* При обходе в обратном порядке корень посещается после левого и правого поддеревьев (в указанном порядке).

Псевдокод всех описанных обходов бинарного дерева очень прост (эти обхо­ды к тому же рассматриваются в любом учебнике по структурам данных). Что же касается анализа эффективности, то он идентичен только что проведенному, поскольку рекурсивный вызов выполняется для каждого узла расширенного би­нарного дерева.

Напоследок следует заметить, что не все задачи, связанные с бинарными дере­вьями, требуют обхода обоих поддеревьев. Например, поиск и вставка в бинарное дерево поиска требуют обработки только одного из двух поддеревьев. Следова­тельно, они должны рассматриваться не как приложения метода декомпозиции, а как примеры метода уменьшения на переменную величину, рассматриваемого в разделе 5.6.

1. Разработайте декомпозиционный алгоритм для вычисления количества уровней в бинарном дереве (в частности, алгоритм должен возвра­щать 0 для пустого дерева и 1 для дерева с одним узлом). К какому классу эффективности относится алгоритм?
2. Обойдите приведенное ниже дерево в прямом, центрированном и об­ратном порядке. Приведите содержимое стека обхода в процессе рабо­ты алгоритмов.



1. Напишите псевдокод для одного из классических алгоритмов обхода бинарного дерева (в прямом, центрированном или обратном порядке). Считая, что рассматриваемый алгоритм рекурсивен, найдите количе­ство выполняемых рекурсивных вызовов.
2. Чему равно наибольшее количество узлов, которые могут одновремен­но находиться в стеке при симметричном обходе бинарного дерева? Опишите вид деревьев, для которых этот максимум достигается на практике.
3. В каком порядке — прямом, центрированном, обратном или никаком из перечисленных — алгоритм поиска высоты дерева, описанный в данном разделе, вычисляет высоты всех поддеревьев входного дерева?
4. Какой из трех классических алгоритмов обхода дерева приводит к от­сортированному списку при применении к бинарному дереву поиска? Докажите это свойство указанного вами алгоритма обхода.
5. В корневом упорядоченном дереве каждый внутренний узел имеет по­ложительное количество упорядоченных потомков, каждый из которых служит корнем упорядоченного поддерева. Разработайте алгоритм для вычисления высоты корневого упорядоченного дерева. (Предполагает­ся, что наименьшее корневое упорядоченное дерево состоит из одного узла, а его высота равна 0.)
6. Приведенный далее алгоритм предназначен для вычисления количе­ства листьев бинарного дерева.

АЛГОРИТМ Leaf Counter (Т)

// Рекурсивно вычисляет количество листьев бинарного дерева // Входные данные: Бинарное дерево Т

**II** Выходные данные: Количество листьев в **Т** if **Т =** 0 return О else

**return *Leaf Counter (Tl)*** + ***Leaf Counter*** **(Tr)**

Корректен ли приведенный алгоритм? Если да, докажите это; если нет, внесите необходимые изменения.

1. Длина внутреннего пути (internal path length) I расширенного бинар­ного дерева определяется как сумма длин путей от корня к внутреннему узлу, взятая по всем внутренним узлам. Аналогично, длина внешнего пути (external path length) Е расширенного бинарного дерева опре­деляется как сумма длин путей от корня к внешнему узлу, взятая по всем внешним узлам. Докажите, что Е = I + 2п, где п — количество внутренних узлов в дереве.
2. Напишите программу для вычисления длины внутреннего пути бинар­ного дерева поиска. Воспользуйтесь ею для эмпирического исследо­вания среднего количества сравнения ключей при поиске в случайно сгенерированном бинарном дереве поиска.
3. Умножение больших целых чисел и алгоритм умножения матриц Штрассена

В этом разделе мы рассмотрим два удивительных алгоритма для кажущих­ся простыми задач: умножения двух чисел и умножения двух квадратных мат­риц. Оба они пытаются уменьшить количество выполняемых умножений ценой небольшого увеличения количества сложений; оба используют метод декомпо­зиции.

Умножение больших целых чисел

В некоторых приложениях, особенно в современной криптологии, требуется работать с целыми числами, длина которых превышает 100 десятичных цифр. Понятно, что такое число слишком велико, чтобы разместить его в одном слове

современного компьютера, так что для работы с ним требуются специальные подходы. В этом разделе мы рассмотрим интересный алгоритм для перемножения таких больших чисел. Очевидно, что если мы используем классический алгоритм умножения двух n-значных чисел в столбик, то каждая из п цифр одного числа будет умножаться на каждую из п цифр второго числа, что в результате дает нам п2 умножений цифр. (Если у одного числа количество цифр меньше, чем у второго, впереди к нему можно дописать некоторое количество нулей, чтобы длина чисел стала одинаковой.) Казалось бы, невозможно разработать алгоритм, выполняющий менее п2 умножений цифр, но это не так. Метод декомпозиции способен творить чудеса.

Для демонстрации идеи, лежащей в основе алгоритма, рассмотрим умножение двух двузначных чисел, скажем, 23 и 14. Эти числа можно представить следую­щим образом:

23 = 2 • 101 + 3 • 10° и 14 = 1 • 101 + 4 • 10°.

Перемножим их:

23 • 14 = (2 • 101 + 3 • 10°) • (1 • 101 + 4 • 10°) =

= (2 • 1) • 102 + (3 • 1 + 2 • 4) • 101 + (3 • 4) • 10°.

Приведенная формула, конечно, дает верный ответ — 322, но в ней использу­ются те же четыре умножения цифр, что и при умножении в столбик. К счастью, можно вычислить средний член при помощи только одного умножения, восполь­зовавшись тем, что нам известны произведения (2 • 1) и (3 • 4), которые все равно будут вычислены:

3 • 1 + 2 • 4 = (2 + 3) • (1 + 4) - (2 • 1) - (3 • 4).

Конечно, в числах, которые мы использовали, нет ничего особенного. Для любой пары двузначных чисел а = aiao и b = bibo их произведение с можно вычислить по формуле

с = а • b = С2 • Ю2 + ci • 101 + со,

где

С2 = ai • bi — произведение первых цифр; со = ао ■ Ьо — произведение вторых цифр;

а = (ai + ao) • (bi + Ьо) — (с2 + со) — произведение суммы цифр а на Сумму ЦИфр Ь МИНуС Сумма С2 и Cq.

Теперь применим этот трюк к произведению двух n-значных чисел а и Ь, где п — положительное четное число. Разделим числа пополам — в конце концов, мы же предупреждали, что будем использовать метод декомпозиции. Используем для первой половины цифр а запись а\, а для второй — а о. Для половин числа Ь будут использованы соответственно обозначения bi и Ьо- При использовании такой записи из а = aiao вытекает а = а\ • 10п/2 + ао, а из Ь = bibo -b = b\- 10n/2 + Ьо-

Таким образом, воспользовавшись описанным ранее методом, для произведения

чисел а и Ь получим

с = а • Ь = (ai • 10п/2 + ао) • (&1 • Ю"/2 + Ьо) =

= (ai • bi) • 10" + (ai • Ьо + do ■ bi) • 10n/2 + (ао • Ьо) =

= с2 • 10" + cilOn//2 + со,

где

С2 = а\-Ь\ — произведение первых половин;

со = ао • Ьо — произведение вторых половин;

ci = (ai + ао) • (bi + Ьо) — (с2 + со) — произведение суммы половин а на сумму половин Ь минус сумма с2 и со- Если п/2 четно, тот же способ можно применить и для вычисления произве­дений с2, ci и со. Следовательно, если п представляет собой степень 2, получаем рекурсивный алгоритм для вычисления произведения двух n-значных целых чи­сел. В чистом виде рекурсия завершается, когда п становится равным единице, но ее можно прекратить и раньше — когда п станет достаточно малым для непо­средственного перемножения чисел такого размера.

Сколько умножений цифр выполняется в данном алгоритме? Поскольку пере­множение n-значных чисел требует трех умножений n/2-значных чисел, рекур­рентное соотношение для количества умножений М (п) имеет вид

М(п) = ЗМ(п/2) при n > 1, М (1) = 1.

Решение этого уравнения методом обратной подстановки для п = 2к дает М (2fc) = 3М (2fc\_1) = 3 [зМ (2fe“2)] = 32М (2fc“2) =

= • • ■ = 3\*М (2fc\_i) = - • • = зкМ (2fc\_fc) = 3fe.

Поскольку к = log2 п,

м (п) = 3log2n = nlog23 « n1-585.

(На последнем шаге преобразований мы воспользовались тем свойством логариф­мов, что alogbC = clogba.)

Не следует забывать, что для чисел средней длины этот алгоритм, вероятно, будет работать дольше, чем классический. Брассард (Brassard) и Брейтли (Bratley) ([22], стр.70-71) пишут, что в их экспериментах алгоритм декомпозиции превос­ходил метод умножения в столбик только для чисел длиной более 600 цифр. Если вы работаете на Java, C++ или Smalltalk, то учтите, что для этих языков имеются специальные классы для работы с большими целыми числами.

Алгоритм Штрассена для умножения матриц

где

Теперь, когда мы продемонстрировали, как применение метода декомпозиции позволяет снизить количество умножений цифр при перемножении больших це­лых чисел, вас не должен удивлять подобный эффект при перемножении матриц. Такой алгоритм был опубликован в 1969 году Штрассеном (V. Strassen) [113]. Основная идея, лежащая в основе этого алгоритма, заключается в открытии, что произведение С двух матриц А и В размером 2x2 можно вычислить с помощью только 7, а не 8 умножений, которые необходимы при использовании алгоритма грубой силы, описанного в примере 3 раздела 2.3. Этого можно достичь, используя следующие формулы:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5  О  8 |  | ООО а01 |  | сг  о  о  СГ  О |
| Сю Си |  | аю ац |  | Ью Ьц |
|  |  | ТП\ + 7714 | - т5 + 7777 | |

**7712 +** тА 771 \ **+** 77ls ~ т2 **+** ГПб

|  |  |  |  |
| --- | --- | --- | --- |
| mi = | в  +  о  о | СГ  О  о | + Ьп) |
| m2 = | (аю + ац) | сг  о  о |  |
| ms = | аоо • (boi — | Ьп) |  |
| 777,4 = | ац • (Ью — | боо) |  |
| Щъ = | (аоо + ooi) | ■Ьп |  |
| m6 = | (аю — аоо) | СГ  О  о | + &0l) |
| 7777 = | (aoi — ац) | • (&ю | + hi) |

Таким образом, для умножения двух матриц размером 2x2 алгоритм Штрассе­на выполняет семь умножений и 18 сложений/вычитаний, в то время как алгоритм на основе грубой силы требует восьми умножений и четырех сложений. Приве­денное выше вовсе не означает, что вы должны использовать алгоритм Штрассена для умножения матриц размером 2x2 — его важность в асимптотическом пре­восходстве над алгоритмом грубой силы, когда порядок перемножаемых матриц стремится к бесконечности.

Соо

Си

-<4оо

***Ао\***

Сю

Си

***Aw***

***А***

и

Пусть А и В — две матрицы размером пхп, где п — степень двойки. (Если п не  
является степенью двойки, можно добавить к матрицам необходимое количество  
строк и столбцов, заполненных нулями.) Матрицы А и В и их произведение С  
можно разделить на четыре подматрицы размером п/2 х п/2 каждую следующим

образом:

|  |  |
| --- | --- |
| Воо | Boi |
| Вю | Вп . |

Нетрудно убедиться, что для получения корректного произведения эти под­матрицы можно рассматривать как числа. Например, Сад можно вычислить как Аоо • Boo + Aqi ■ Вю или как М\ + М4 — М5 + М7, где М\, М4, М5 и М7 вычис­ляются по формулам Штрассена, в которых числа заменены соответствующими подматрицами. Алгоритм Штрассена состоит в том, что требуемые семь произве­дений подматриц размером п/2 х п/2 вычисляются рекурсивно с использованием описанного метода.

Давайте определим асимптотическую эффективность данного алгоритма. Ес­ли М (п) — количество умножений, выполняемых алгоритмом Штрассена для умножения двух матриц размером пхп (где п — степень двойки), то мы получим следующее рекуррентное соотношение для М (п):

М (п) = 7М (п/2) при n > 1, М (1) = 1.

Поскольку п = 2к,

М (2fc) = 7М (2fe\_1) = 7 [7М (2fe“2)] = 72М (2fc~2)

= ГМ (2fc\_i) = • • • = 7кМ (2к~кЛ) = 7к.

Подставляя к = log2 п, получаем:

м (n) = 7log2п = nlog27 « п2-807,

что меньше, чем п3, необходимое для алгоритма на основе грубой силы.

Поскольку экономия количества умножений достигается ценой большего коли­чества сложений, следует рассмотреть количество сложений А (п), выполняемых алгоритмом Штрассена. Для умножения двух матриц порядка п > 1 алгорит­му требуется семь умножений и 18 сложений матриц размером п/2 х п/2; при п — 1 сложений вообще не требуется, поскольку в этом случае задача вырожда­ется в перемножение двух чисел. Все это приводит к следующему рекуррентному уравнению:

А (п) = 7А (п/2) + 18 (п/2)2 при n > 1, А (1) = 0.

Хотя можно получить точное решение этого уравнения (см. упражнение 4.5.8), в настоящий момент нас интересует только порядок роста решения этого со­отношения. Согласно Основной теореме, приведенной в начале главы, А{п) Е Е © (nlog2 7). Другими словами, количество сложений имеет тот же порядок роста, что и количество умножений. Таким образом, можно сделать вывод о том, что эф­фективность алгоритма Штрассена — © (nlog2?), что лучше, чем эффективность алгоритма на основе грубой силы 0 (п3).

После Штрассена открыт ряд других алгоритмов для перемножения матриц действительных чисел размером гг х п за время О (па) с постоянно уменьша­ющимся значением показателя а. Наиболее быстрый алгоритм был предложен Куперсмитом (Coopersmith) и Виноградом (Winograd) [31] — его эффективность равна © (гг2 376). Значение показателя уменьшается за счет увеличения сложности алгоритмов. Из-за большой величины постоянного множителя в формуле време­ни работы алгоритма все предложенные алгоритмы умножения матриц не имеют практической ценности, но представляют большой теоретический интерес. С од­ной стороны, новые алгоритмы приближаются к теоретическому пределу, который равен п[[31]](#footnote-31) умножений, но точная величина “зазора” между теоретическим пределом и наилучшим из возможных алгоритмом остается неизвестна. С другой стороны, умножение матриц представляет собой вычислительный эквивалент некоторых других важных задач, например, таких, как решение систем линейных уравнений.

Упражнения 4.5

1. Проверьте справедливость формул, лежащих в основе алгоритма Штрассена для перемножения матриц размером 2x2.
2. Примените алгоритм Штрассена для вычисления

|  |  |  |
| --- | --- | --- |
| 1  г-Ч  СМ  о  т—1 1 |  | 1  т—1  О  т—1 ,° |
| 4 110 |  | 2 10 4 |
| 0 13 0 |  | 2 0 11 |
| 5 0 2 1 |  | 13 5 0 |

Рекурсию при т = 2 следует прекратить, т.е. перемножение матриц размером 2x2 выполняется с помощью алгоритма на основе грубой силы.

1. Решите рекуррентное уравнение для количества сложений, выполняе­мых алгоритмом Штрассена (в предположении, что п является степе­нью 2).
2. Пан (V. Pan) [85] открыл метод декомпозиции для умножения матриц, основанный на перемножении матриц размером 70 х 70 при помощи 143 640 умножений чисел. Найдите асимптотическую эффективность алгоритма Пана (сложения можете игнорировать) и сравните ее с эф­фективностью алгоритма Штрассена.
3. Практические реализации алгоритма Штрассена обычно переключа­ются на использование алгоритма, основанного на грубой силе, когда размеры матриц становятся меньше некоторой “точки пересечения”. Проведите вычислительный эксперимент и определите точку пересе­чения для вашего компьютера.
4. Решение задач о паре ближайших точек и о выпуклой оболочке методом декомпозиции

В разделе 3.3 мы рассматривали основанные на грубой силе алгоритмы для ре­шения двух классических задач вычислительной геометрии: задачу о паре ближай­ших точек и о выпуклой оболочке. Вы видели, что двумерные версии этих задач могут быть решены методом грубой силы за время 0 (п2) и О (п3) соответствен­но. Здесь мы рассмотрим более сложные и асимптотически более эффективные алгоритмы решения указанных задач, основанные на методе декомпозиции.

Задача о паре ближайших точек

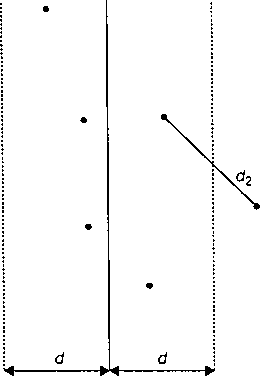
Пусть Pi = (xi, у\),..., Рп = (хп, уп) — множество 5, состоящее из п точек на плоскости (для простоты будем считать, что п является степенью двойки). Без потери общности можно считать, что точки упорядочены в соответствии с воз­растанием их координаты х (если это не так, то их можно упорядочить за время О {п log гг), например, при помощи сортировки слиянием). Точки можно разделить на два подмножества S\ и 52, состоящие из гг/2 точек каждое, проведя вертикаль­ную линию х = с так, чтобы слева и справа от нее оказалось по гг/2 точек (один из способов поиска значения константы с состоит в использовании медианы /х координат х точек).

Следуя методу декомпозиции можно рекурсивно найти пары ближайших точек в левом подмножестве 5i и в правом подмножестве 52- Пусть d\ и d<i -г- наимень­шие расстояния между парами точек в подмножествах 5i и 52, соответственно. Обозначим d = min{di,d2}- К сожалению, d не обязательно представляет собой наименьшее расстояние между парами точек в 5i и 5г, поскольку пара точек с минимальным расстоянием между ними может лежать по разные стороны от разделяющей линии. Таким образом, этап комбинирования должен включать рас­смотрение таких точек. Очевидно, что можно ограничиться рассмотрением точек, попадающих в вертикальную полосу шириной 2d, поскольку расстояние между любыми другими парами точек по разные стороны от разделяющей линии заве­домо превосходит расстояние d (рис. 4.6а). Пусть С\ и С2 — подмножества точек в левой и правой части полосы, соответственно.

Теперь для каждой точки Р (х, у) из С\ мы должны рассмотреть точки в С2, которые могут оказаться от точки Р на расстоянии, меньшем d. Совершенно очевидно, что координаты у таких точек не могут выходить за пределы интервала [у — d, у + d]. Наиболее важным является то, что таких точек может быть не более шести, поскольку любая пара точек в С2 находится как минимум на расстоянии d друг от друга. (Вспомним, что d ^ где di — наименьшее расстояние между точками справа от разделяющей линии.) Наихудший случай проиллюстрирован на рис. 4.66.

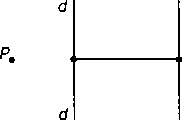
Второе важное наблюдение заключается в том, что можно поддерживать спис­ки точек в С\ и С2 отсортированными в порядке возрастания их координат у (эти списки можно рассматривать как проекции точек на разделяющую линию). Кро­ме того, такой порядок можно поддерживать не путем пересортировки точек при каждой итерации, а слиянием двух ранее отсортированных списков (см. алгоритм Merge из раздела 4.1). Можно рассматривать точки Сь в то время как указатель в список С2 будет выполнять осцилляции в пределах отрезка 2d, выбирая шесть кандидатов, для которых будут вычисляться расстояния до текущей точки Р из списка С\. Время М (гг), необходимое для “слияния” решений меньших подзадач, равно О (гг).

***х = с***



а)

***d***



б)

Рис. 4.6. а) Идея метода декомпозиции для решения задачи о па­ре ближайших точек, б) Шесть точек, которые может потребо­ваться рассмотреть для точки Р

В результате рекуррентное соотношение для времени работы Т (п) описанного алгоритма над п предварительно отсортированными точками имеет вид:

Т (п) = 2Т (п/2) + М (п).

Применяя О-версию Основной теоремы (для а = 2, b — 2nd=l, получаем, что Т(п) € O(nlogn). Возможная необходимость предварительной сортировки точек не меняет класс эффективности алгоритма, поскольку может быть выпол­нена за время О (nlogn). Мы получили алгоритм с наивысшим классом эффек­тивности для данной задачи, так как было доказано, что любой алгоритм для ее решения принадлежит к Г2 (nlogn) (см. [90], стр. 188).

Задача о выпуклой оболочке

Для решения задачи о выпуклой оболочке, в которой требуется найти мини­мальный выпуклый многоугольник, содержащий п точек на плоскости, на самом деле имеется несколько алгоритмов, основанных на методе декомпозиции. Мы рассмотрим простейший из них, который иногда называют быстрой оболочкой (quickhull) по аналогии с быстрой сортировкой, работу которой он напоминает.

Пусть Р\ = (x'l, yi),.... Рп (хп. уп) — множество из п точек на плоскости. Мы считаем, что эти точки отсортированы в возрастающем порядке по координате х, а в случае совпадения координаты х — в возрастающем порядке по координате у. Нетрудно доказать геометрически очевидный факт, что точки, крайние слева Pi) и справа Рп), должны принадлежать множеству вершин выпуклой оболочки (рис. 4.7). Пусть Р\Рп — прямая линия, проведенная от точки Р\ к точке Рп. Эта линия делит множество точек на два подмножества: Si — множество точек, лежа­щих слева от линии или на ней, и 5г — множество точек, которые лежат справа от линии. (Мы говорим, что точка рз находится слева от линии pipL проведенной от точки pi к точке р2, если точки Р1Р2Р3 образуют цикл против часовой стрелки. Позже мы укажем аналитический способ проверки этого условия, основанный на определении знака детерминанта, образуемого координатами этих трех точек.)

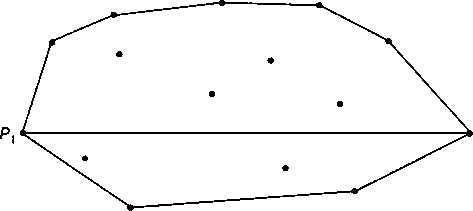


Рис. 4.7. Верхняя и нижняя оболочки множества точек

Выпуклая оболочка множества Si состоит из отрезка с конечными точками Pi и Рп и верхней границы, представляющей собой ломаную линию, состоящую из сторон многоугольника, т.е. из последовательности отрезков, соединяющих некоторые точки Si. Верхняя граница называется верхней оболочкой (upper hull). Аналогично, ломаная линия, служащая нижней границей выпуклой оболочки S2, называется нижней оболочкой (lower hull). Тот факт, что выпуклая оболочка всего множества S состоит из верхней и нижней оболочек, которые могут быть

построены независимо одним и тем же способом, оказывается очень важным и используется в ряде алгоритмов решения данной задачи.

Для конкретности рассмотрим, как алгоритм быстрой оболочки строит верх­нюю оболочку; нижняя оболочка строится аналогично. Во-первых, алгоритм опре­деляет точку Ртах в Si, наиболее удаленную от линии Р\Рп (рис. 4.8). Если таких точек несколько, в качестве Ртах выбирается та, для которой угол ZPmaxPiPn максимален. (Заметим, что если мы будем рассматривать все треугольники, две вершины которых — Pi и Рп, а третья — некоторая точка из множества Si, то при выборе в качестве третьей вершины Ртах площадь треугольника будет максималь­на.) После этого алгоритм определяет, какие точки множества Si находятся слева

ОТ ЛИНИИ Pi Ртах i ЭТИ ТОЧКИ ВМвСТе С точками Pi И Ртах образуют множество Si д.

► ’

Точки Si, лежащие слева от линии PmaxPn, вместе с точками Ртах и Рп образуют

множество Si^. Нетрудно доказать, что не существует точек, лежащих слева от обеих линий. Точки внутри треугольника Д PiPmaxPn можно в дальнейшем не рассматривать. Таким образом, алгоритм может рекурсивно продолжать постро­ение верхней оболочки Si,i и Si,2, а затем просто соединить их для получения верхней оболочки множества Si.

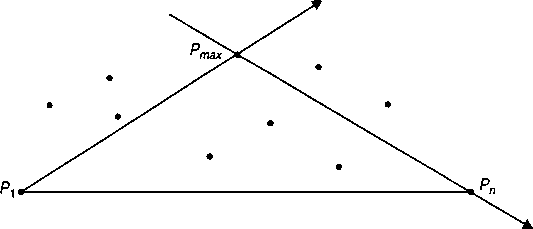


Рис. 4.8. Идея алгоритма быстрой оболочки

ад У1 1

х2 У2 1

ад Уз 1

Теперь выясним, как реализуются геометрические операции алгоритма. К сча­стью, мы можем воспользоваться следующим фактом из аналитической геомет­рии: если pi = (ад, yi), р2 = (ад, у2) ир3 = (ад, у3) — три произвольные точки на плоскости, то площадь треугольника Д р\р2рг равна половине абсолютного значения детерминанта

= адуг + адуi + адуз - аду2 - x2yi - аду3,

а само это выражение положительно тогда и только тогда, когда точкарз = (ад, у3) находится слева от линии pip2. Воспользовавшись этой формулой, можно за фик­сированное время определить, лежит ли точка слева от линии, образованной двумя другими точками, и найти расстояние от точки до этой линии.

Алгоритм быстрой оболочки имеет ту же эффективность, что и алгоритм быст­рой сортировки: 0 (n log гг) в среднем, и 0 (гг[[32]](#footnote-32)) — в наихудшем случае. Хотя это и выше эффективности алгоритма, основанного на грубой силе (который был рассмотрен нами в разделе 3.3), имеются, как уже упоминалось, и более слож­ные алгоритмы для решения задачи о выпуклой оболочке, время работы которых в наихудшем случае составляет 0 (гг log гг) (см., например, [90]).

Упражнения 4.6

1. Поясните, как найти точку Ртах в алгоритме быстрой оболочки анали­тически.
2. Опишите с использованием геометрических представлений входные данные, для которых множество Si будет пустым. Что будет верхней оболочкой в этой ситуации?
3. Какова эффективность алгоритма быстрой оболочки в наилучшем случае?
4. Приведите пример входных данных, для которых алгоритм быстрой оболочки выполняется за квадратичное время.
5. Реализуйте алгоритм быстрой оболочки на своем любимом языке про­граммирования.

Резюме

* Метод декомпозиции (“разделяй и властвуй”) представляет собой об­щий метод разработки алгоритмов, когда задача решается путем раз­деления ее на несколько меньших подзадач (в идеале — одинакового размера), рекурсивного решения каждой из них с последующим объ­единением полученных решений подзадач в решение исходной задачи. На основе этого метода построены многие эффективные алгоритмы, хо­тя к некоторым задачам он может быть неприменим, а в других случаях давать худшие результаты, чем иное, более простое алгоритмическое решение.
* Временная эффективность Т (п) многих алгоритмов декомпозиции удо­влетворяет рекуррентному соотношению Т (п) = аТ (п/Ъ) + / (п). Ос­новная теорема позволяет определить порядок роста решения такого рекуррентного соотношения.
* Сортировка слиянием представляет собой алгоритм сортировки, осно­ванный на методе декомпозиции. Она выполняется путем разделения входного массива на две половины, рекурсивной сортировки этих по­ловин с последующим их слиянием для получения отсортированного исходного массива. Временная эффективность алгоритма — © (n log п) для любых входных данных; при этом количество сравнений ключей очень близко к теоретическому минимуму. Основным недостатком дан­ного алгоритма является необходимость большого количества допол­нительной памяти.
* Быстрая сортировка представляет собой алгоритм сортировки, осно­ванный на методе декомпозиции. Она работает путем разбиения вход­ных элементов в соответствии с их значениями относительно некото-

рого предварительно выбранного элемента. Среди прочих алгоритмов класса nlogn быстрая сортировка известна своей высокой эффектив­ностью при сортировке случайных массивов, но в наихудшем случае ее эффективность — квадратична.

* Бинарный поиск представляет собой алгоритм поиска в отсортирован­ном массиве со временем работы O(logn). Это нетипичный пример применения метода декомпозиции, поскольку требует решения только одной подзадачи половинного размера на каждой итерации.
* Классические алгоритмы обхода бинарного дерева — в прямом, цен­трированном и обратном порядке — и аналогичные алгоритмы, тре­бующие рекурсивной обработки левого и правого поддеревьев, могут рассматриваться как примеры метода декомпозиции. Их анализ упро­щается при замене всех пустых поддеревьев рассматриваемого дерева специальными внешними узлами.

• Имеется алгоритм декомпозиции, который позволяет выполнить умно­жение двух n-значных чисел с использованием п1-585 умножений цифр.

* Алгоритму Штрассена для умножения двух матриц размером 2x2 тре­буется всего 7 умножений, хотя при этом количество сложений у него выше, чем у алгоритма, основанного на определении умножения мат­риц. Метод декомпозиции позволяет перемножать матрицы размером п х п с помощью около п2 807 умножений.
* Метод декомпозиции с успехом применяется для решения таких важ­ных задач вычислительной геометрии, как задача о паре ближайших точек и задача о выпуклой оболочке.

image83

**Глава**

Метод уменьшения размера задачи

Плутарх говорит, что Серториус, желая показать своим солдатам, что настойчивость и остроумность лучше грубой силы, поставил перед ни­ми двух лошадей и приказал двум людям вырвать у них хвосты. Один из них — сущий Геркулес — безуспешно тянул и рвал хвост целиком, в то время как другой — хитроумный худышка-портной, выдергивал хвост по одному волоску, и довольно быстро и без усилий оставил лошадь бесхвостой.

— Кобхэм Брювер (Е. Cobham Brewer), Словарь идиом и аллегорий (Dictionary of Phrase and Fable), 1898.

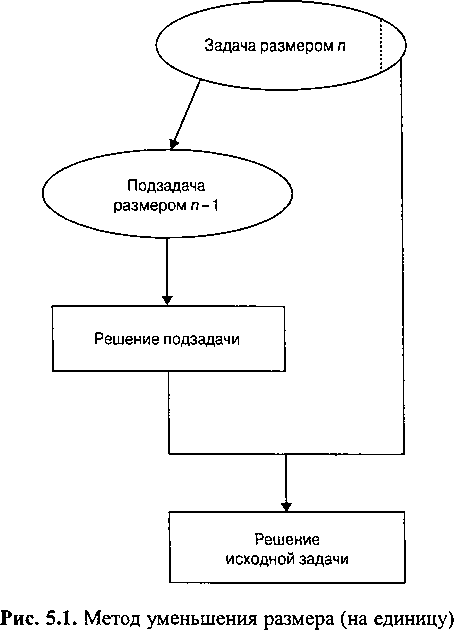
М

етод уменьшения размера задачи (“уменьшай и властвуй”) основан на ис­пользовании связи между решением данного экземпляра задачи и решени­ем меньшего экземпляра той же задачи. Если такая связь установлена, ее можно использовать либо сверху вниз (рекурсивно), либо снизу вверх (без использования рекурсии). Имеется три основных варианта метода уменьшения размера:

* уменьшение на постоянную величину;
* уменьшение на постоянный множитель;
* уменьшение переменного размера.

При уменьшении на постоянную величину размер экземпляра задачи сни­жается на одну и ту же постоянную величину при каждой итерации алгоритма. Обычно эта величина равна единице (рис. 5.1), хотя встречается и снижение раз­мера на два — в алгоритмах, которые по-разному работают для экземпляров задач четного и нечетного размера.

Рассмотрим в качестве примера задачу вычисления ап для положительных целых показателей степени. Связь между решением экземпляра размером п и эк­земпляром размером п — 1 выражается очевидной формулой ап = ап~1 • а. Таким образом, функция / (гг) = ап может быть вычислена либо “сверху вниз” с исполь-



зованием рекурсивного определения

/Н =

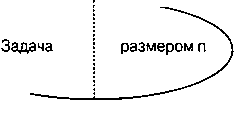
(5.1)

/ (п — 1) • а при п > 1 о при п = 1

либо “снизу вверх” путем умножения а на себя п — 1 раз (да, это тот же алгоритм, что и при использовании грубой силы, но мы пришли к нему в результате другого подхода). С более интересными примерами алгоритмов уменьшения размера на единицу вы встретитесь в разделах 5.1-5.4.

Уменьшение на постоянный множитель предполагает уменьшение размера экземпляра задачи при каждой итерации алгоритма на один и тот же множитель. В большинстве приложений этот множитель равен двум. (Можете ли вы привести пример подобного алгоритма?) Идея метода уменьшения на постоянный множи­тель показана на рис. 5.2.

В качестве примера вновь обратимся к задаче возведения в степень. Для реше­ния экземпляра задачи размером п — вычисления величины ап — решается задача половинного размера, т.е. вычисляется значение а"/2, которое связано с решением



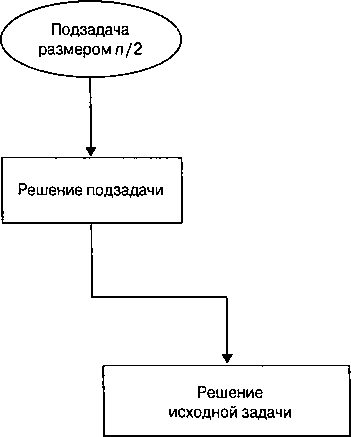


Рис. 5.2. Метод уменьшения размера (вдвое)

исходной задачи очевидным соотношением ап = (ап/2) . Однако поскольку мы рассматриваем задачу только для целых показателей степени, описанный метод годится только для четных п. Если же п нечетно, то мы должны вычислить ап~1 с использованием правила для четных показателей степени, а затем умножить результат на а. Итак, необходимо следовать такой формуле:

г (ап/2) при п положительном четном,

***ап =*** <

(5.2)

(а(п-1)/2)2 • а при нечетном гг, большем 1, а при п = 1.

При рекурсивном вычислении ап в соответствии с формулой (5.2) и измере­нии эффективности алгоритма посредством количества выполняемых умножений, следует ожидать, что он принадлежит классу О (log гг), потому что при каждой итерации размер задачи снижается как минимум вдвое, ценой не более двух умно­жений.

Обратите внимание на отличие этого алгоритма от алгоритма, основанного на методе декомпозиции и решающего две задачи возведения в степень с разме-

ром п/2:

ап = {aLU/2J ’ al"/21 ПрИ П > (5 3)

la при п = 1.

Алгоритм, основанный на формуле (5.3), неэффективен (почему?), так что алгоритм на основе формулы (5.2) работает гораздо быстрее.

Несколько других примеров алгоритмов с использованием метода уменьшения на постоянный множитель приведены в разделе 5.5 и упражнениях к нему. Такие алгоритмы весьма эффективны, однако мало распространены.

Наконец, при уменьшении переменного размера величина снижения разме­ра задачи изменяется от итерации к итерации. Примером такого алгоритма мо­жет служить алгоритм Евклида для вычисления наибольшего общего делителя. Вспомним, что этот алгоритм основан на формуле

gcd (га, п) = gcd (гг, тп mod гг).

Хотя аргументы в правой части уравнения всегда меньше аргументов в левой части (как минимум, начиная со второй итерации алгоритма), они не меньше ни на постоянное значение, ни на постоянный множитель. Несколько других примеров алгоритмов такого вида описано в разделе 5.6.

1. Сортировка вставкой

В этом разделе мы рассмотрим применение метода уменьшения на единицу к сортировке массива А [0..п — 1]. Следуя идее метода, полагаем, что задача мень­шего размера, а именно сортировка массива А [0..п — 2], уже решена и мы имеем отсортированный массив размером п — 1: А [0] ^ ^ А [п — 2]. Каким образом

воспользоваться этим решением задачи меньшего размера для получения реше­ния исходной задачи, учитывающей наличие элемента А [п — 1]? Очевидно, все, что необходимо, — это найти нужную позицию среди отсортированных элементов и вставить туда элемент А [п — 1].

Имеется три подходящих способа сделать это. Во-первых, мы можем скани­ровать отсортированный подмассив слева направо, пока не достигнем первого элемента, большего или равного А[п — 1], и после этого осуществить вставку непосредственно перед найденным элементом. Во-вторых, мы можем сканиро­вать подмассив справа налево, пока не найдем первый элемент, меньший или равный А [п — 1], и затем вставить А [п — 1] непосредственно за ним. По сути, оба варианта эквивалентны; на практике обычно используется второй способ, поскольку он лучше работает с отсортированными или почти отсортированными массивами (почему?). Получившийся в результате алгоритм сортировки называет­ся простой сортировкой вставкой (straight insertion sort), или просто сортиров­кой вставкой. Третий способ заключается в применении бинарного поиска для определения позиции вставки элемента А [п — 1] в отсортированную часть мас­сива. Получающийся при этом алгоритм носит название бинарной сортировки вставкой (binary insertion sort). В упражнениях к данному разделу вам предлагает­ся самостоятельно реализовать эту идею и исследовать эффективность бинарной сортировки вставкой.

Хотя сортировка вставкой основана на рекурсивном подходе, более эффектив­ной будет ее реализация снизу вверх, т.е. итеративная. Как показано на рис. 5.3, элемент А [г] (начиная с элемента А [1] и заканчивая А [п — 1]) вставляется в соот­ветствующее место среди первых г элементов массива (которые к этому моменту уже отсортированы). Однако в отличие от сортировки выбором элемент в об­щем случае вставляется не в окончательную позицию, которую он будет занимать в полностью отсортированном массиве.

i 1

A[0]Z-'ZA[j]<A[J+l]&-ZA[i-l] **|**

Меньше или равно A[i] Больше A[i]

Рис. 5.3. Итерация сортировки вставкой:

А [г] вставляется в корректную позицию среди уже отсортированных элементов

Ниже приведен псевдокод данного алгоритма.

Алгоритм Insertions art (А [0..п - 1])

// Сортировка массива методом сортировки вставкой // Входные данные: Массив А[0..п — 1] из п упорядочиваемых

// элементов

// Выходные данные: Массив А[0..п — 1], отсортированный // в неубывающем порядке

for i <— 1 to п — 1 do

***v*** <—■ ***A[i\ j \*— i —*** 1

while j ^ 0 and A\j) > v do A[j + l)^A[j\ j «- 3 -1 A\j + 1] <- v

Работа алгоритма проиллюстрирована на рис. 5.4.

Базовой операцией алгоритма является сравнение ключей A \j] > v. (Почему не j ^ 0? Потому что это сравнение почти всегда -быстрее предыдущего в реаль­ных реализациях. Кроме того, это сравнение не является неотъемлемой частью алгоритма: реализация с ограничителем (см. упражнение 5.1.5) позволяет полно­стью его устранить.)

**89 |** 45 **68 90 29 34**

**45 89 | 68 90 29 34**

**45 68 89 | 90 29 34**

**45 68 89 90 | 29 34**

**29 45 68 89 90 |** 34

**29 34 45 68 89 90 |**

**17 29 34 45 68 89 90**

Рис. 5.4. Пример сортиров­ки вставкой. Вертикальная чер­та отделяет отсортированную часть массива от остальных эле­ментов; вставляемый элемент выделен полужирным шрифтом

image87

Количество сравнений ключей в этом алгоритме, очевидно, зависит от при­роды входных данных. В худшем случае сравнение A [j] > v выполняется наи­большее количество раз, т.е. для всех j = i — 1,..., 0. Поскольку v = A[i\, это происходит тогда и только тогда, когда A [j] > А [г] для j = г — 1,..., 0. (Заме­тим, что мы используем тот факт, что на г-ой итерации сортировки вставкой все элементы, предшествующие А [г], — первые г элементов входных данных, хотя и расположенные в отсортированном порядке.) Таким образом, для входных дан­ных для наихудшего случая мы получаем А [0] > А [1] (для i = 1), А [1] > А [2] (для г = 2), ..., А [п — 2] > А [п — 1] (для i = п — 1). Другими словами, входные данные в наихудшем случае представляют собой массив строго уменьшающихся значений. Количество сравнений ключей для таких входных данных равно

image88

г=1 j=0 г=1

Итак, в наихудшем случае сортировка вставкой выполняет столько же сравнений, сколько и сортировка выбором (см. раздел 3.1).

В лучшем случае сравнение A [j] > v выполняется только один раз на каждой итерации внешнего цикла. Это происходит тогда и только тогда, когда А [г — 1] ^ ^ А[г\ для всех г = 1,...,тг — 1, т.е. если входной массив уже отсортирован в возрастающем порядке. (Хотя “разумно”, что наилучший случай алгоритма со­ответствует уже решенной задаче, это далеко не всегда так; вспомните, например, обсуждение быстрой сортировки в главе 4.) Итак, для отсортированного входного массива количество сравнений ключей равно

гг—1

Cbest (п) = 1 = п - 1 € 0 (гг).

Эта очень хорошая производительность, которая достигается в наилучшем случае для отсортированного входного массива, не слишком полезна сама по се­бе, поскольку нет никаких оснований ожидать именно таких приятных входных данных. Однако в ряде приложений приходится сталкиваться с почти отсорти­рованными файлами, и в этом случае сортировка вставкой имеет возможность проявить себя во всей красе. Например, при быстрой сортировке массива мож­но прекратить итерации алгоритма, когда подмассивы достигнут некоторых пред­определенных размеров (например, будут содержать меньше 10 элементов). В этот момент весь исходный массив почти отсортирован, и мы можем завершить нашу работу, применив сортировку вставкой. Обычно такое видоизменение алгоритма позволяет уменьшить общее время сортировки примерно на 10%.

Строгий анализ эффективности алгоритма в среднем случае основан на ис­следовании количества пар с нарушенным условием сортировки (см. упражне­ние 5.1.8). Он показывает, что при работе со случайными массивами сортировка вставкой выполняет примерно в два раза меньше сравнений, чем в случае убыва­ющего массива, т.е.

„2

Cavg (п) — € 0 (п[[33]](#footnote-33)) .

Такая удвоенная по сравнению с наихудшим случаем производительность ал­горитма в среднем случае в совокупности с превосходной производительностью для почти отсортированных массивов выделяет сортировку вставкой из числа дру­гих элементарных алгоритмов сортировки — сортировки выбором и пузырьковой сортировки. Кроме того, расширение сортировки вставкой — сортировка Шелла (shellsort), открытая Д. Шеллом (D.L. Shell) [107], дает еще лучший алгоритм для сортировки достаточно больших файлов (см. упражнение 5.1.10).

image89

Упражнения 5.1

1. Разработайте алгоритм на основе уменьшения размера на единицу для генерации показательного множества для множества из п элементов. (Показательное множество (power set) для множества S — это множе­ство всех подмножеств S, включая пустое множество и само множе­ство S.)
2. С помощью сортировки вставкой отсортируйте список Е, X, А, М, Р, L, Е в алфавитном порядке.
3. а) Какой ограничитель должен быть помещен перед первым элементом

сортируемого массива для того, чтобы избежать проверки выхода за границы массива j ^ 0 на каждой итерации внутреннего цикла сортировки вставкой? б) Будет ли версия алгоритма с ограничителем принадлежать тому же классу эффективности, что и исходная версия?

1. Возможно ли реализовать сортировку вставкой для сортировки связан­ных списков? Будет ли она иметь ту же эффективность О (п.2), что и версия для массивов?
2. Сравните реализацию сортировки вставкой в тексте раздела и следую­щую версию алгоритма.

Алгоритм Inserts ort2 (А [0..п - 1]) for г <— 1 to n — 1 do

j <- г - 1

while j ^ 0 and A\j] > A\j + 1] do

swap(A[j], A[j + 1]) // Обмен A[j] и A[j + 1]

j «- 3 - 1

Какова временная эффективность данного алгоритма? Сравните ее с эф­фективностью алгоритма, приведенного в тексте раздела.

1. Пусть А [0..п — 1] — массив из п сортируемых элементов. (Для просто­ты можете считать, что все элементы различны.) Пара индексов (i,j) называется инверсией (inversion), если г < j и А [г] > A [j].

а) Какой массив размером п имеет наибольшее количество инверсий, и чему оно равно? Ответьте на тот же вопрос для наименьшего числа инверсий.

б) Покажите, что количество сравнений ключей в среднем случае сор­тировки вставкой дается формулой

1. Бинарная сортировка вставкой использует для поиска позиции встав­ки элемента А [г] среди предварительно отсортированных элементов А [0] < • • • < А [г — 1] бинарный поиск. Определите класс эффектив­ности этого алгоритма в наихудшем случае.
2. Сортировка Шелла представляет собой важный алгоритм сортировки, который работает путем применения сортировки вставкой к каждому из нескольких чередующихся подфайлов данного файла. При каждом проходе по файлу подфайлы образуются путем смещения на величи­ну hi, взятую из предопределенной убывающей последовательности hi >■■■> hi >■■■> 1; при последнем проходе смещение долж­но быть равро 1. (Алгоритм работает при использовании любой такой последовательности, хотя некоторые из них с точки зрения эффектив­ности лучше других. Например, последовательность 1, 4, 13, 40, 121, ... — конечно, используемая в обратном порядке, известна как одна из лучших для данного алгоритма.)[[34]](#footnote-34)

а) Примените сортировку вставкой к списку ***S, Н, Е, L, L, S, О, R, Т, I, S, U, S, Е, F, U, L.***

б) Устойчива ли сортировка Шелла?

в) Реализуйте сортировку Шелла, простую сортировку вставкой, би­нарную сортировку вставкой, сортировку слиянием и быструю сор­тировку на вашем любимом языке программирования и сравните их производительность для случайных массивов размером 10[[35]](#footnote-35),10[[36]](#footnote-36),10[[37]](#footnote-37) и 10[[38]](#footnote-38) элементов, а также для возрастающих и убывающих файлов этих размеров.

1. Поиск в глубину и поиск в ширину

В двух следующих разделах главы мы поговорим об очень важных алгорит­мах для работы с графами, которые можно рассматривать как применение метода уменьшения размера. Предполагается, что читатель знаком с концепцией гра­фа и его основными разновидностями (неориентированный, ориентированный, взвешенный графы) и двумя основными представлениями (матрица смежности и связанные списки), а также такими понятиями, как связность и ацикличность графа. Краткий обзор этого материала можно найти в разделе 1.4.

Как указывалось в разделе 1.3, графы представляют собой интересные струк­туры с широкой областью применения. Многие алгоритмы, работающие с графа­ми, требуют определенной систематической обработки вершин или ребер графа. Для таких обходов графа имеется два основных алгоритма — поиск в глубину (depth-first search, DFS) и поиск в ширину (breadth-first search, BFS). Кроме своей основной работы (посещения всех вершин и обхода ребер графа), эти алгоритмы весьма полезны при исследовании ряда важных свойств графов.

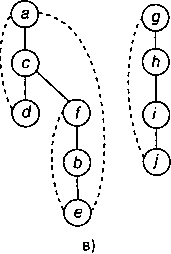
Поиск в глубину

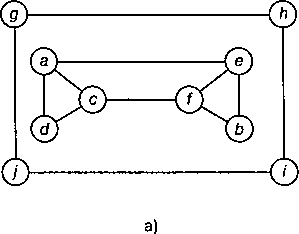
Поиск в глубину начинает посещение вершин графа с произвольной вершины, помечая ее как посещенную. На каждой итерации алгоритм обрабатывает непо- сещенные вершины, смежные с текущей. (Если таких вершин несколько, они об­рабатываются в произвольном порядке. С практической точки зрения выбор кан­дидатов для посещения определяется структурой данных, представляющей граф. В примерах мы всегда будем использовать алфавитный порядок.) Этот процесс продолжается до достижения тупика — вершины, у которой нет непосещенных смежных вершин. По достижении тупика алгоритм возвращается на одно ребро назад (в вершину, из которой он попал в тупик) и пытается продолжить посещения смежных непосещенных вершин из этого места. В конце концов алгоритм прекра­щает работу, когда возвращается в начальную вершину, которая к этому моменту становится тупиком. В этот момент все вершины данного связного компонента графа оказываются посещенными. Если в графе после этого имеются непосе- щенные вершины, процесс должен повториться, при этом в качестве стартовой используется любая из непосещенных вершин.

Для выполнения поиска в глубину удобно использовать стек. Мы помещаем вершину в стек, когда впервые посещаем ее (т.е. когда начинается обход верши­ны), и снимаем ее со стека, когда она становится тупиком (т.е. обход вершины завершается).

Очень полезно одновременно с обходом графа строить так называемый лес поиска в глубину (depth-first search forest). Стартовая вершина поиска служит кор­нем первого дерева в таком лесу. Когда впервые достигается новая непосещен- ная ранее вершина, она добавляется в качестве дочерней к вершине, из которой была достигнута. Соответствующее ребро называется ребром дерева (tree edge), поскольку множество таких ребер образует лес. Алгоритм может встретиться с ребром, которое ведет в уже посещенную вершину, не являющуюся непосред­ственным предшественником данной (т.е. не являющейся родительским узлом в дереве). Такое ребро именуют обратным ребром (back edge), поскольку оно соединяет вершину с ее предком, не являющимся родительским узлом в дереве

|  |  |  |
| --- | --- | --- |
|  | e6.2 |  |
|  | 65,3 | /l0,7 |
| d3,1 | \*4,4 | \*9,8 |
| c2,5 |  | h8,9 |
| a1,6 |  | 97, 10 |





б)

Рис. 5.5. Пример поиска в глубину, а) Граф. б) Стек обхода (первый индекс указывает порядок, в котором вершины посещались (вносились в стек), а вто­рой — в каком они становились тупиками (снимались со стека)), в) Лес поиска в глубину (ребра деревьев показаны сплошными линиями, а обратные ребра — пунктиром)

поиска в глубину. На рис. 5.5 приведен пример поиска в глубину, со стеком обхода и соответствующим лесом поиска в глубину.

Вот как выглядит псевдокод поиска в глубину.

**Алгоритм DFS** **(G)**

// Реализует поиск в глубину для данного графа // Входные данные: Граф G = (V, Е)

/I Выходные данные: Граф G, вершины которого пронумерованы // последовательными числами в порядке их

// первого посещения в процессе поиска

Помечаем все вершины в V числом 0 (непосещенная вершина) count <— О

for (для) каждой вершины г; из V do if вершина v помечена О dfs(v)

***dfs*** (***v)***

II Рекурсивно посещает все непосещенные вершины, связанные // с вершиной v, и присваивает им номера в порядке посещения // при помощи глобальной переменной count count <— count + 1; Помечаем и числом count for (для) каждой вершины w из V, смежной с v do if w помечена О dfs(w)

Краткость псевдокода процедуры DFS и простота, с какой ее можно вы­полнить вручную, создает ложное впечатление об уровне сложности данного ал­горитма. Чтобы оценить истинную мощность и глубину алгоритма, вы должны пошагово пройти его, пользуясь не графическим представлением графа, а матри­цей смежности или связанными списками смежных вершин (попробуйте сделать это для графа, изображенного на рис. 5.5, или даже для графа меньшего размера).

Насколько эффективен алгоритм поиска в глубину? Нетрудно увидеть, что этот алгоритм достаточно эффективен, поскольку время его работы пропорцио­нально размеру используемой для представления графа структуры данных. Так, для представления с использованием матрицы смежности временная эффектив­ность обхода равна 0(|V|[[39]](#footnote-39)), а для представления с использованием связанных списков — © (jV'l 4- l^l), где |V| и |£| — соответственно, количество вершин и ре­бер графа.

Лес поиска в глубину, который получается как побочный результат обхода, также заслуживает внимания. Начнем с того, что в действительности это не лес. Как мы уже видели в рассмотренном примере, ребра графа делятся при поиске в глубину на два непересекающихся класса — ребра дерева и обратные ребра (в лесу поиска в глубину для неориентированного графа не могут быть другие типы ребер). Ребра дерева — это те ребра, которые использовались поиском в глу­бину для достижения ранее непосещенных узлов. Рассматривая только эти ребра, мы в действительности получим лес. Обратные ребра соединяют вершины с ра­нее посещенными вершинами, которые не являются непосредственными предше­ственниками рассматриваемых при обходе. Обратные ребра соединяют вершины с предками, не являющимися непосредственными родителями.

Поиск в глубину и соответствующее представление графа в виде леса оказыва­ются очень полезными инструментами при разработке эффективных алгоритмов для проверки многих важных свойств графов.2 Заметим, что поиск в глубину дает два упорядочения вершин: порядок, в котором вершины впервые посещаются при обходе (вносятся в стек), и порядок, в каком они становятся тупиками (снимают­ся со стека). Это качественно различные порядки, и они применяются в разных типах приложений.

Важным применением поиска в глубину являются проверка связности графа и его ацикличности. Поскольку поиск в глубину завершается после посещения всех вершин, соединенных некоторым путем со стартовой, проверку связности графа можно выполнить следующим образом. Начнем поиск в глубину с про­извольной вершины и после завершения проверим, все ли вершины графа по­сещены. Если все, то граф связный, в противном случае — нет. Вообще говоря, поиск в глубину можно использовать для определения связных компонентов гра­фа (как?).

Для проверки наличия циклов в графе можно воспользоваться представлением графа в виде леса поиска в глубину. Если в нем не окажется обратных ребер, оче­видно, что граф ацикличен. Если же в лесу имеется обратное ребро от некоторой вершины и к вершине v (например, обратное ребро от d к а на рис. 5.5в), то граф имеет цикл, который включает в себя путь от и к л в виде последовательности ребер дерева, и обратное ребро от ukv.

Далее в книге вы найдете некоторые другие применения поиска в глубину, хотя ряд более сложных приложений, например, поиск точек сочленения графа, в ней отсутствуют. (Вершина связного графа называется точкой сочленения (ar­ticulation point), если при ее удалении вместе со всеми инцидентными ей ребрами граф распадается на непересекающиеся части.)

Поиск в ширину

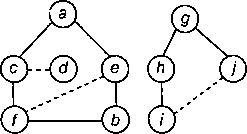
Если поиск в глубину можно назвать обходом для смелых (алгоритм идет как можно дальше от “дома”), то поиск в ширину — это обход для осторожных. Он работает “кругами”, сперва посещая все вершины, смежные со стартовой, затем — вершины, которые лежат на расстоянии двух ребер от стартовой, и т.д., пока не будут посещены все вершины связного компонента, которому принадлежит стартовая точка.

Для выполнения поиска в ширину удобно использовать очередь (обратите вни­мание на это отличие от поиска в глубину!). Очередь инициализируется стартовой вершиной поиска, помеченной как посещенная. На каждой итерации алгоритм находит все непосещенные вершины, смежные с вершиной в начале очереди, по­мечает их как посещенные и вносит в очередь; после этого вершина в начале очереди изымается из нее.

Как и в случае поиска в глубину, поиск в ширину полезно сопровождать построением так называемого леса поиска в ширину (breadth-first search forest). Стартовая вершина поиска в глубину служит корнем первого дерева в таком лесу. Когда при поиске впервые достигается новая непосещенная вершина, она присо­единяется в качестве дочерней к вершине, из которой достигнута, и соединяющее их ребро называется ребром дерева (tree edge). Если ребро ведет к уже посещен­ной вершине, отличной от непосредственного предшественника (т.е. не являю­щейся родительской вершиной в дереве), такое ребро помечается как поперечное (cross edge). На рис. 5.6 приведен пример поиска в ширину, с очередью поиска и соответствующим лесом поиска в ширину.

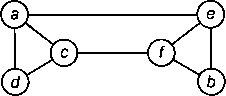
Далее представлен псевдокод поиска в ширину.

**d>**



ai с2 of3 е4 f5 Ь6 97 h8 /9 \*10

в)



б)

а)

<5

О

Рис. 5.6. Пример поиска в ширину, а) Граф. б) Очередь поиска, в которой индексы указывают очередность посещения вершин (т.е. последовательность их внесения в очередь (или извлечения из нее)), в) Лес поиска в ширину (ребра дерева показаны сплошными линиями, поперечные ребра — пунктиром)

**Алгоритм BFS** **(G)**

// Реализует поиск в ширину в данном графе // Входные данные: Граф G = (V, Е)

И Выходные данные: Граф G, вершины которого маркированы // последовательными целыми числами в том

// порядке, в каком они посещались при

// поиске в ширину

Помечаем каждую вершину в V числом 0, как непосещенную countО

for (для) каждой вершины v из V do if у помечена значением О

***bfs(v)***

***bfs (v)***

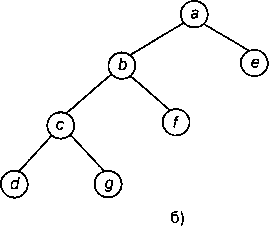
// Обход всех вершин, связанных с v и назначение им // номеров в порядке посещения при помощи глобальной // переменной count count <— count + 1

Присваиваем v значение count и инициализируем очередь вершиной v while (пока) очередь не пуста do

for (для) каждой вершины w из V, смежной с вершиной, находящейся в начале очереди do if w помечена значением О count <— count + 1 Помечаем w значением count Добавляем w в очередь Убираем из очереди первый элемент

Поиск в ширину имеет ту же эффективность, что и поиск в глубину: 0(|У|2) для представления графа с использованием матрицы смежности и © (\V\ + |.Е|) для представления с использованием связанных списков. В отличие от поиска в глубину поиск в ширину дает единственное упорядочение вершин, поскольку очередь подчиняется принципу FIFO (first-in-first-out, первым вошел — первым вышел), следовательно, порядок добавления вершин в очередь совпадает с поряд­ком их извлечения из очереди. Что касается структуры дерева поиска в ширину, то в ней так же, как и в дереве поиска в глубину, имеется два вида ребер — ребра дерева и поперечные ребра. Ребра дерева — это те ребра, которые используются для достижения ранее непосещенных вершин; поперечные ребра соединяют уже посещенные ранее вершины, но, в отличие от дерева поиска в глубину, соединя­ют либо сестринские узлы, либо “двоюродные” на том же уровне дерева поиска в ширину или на соседних уровнях.

Наконец, поиск в ширину может использоваться для проверки связности и ацикличности графа точно так же, как и поиск в глубину. Однако для некото­рых не столь простых алгоритмов, например, таких, как поиск точек сочленения, поиск в ширину неприменим. С другой стороны, имеются ситуации, где можно использовать поиск в ширину, но не в глубину — например, при поиске пути между двумя вершинами, состоящего из минимального количества ребер. В этом случае мы начинаем с поиска в ширину в одной из двух заданных вершин и за­вершаем его, когда нужная вершина достигнута. Простой путь от корня дерева поиска в ширину ко второй вершине и есть искомый кратчайший путь. Например, в графе, представленном на рис. 5.7, путь а — Ъ — с — д имеет минимальное ко­личество ребер из всех путей между вершинами а и д. Хотя корректность такого применения поиска в ширину, как представляется, следует непосредственно из принципа работы поиска в ширину, математическое доказательство этого факта не назовешь элементарным (см., например, [32]).



а)

Рис. 5.7. Иллюстрация применения поиска в ширину для определе­ния кратчайшего пути между двумя вершинами, а) Граф. б) Часть дерева поиска в ширину, позволяющая определить кратчайший путь от вершины а к вершине д

В табл. 5.1 приведены основные сведения о поиске в глубину и в ширину.

Таблица 5.1. Основные сведения о поиске в глубину (DFS) и в ширину (BFS)

|  |  |  |
| --- | --- | --- |
|  | DFS | BFS |
| Структура данных | Стек | Очередь |
| Количество упорядочений вершин | 2 | 1 |
| Типы ребер (в неориентированном | Ребра дерева и обрат­ | Ребра дерева и попе­ |
| графе) | ные ребра | речные ребра |
| Приложения | Связность, ациклич­ | Связность, ациклич­ |
|  | ность, поиск точек | ность, поиск кратчай­ |
|  | сочленения | шего пути |
| Эффективность при использовании матриц смежности | e(|F|2) | © (У|2) |
| Эффективность при использовании связанных списков смежности | 0(1^1 + \Е\) | 0(И + |£|) |
| Упражнения 5.2 |  |  |

1. Рассмотрим граф

(7)**—**СЬ**)** (с)—(д

(dj—(а)—Те)

а) Запишите матрицу смежности и связанные списки, представляю­щие данный граф (считаем, что строки и столбцы матрицы, а также вершины в связанных списках следуют алфавитному порядку меток вершин).

б) Начиная с вершины а в качестве стартовой и работая в алфавитном порядке, обойдите граф при помощи поиска в глубину и построй­те соответствующее дерево. Укажите порядок, в котором вершины впервые посещались при обходе (и вносились в стек обхода), и по­рядок, в котором они становились тупиками (и снимались со стека).

1. Если определить разреженный граф как такой, у которого \Е\ Е О (|V\), то какая из реализаций поиска в глубину для него будет более эффек­тивна — с использованием матрицы смежности или связанных списков?
2. Пусть G — граф с п вершинами и m ребрами. Верны или нет следую­щие утверждения.

а) Все леса поиска в глубину (получающиеся при обходах, которые начинаются в разных вершинах) будут состоять из одинакового ко­личества деревьев?

б) Все леса поиска в глубину будут иметь одно и то же количество ребер деревьев и одинаковое количество обратных ребер.

1. Начиная с вершины а в качестве стартовой и работая в алфавитном порядке, выполните поиск в ширину для графа из упражнения 1 и по­стройте соответствующее дерево.
2. Докажите, что поперечные ребра в дереве поиска в ширину могут со­единять вершины только на одном уровне или на соседних уровнях дерева поиска в ширину.
3. а) Объясните, как убедиться в ацикличности графа при помощи поиска

в ширину.

б) Верно ли утверждение, что один из поисков — в глубину или в ши­рину — всегда находит циклы быстрее другого? Если да, то какой из них более эффективен в этом смысле и почему. Если нет, приведите два примера, подтверждающие это.

1. Поясните, как можно идентифицировать связные компоненты графа

а) при использовании поиска в глубину;

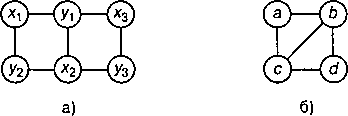
б) при использовании поиска в ширину.

1. Граф называется двудольным (bipartite), если все его вершины мож­но разделить на два непересекающихся подмножества X и Y такие, что любое ребро соединяет вершину из X с вершиной из Y. (Мы можем также сказать, что граф двудольный, если его вершины могут

быть окрашены в два цвета так, что каждое ребро соединяет верши­

а) Разработайте алгоритм на основе поиска в глубину, который бы определял, является ли граф двудольным.

ны разных цветов; такие графы называются 2-раскрашиваемыми (2- colorable). Так, на представленном ниже рисунке граф а — двудольный, а граф б — нет.

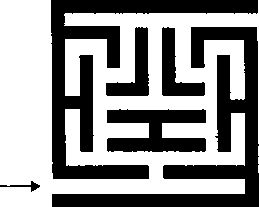


б) Разработайте алгоритм на основе поиска в ширину, который бы определял, является ли граф двудольным.

image96

1. Напишите программу, которая для данного графа выводила бы 1) вер­шины всех связных компонентов; 2) его цикл или сообщение, что граф ацикличен.
2. Лабиринт можно смоделировать при помощи вершин, соответствую­щих входу в лабиринт, выходу, тупикам и всем точкам лабиринта, в ко­торых есть возможность выбора пути, и затем соединить эти вершины ребрами, соответствующими путям в лабиринте,

а) Постройте такой граф для показанного лабиринта.

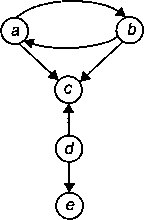


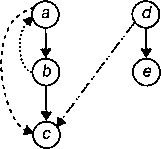
б) Какой обход — в глубину или в ширину — вы предпочтете, попав в лабиринт, и почему?

1. Топологическая сортировка

В этом разделе мы рассмотрим важную задачу, связанную с ориентированны­ми графами. Перед тем как приступить к ней, познакомимся с ориентированными графами поближе. Ориентированный граф (directed graph, digraph) — это граф, для каждого ребра которого указано направление (см., например, рис. 5.8а). Име­ется два основных вида представления ориентированных графов: при помощи матриц смежности и при помощи связанных списков. Имеются два основных раз­личия в представлении неориентированных и ориентированных графов: 1) матри­ца смежности ориентированного графа не обязана быть симметричной и 2) ребро ориентированного графа имеет только один (а не два) соответствующий узел в связанных списках смежности.

Поиск в глубину и поиск в ширину являются основными алгоритмами обхо­да ориентированных графов, но структура образующихся при этом лесов более сложна, чем в случае неориентированных графов. Так, даже для простого графа, показанного на рис. 5.8а, лес поиска в глубину, представленный на рис. 5.86, демонстрирует все четыре типа различных ребер, которые могут присутствовать в лесу поиска в глубину для ориентированного графа. Это ребра дерева (tree edge)





а) б)

Рис. 5.8. а) Ориентированный граф. б) Лес поиска в глубину для данного ориентиро­ванного графа; начальная вершина поис­ка — а

{ab, be, de), обратные ребра (back edge) (ba) от вершин к их предшественникам, прямые ребра (forward edge) {ас) от вершин к их потомкам, не являющимся непо­средственными дочерними узлами, и поперечные ребра (cross edge) (<dc), которые не принадлежат ни к одному из перечисленных ранее типов.

Заметим, что обратное ребро в лесу поиска в глубину может соединять верши­ну с родительским узлом. Так ли это или нет — в любом случае наличие обратных ребер указывает на то, что ориентированный граф имеет цикл. {Ориентированный цикл в ориентированном графе представляет собой последовательность его вер­шин, которая начинается и заканчивается в одной и той же вершине и в которой каждая вершина соединена со своим непосредственным предшественником реб­ром, направленным от предшественника к данной вершине.) Если же лес поиска в глубину для ориентированного графа не имеет обратных ребер, то такой ори­ентированный граф является ориентированным ациклическим графом (directed acyclic graph, dag).

Наличие ориентации у ребер графа приводит к новым вопросам, которые для неориентированного графа бессмысленны или тривиальны. В этом разделе будет рассмотрена одна такая задача. В качестве “затравки” рассмотрим следующий пример. Имеется множество из пяти курсов {С 1, С2, С3, С4, С5}, которые долж­ны пройти студенты. Курсы могут быть прочитаны в любом порядке с учетом следующих требований: для прослушивания курса СЪ следует сначала прослу­шать курсы С1 и С2, курс С4 следует прослушать после курса СЗ, а С5 — после СЪ и С4. Курсы читаются строго последовательно. В каком порядке должны быть прочитаны упомянутые курсы?

Ситуация может быть смоделирована при помощи графа, вершины которого представляют собой курсы, а ориентированные ребра указывают предъявляемые к ним требования (рис. 5.9). В терминах данного ориентированного графа во-

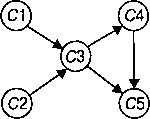


Рис. 5.9. Ориентированный граф, представляющий тре­бования к курсам

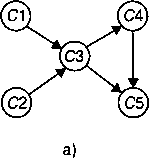
прос в том, как перечислить его вершины в таком порядке, чтобы для каждого ребра ориентированного графа вершина, в которой оно начинается, при перечис­лении оказывалась до вершины, в которой ребро оканчивается (можете ли вы указать такой порядок для приведенного ориентированного графа?). Эта задача называется задачей топологической сортировки (topological sorting). Она может быть поставлена для любого ориентированного графа, но легко понять, что в слу­чае графа с ориентированным циклом задача неразрешима. Чтобы можно было выполнить топологическую сортировку вершин графа, он должен быть ориенти­рованным ациклическим графом. Оказывается, ориентированный ациклический граф — не только необходимое, но и достаточное условие для возможности то­пологической сортировки; т.е. если ориентированный граф не имеет циклов, то задача топологической сортировки его вершин разрешима. Имеется два эффектив­ных алгоритма, которые наряду с проверкой того, является ли ориентированный граф ациклическим, выполняют (если это так) топологическую сортировку его вершин.

Первый алгоритм представляет собой простое применение поиска в глубину: выполним поиск в глубину и обратим внимание на порядок, в котором вершины становятся тупиками (т.е. снимаются со стека обхода). Обращение этого порядка дает нам решение задачи топологической сортировки — конечно, если в процессе обхода не встречаются обратные ребра. Наличие обратных ребер говорит о том, что ориентированный граф не является ациклическим, и топологическая сорти­ровка его вершин невозможна.

Почему работает данный алгоритм? Когда вершина v снимается со стека, среди уже снятых со стека к этому моменту вершин нет ни одной вершины и, для которой имеется ребро от и к v (в противном случае ребро (и, v) — обратное). Следовательно, любая такая вершина и будет находиться в списке снятия со стека после v, и перед ней — в обращенном списке.

На рис. 5.10 показано применение этого алгоритма к ориентированному гра­фу, изображенному на рис. 5.9. Обратите внимание на рис. 5.10в, где показаны ребра ориентированного графа: все они направлены слева направо, как и требу­ется в условии задачи. Это — удобный способ визуальной проверки корректности решения экземпляра задачи топологической сортировки.

Порядок снятия со стека:



С5,

С42

С33

С14 С25 б)

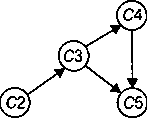
С5, С4, СЗ, С2, С1

Топологически отсортированный список: С2 С1—►СЗ—►С4—►Сб

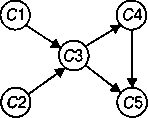
в)

Рис. 5.10. а) Ориентированный граф, для которого выполняется топологи­ческая сортировка, б) Стек обхода в глубину; индекс указывает порядок снятия со стека, в) Решение задачи топологической сортировки

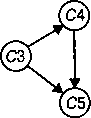
Второй алгоритм основан на непосредственной реализации метода уменьше­ния размера на единицу: в нем выполняется итеративное определение источника (source) оставшегося ориентированного графа, представляющего собой вершину, в которую не входит ни одно ребро, и удаление его вместе со всеми исходящими из него ребрами. Если таких источников несколько, произвольно выбирается один из них. Если же источников нет, задача топологической сортировки неразрешима (см. упражнение 5.3.6а). Порядок удаления вершин дает решение задачи топо­логической сортировки. Применение этого алгоритма к уже рассматривавшемуся ориентированному графу пяти курсов показано на рис. 5.11.



Удаление С2



Удаление С1



Удаление СЗ Удаление С4 Удаление С5

(С5)

Полученное решение: С1, С2, СЗ, С4, С5

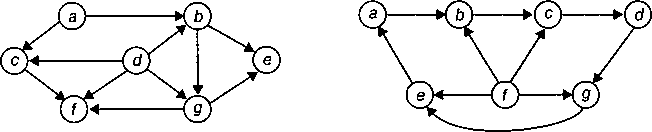
Рис. 5.11. Применение алгоритма удаления источника для решения задачи топологической сортировки. На каждой итерации из ориентированного графа удаляется вершина, в которую не входит ни одно ребро

Заметим, что решение, полученное при использовании алгоритма удаления источника, отличается от решения, которое дает алгоритм на основе поиска в глу­бину. Конечно, оба они корректны, просто задача топологической сортировки может иметь несколько различных решений.

Небольшой размер использованного нами примера ориентированного графа может создать ложное впечатление о задаче топологической сортировки. Но пред­ставьте сложный проект, например, конструкторский или исследовательский, в ко­торый включены тысячи взаимосвязанных заданий с определенными требовани­ями. Первое, что вы должны сделать в такой ситуации, — убедиться в непротиво­речивости требований. Удобным способом сделать это может послужить решение задачи топологической сортировки ориентированного графа проекта. И только после этого вы можете думать о расписании, которое позволит, например, мини­мизировать полное время работы над проектом. Само собой, для этого потребуется другой алгоритм, который вы сможете найти в книгах по исследованию операций или в специализированной литературе, посвященной методу критического пути (Critical Path Method, СРМ) или т.н. системе ПЕРТ (планирования и руководства разработками) (Program Evaluation and Review Technique, PERT).

Упражнения 5.3

1. Примените алгоритм на основе поиска в глубину для решения задачи топологической сортировки следующих ориентированных графов.



1. а) Докажите, что задача топологической сортировки ориентированного

графа имеет решение тогда и только тогда, когда он ациклический,

б) Чему равно максимально возможное количество различных реше­ний задачи топологической сортировки ориентированного графа с п вершинами?

1. а) Какова временная эффективность алгоритма топологической сорти­

ровки на основе поиска в глубину?

б) Как можно модифицировать алгоритм топологической сортировки на основе поиска в глубину, чтобы избежать обращения последова­тельности вершин, генерируемой при поиске в глубину?

1. Можно ли использовать для решения задачи топологической сорти­ровки последовательность, в которой вершины вносятся в стек поиска в глубину?
2. Примените алгоритм удаления источника к ориентированным графам из упражнения 1.
3. а) Докажите, что ориентированный ациклический граф должен иметь

как минимум один источник.

б) Каким образом можно найти вершину, в которую не входит ни одно ребро (или убедиться, что такой вершины не существует), в ори­ентированном графе, представленном матрицей смежности? Какова временная эффективность данной операции?

в) Каким образом можно найти источник (или убедиться, что он не существует) ориентированного графа, представленного связанны­ми списками смежности? Какова временная эффективность данной операции?

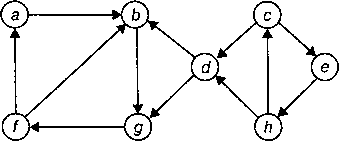
1. Можете ли вы реализовать алгоритм удаления источника для ориенти­рованного графа, представленного связанными списками смежности, так, чтобы время его работы составляло О (|V] + |-Е|)?
2. Реализуйте два алгоритма топологической сортировки на вашем люби­мом языке программирования. Проведите эксперимент по сравнению времени их работы.
3. Ориентированный граф называется сильно связным (strongly connec­ted), если для любой пары двух различных вершин существует ори­ентированный путь от и к v и ориентированный путь от v к и. В об­щем случае вершины ориентированного графа могут быть разбиты на непересекающиеся максимальные подмножества вершин, взаимно­достижимых посредством ориентированных путей ориентированного графа. Такие подмножества называются сильно связными компонен­тами (strongly connected components). Есть два алгоритма, основанных на поиске в глубину, позволяющих найти строго связанные компонен­ты. Вот как выглядит более простой (и менее эффективный) из них. Шаг 1. Выполнить поиск в глубину в данном ориентированном графе

и пронумеровать его вершины в порядке снятия со стека (в кото­ром они становятся тупиками).

Шаг 2. Обратить направления всех ребер графа.

Шаг 3. Выполнить поиск в глубину в новом ориентированном графе, начиная (и, при необходимости, повторно начиная) обход с верши­ны с максимальным номером среди еще непосещенных вершин. Сильно связные компоненты представляют собой подмножества вер­шин в каждом дереве леса поиска в глубину, полученного при послед­нем обходе.

а) Примените описанный алгоритм к приведенному ориентированному графу для поиска его сильно связных компонентов.



б) К какому классу временной эффективности относится данный ал­горитм? Дайте отдельные ответы для представления с использо­ванием матрицы смежности и с применением связанных списков смежности.

в) Сколько строго связных компонентов имеет ориентированный ацик­лический граф?

1. Транзитивное замыкание (transitive closure) ориентированного гра­фа с п вершинами можно определить как булеву матрицу Т разме­ром пхп такую, что T[i,j] = 1, если имеется путь положительной длины от вершины i к вершине j, и Т [г, j] = 0 в противном случае (1 ^ ^ гг). Разработайте алгоритм для вычисления транзитивного

замыкания и определите его временную эффективность.

1. Алгоритмы генерации комбинаторных объектов

В этом разделе мы сдержим данное ранее обещание рассмотреть алгоритмы для генерации комбинаторных объектов. Наиболее важными типами комбинатор­ных объектов являются перестановки, сочетания и подмножества данного мно­жества. Обычно они возникают в задачах, требующих рассмотрения различных вариантов выбора. В частности, мы встречались с ними в главе 3 при рассмот­рении исчерпывающего перебора. Комбинаторные объекты изучаются в разде­ле дискретной математики, именуемом комбинаторикой. Математиков, конечно, в первую очередь интересуют различные формулы для подсчета таких объектов, и мы должны быть признательны им за эти формулы, которые говорят нам, сколько элементов мы должны сгенерировать. (В частности, эти формулы предупреждают нас о том, что обычно количество комбинаторных объектов с увеличением разме­ра задачи растет экспоненциально или даже быстрее.) Однако сейчас нас больше всего интересуют алгоритмы для генерации комбинаторных объектов, а не для их подсчета.

Генерация перестановок

Начнем с перестановок. Для простоты будем считать, что множество пере­ставляемых элементов — это просто множество целых чисел от 1 до п. В общем случае их можно интерпретировать как индексы элементов n-элементного множе­ства {ai,....ап}. Как будет выглядеть применение метода уменьшения размера к задаче о генерации всех п! перестановок множества {1,..., п}? Задача меньшего на единицу размера состоит в генерации всех (п — 1)! перестановок. Полагая, что задача меньшего размера успешно решена, мы можем получить решение боль­шей задачи путем вставки п в каждую из п возможных позиций среди элементов каждой из перестановок п — 1 элементов. Все получаемые таким образом переста­новки будут различны (почему?), а их общее количество будет равно п (п — 1)! = = п\. Следовательно, таким образом мы получим все перестановки множества {1,...,п}.

Мы можем вставлять п в ранее сгенерированные перестановки слева направо или справа налево. Как оказывается, выгодно начинать вставку п в последова­тельность 12 ... (п — 1) справа налево и изменять направление всякий раз при переходе к новой перестановке множества {1,...,тг — 1}. Пример применения такого подхода для п = 3 показан на рис. 5.12.

Начало 1

Вставка 2 ^12 ^ 21^

Справа налево Вставка 3 123 132 312 321 231 213

ч J \ w J

у у

Справа налево Слева направо

Рис. 5.12. Восходящая генерация пе­рестановок (снизу вверх)

Преимущество такого порядка генерации перестановок связано с тем фактом, что этот порядок удовлетворяет так называемому требованию минимальных из­менений (minimal-change requirement): каждая перестановка получается из свей непосредственной предшественницы при помощи обмена местами только двух элементов (проверьте выполнение этого свойства для перестановок, показанных на рис. 5.12). Требование минимальных изменений выгодно как с точки зрения скорости работы алгоритма, так и для приложения, которое будет использовать сгенерированные перестановки. Например, в разделе 3.4 нам были нужны пе­рестановки городов для решения задачи коммивояжера методом исчерпывающе­го перебора. Если такие перестановки генерируются алгоритмом, удовлетворяю­щим требованию минимальных изменений, то мы можем вычислить длину нового маршрута на основе длины старого маршрута за постоянное, а не линейное вре­мя (как?).

Можно получить тот же порядок перестановок п элементов и без явной генера­ции перестановок для меньших значений п. Это можно сделать, связав с каждым компонентом к перестановки направление. Будем указывать это направление при помощи стрелки над рассматриваемым элементом, например:

3\*\*2 4\* V.

Компонент к в такой перестановке с использованием стрелок называется мобиль­ным (mobile), если стрелка указывает на меньшее соседнее число. Например, в перестановке 3 2 4 1 числа 3 и 4 мобильны, а 2 и 1 — нет. Воспользо­

вавшись понятием мобильного элемента, мы получаем следующее описание так называемого алгоритма Джонсона-Троттера (Johnson-Trotter algorithm) для ге­нерации перестановок.

**Алгоритм *JohnsonTrotter*** **(*п)***

II Реализация алгоритма Джонсона-Троттера // для генерации перестановок // Входные данные: Натуральное число п

II Выходные данные: Список перестановок множества (1,..., п} Инициализируем первую перестановку значением 1 2 ... гсГ while (пока) имеется мобильное число к do Находим наибольшее мобильное число к Меняем местами к и соседнее целое число, на которое указывает стрелка у к Меняем направление стрелок у всех целых чисел, больших к

Вот пример использования этого алгоритма для п — 3 (наибольшее мобильное число показано полужирным шрифтом):

t 2” 3 Т 3 2” 3\*Т 2 3 2 Т 2"3 Т \*2 Т 3\*.

Этот алгоритм — один из наиболее эффективных для генерации перестановок и может быть реализован со временем работы, пропорциональным количеству перестановок, т.е. © (гг!). Конечно, он очень медленный для всех значений п, кроме самых малых, но это беда не алгоритма, а задачи, которая требует от алгоритма генерации слишком большого количества элементов.

Можно показать, что порядок перестановок, генерируемых алгоритмом Джон­сона-Троттера, не совсем естественный; например, было бы естественным ожи­дать, что последняя перестановка в списке будет иметь вид п (п — 1)... 1. Именно такая перестановка окажется последней, если перестановки будут упорядочены в соответствии с лексикографическим порядком (lexicographic order), т.е. поряд­ком, в котором они были бы перечислены в словаре, если рассматривать цифры как буквы алфавита:

123 132 213 231 312 321.

Каким образом мы можем сгенерировать перестановку, следующую за а\сь2 •.. ап-\ап в лексикографическом порядке? Если an\_i < ап, мы просто меняем места­ми два последних элемента (например, за 12 3 следует 13 2). Если an\_i > an, мы должны обратиться к элементу an\_2. Если an\_2 < an\_i, мы должны переставить последние три элемента, минимально увеличивая (п — 2)-ой элемент , т.е. поме­щая на это место следующий больший an\_2 элемент, выбранный из an\_ 1 и ап, и заполняя позиции (гг — 1) и п оставшимися двумя из трех элементов an\_2, an\_ 1 и ап в возрастающем порядке. Например, за 132 следует 213, а за 231 — 312. В общем случае мы сканируем текущую перестановку справа налево в поисках первой пары соседних элементов а; и а\*+1 таких, что a\* < a\*+i (и, следовательно, aj+i > • • • > ап). Затем мы находим наименьший элемент из “хвоста”, больший аи т.е. min {aj | aj > a\*, j > г}, и помещаем его в позицию г; позиции с г + 1-ой по тг-ую заполняются элементами a\*. aj+i,..., ап, из которых изъят элемент для вставки в позицию г, в возрастающем порядке. Написание полного псевдокода этого алгоритма остается читателю в качестве самостоятельного упражнения.

Генерация подмножеств

Вспомним, как в разделе 3.5 мы рассматривали задачу о рюкзаке, в кото­рой требовалось найти наиболее ценное подмножество элементов, размещаю­щихся в рюкзаке данного объема. Мы рассматривали подход к решению этой задачи с использованием исчерпывающего перебора, который основан на гене­рации всех подмножеств данного множества элементов. В этом разделе мы рас­смотрим алгоритм для генерации всех 2п подмножеств абстрактного множества А = {ai,...,an} (математики называют множество всех подмножеств данного множества показательным множеством (power set)).

К этой задаче также непосредственно применим метод уменьшения размера на единицу. Все подмножества множества А = {ai,..., ап} можно разделить на две группы — те, которые содержат элемент an, и те, которые не содержат его. Первая группа представляет собой не что иное, как все подмножества множе­ства (ai,... ,an\_i}, в то время как все элементы второй группы можно полу­чить путем добавления элемента ап к подмножествам множества (ai,..., an\_ 1}. Следовательно, как только мы получим список всех подмножеств множества {ai,..., ап-1}, мы можем получить все подмножества множества (ai,..., an}, просто добавляя к списку все его элементы с добавленным к каждому из них элементом ап. Применение этого алгоритма для генерации всех подмножеств множества {ах,а2,аз} показано в табл. 5.2.

Таблица 5.2. Восходящая генерация подмножеств

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| п | Подмножества | | | |
| 0 | 0 |  |  |  |
| 1 | 0 | {ai} |  |  |
| 2 | 0 | W | Ы | (ai, <22} |
| 3 | 0 | {ai} | {«2} | {<21,(22} {аз} {ах, <23} {<22, <23} {аьа2,аз} |

Как и в случае перестановок, мы не обязаны генерировать все показательные множества для множеств меньших размеров. Удобный способ решения постав­ленной задачи основан на взаимно однозначном соответствии между всеми 2п подмножествами тг-элементного множества А = {ах,...,ап} и всеми 2П бито­выми строками bi,... , Ьп длиной п. Простейший способ установить такое соот­ветствие — назначить подмножеству битовую строку, в которой bi — 1, если а\* принадлежит данному подмножеству, и bi = 0 в противном случае (эта идея уже упоминалась в разделе 1.4, когда мы говорили о битовых векторах). Например, битовая строка ООО соответствует пустому подмножеству трехэлементного мно­жества, а 111 — самому исходному трехэлементному множеству, т.е. {ах,а2,аз}. Понятно, что строка 110 представляет подмножество {ai,a2}. Используя такое соответствие, мы можем сгенерировать все битовые строки длиной гг, просто генерируя последовательно двоичные числа от 0 до 2n — 1, добавляя при необхо­димости соответствующее количество ведущих нулей. Например, при п = 3 мы получим:

Битовые строки ООО 001 010 011 100 101 110 111

Подмножества 0 {а3} {а2} {<\*2,аз} {ах} {ai,a3} {ai,a2} {ах,а2,аз}

Заметим, что в то время как битовые строки, сгенерированные данным ал­горитмом, находятся в лексикографическом порядке (с использованием двухсим­вольного алфавита {0,1}), порядок подмножеств выглядит далеко не естествен­но. Например, мы можем захотеть получить так называемый плотный поря- док (squashed order), когда подмножество, включающее aj, может находиться в списке только после всех подмножеств, включающих элементы ах,...,а^\_х (j = 1,...,п), как, например, в случае трехэлементного исходного множества, показанного в табл. 5.2. Описанный алгоритм, основанный на использовании би­товых строк, весьма легко модифицировать для получения желаемого плотного порядка (см. упражнение 5.4.6).

Более интересный вопрос — о существовании алгоритма генерации битовых строк, соответствующего требованию минимальных изменений, так чтобы каждая строка отличалась от непосредственно предшествующей ей только одним битом (в переводе на язык подмножеств это означает, что каждое подмножество отли­чается от своего предшественника в списке либо добавлением, либо удалением одного элемента, но не более). Ответ на этот вопрос положительный; например, для п = 3 мы можем получить строки

ООО 001 011 010 110 111 101 100.

Это — пример кода Грея (Gray code). Коды Грея обладают многими инте­ресными свойствами и рядом полезных приложений, о чем вы можете прочесть в [25].

Упражнения 5.4

1. Реально ли реализовать алгоритм, который требует генерации всех пе­рестановок 25-элементного множества, на вашем компьютере? А в слу­чае генерации всех подмножеств данного множества?
2. Сгенерируйте все перестановки множества (1,2,3,4} с использовани­ем

а) восходящего алгоритма, соответствующего требованию минималь­ных изменений;

б) алгоритма Джонсона-Троттера;

в) алгоритма лексикографического упорядочения.

1. Напишите компьютерную программу для генерации перестановок в лексикографическом порядке.
2. Рассмотрим следующую реализацию алгоритма для генерации пере­становок, открытого Хипом (В. Heap) [50].

Алгоритм ***HeapPermute (п)***

// Реализация алгоритма Хипа для генерации перестановок // Входные данные: Натуральное п и глобальный массив А[1..ть]

II Выходные данные: Все перестановки элементов множества А if гг = 1 write А else

for г <— 1 to n do

***HeapPermute(n —*** 1) **if *n*** нечетно

Обмениваем A{ 1] и A[n]

else

Обмениваем А [г] и А[п]

а) Пошагово пройдите алгоритм для п — 2, 3 и 4.

б) Докажите корректность алгоритма Хипа.

в) Определите временную эффективность алгоритма HeapPermute.

1. Сгенерируйте все подмножества четырехэлементного множества А = = {ai, 02) 03.04} при помощи обоих описанных в тексте раздела алго­ритмов.
2. Какой простой трюк заставляет алгоритм с использованием битовых строк генерировать подмножества в плотном порядке.
3. Напишите псевдокод для рекурсивного алгоритма генерации всех 2" битовых строк длиной п.
4. Напишите нерекурсивный алгоритм для генерации 2п битовых строк длиной п, который реализует битовые строки в виде массивов и не использует бинарного сложения.
5. а) Используйте метод уменьшения размера на единицу для генерации

кода Грея для п — 4.

б) Разработайте алгоритм на основе метода уменьшения размера на единицу для генерации кода Грея произвольного порядка п.

1. Разработайте алгоритм на основе метода уменьшения размера для ге­нерации всех комбинаций из к элементов, выбранных из п-элементного множества (т.е. всех fc-элементных подмножеств данного п-элементного множества). Удовлетворяет ли ваш алгоритм требованию минимальных изменений?
2. Алгоритмы с использованием

уменьшения на постоянный множитель

Во введении к этой главе упоминалось, что алгоритмы, основанные на умень­шении размера задачи на постоянный множитель — вторая важная разновидность алгоритмов, основанных на уменьшении размера задачи. Мы уже встречались в книге с примерами этого метода — это бинарный поиск (раздел 4.3) и возве­дение в степень посредством возведения в квадрат (введение к данной главе). В этом разделе мы познакомимся с другими примерами алгоритмов, основанных на уменьшении размера задачи на постоянный множитель. Однако мы не должны ожидать наличия большого количества таких алгоритмов. Обычно такие алго­ритмы — логарифмические и, будучи очень быстрыми, встречаются достаточно редко. Особенно большая редкость — множитель, не равный двум.

**Задача поиска фальшивой монеты**

Из целого ряда различных задач поиска фальшивой монеты мы рассмотрим одну, которая наилучшим образом иллюстрирует интересующий нас метод умень­шения размера задачи на постоянный множитель. Среди п одинаково выглядящих монет одна — фальшивая. На рычажных весах вы можете сравнить любые два мно­жества монет и получить ответ, какое из множеств тяжелее (или что они равны по весу), но не на какую именно величину. Задача заключается в том, чтобы раз­работать эффективный алгоритм для поиска фальшивой монеты. В простейшей версии задачи, рассматриваемой в этом разделе, считается, что вы знаете, легче или тяжелее фальшивая монета по сравнению с подлинной.[[40]](#footnote-40) (Здесь мы будем считать, что фальшивая монета легче настоящей.)

Наиболее естественный подход к решению данной версии задачи — разделить п монет на две кучки по [п/2\ монет в каждой, отложив одну монету, если п — нечетное число. Если кучки имеют одинаковый вес, то отложенная монета и является фальшивой; если нет — мы выбираем более легкую кучку и выполняем описанные действия с ней. Заметим, что, хотя мы и разделили все монеты на две кучки, после взвешивания надо решить только одну задачу половинного размера. Следовательно, в соответствии с нашей классификацией методов проектирования алгоритмов, этот алгоритм относится к алгоритмам уменьшения размера задачи (вдвое), а не к алгоритмам на основе декомпозиции.

Легко записать рекуррентное уравнение для количества взвешиваний W (п), необходимых алгоритму в наихудшем случае:

W(n) = W([n/2J) + 1 при гг > 1,^(1) -0.

Это рекуррентное соотношение должно быть вам знакомо. В самом деле, оно практически идентично соотношению для количества сравнений в бинарном по­иске в наихудшем случае (отличие только в начальном условии). Эта схожесть неудивительна, поскольку оба алгоритма основаны на одном и том же методе — уменьшения размера экземпляра задачи в два раза. Решение данного рекуррент­ного соотношения также очень похоже на решение соотношения для бинарного поиска: W (п) = |\_log2 п\ •

Все это выглядит элементарно, если не просто надоедливо. Но подождите немного: самое интересное в том, что это решение — не самое эффективное. Мы можем поступить разумнее, если будем делить монеты на три кучки, примерно по п/3 монет в каждой. (Детали точной формулировки остаются в качестве упраж­нения. Не пропустите его! Даже если преподаватель забудет об упражнении 5.5.3, напомните ему о том, что это упражнение должно быть обязательно выполнено.) После взвешивания двух кучек мы можем уменьшить размер экземпляра задачи в три раза, так что следует ожидать, что необходимое количество взвешиваний будет примерно равно log3 п, что меньше, чем log2 п (можете ли вы сказать, во сколько раз?).

**Умножение по-русски**

Сейчас мы рассмотрим один нестандартный алгоритм для умножения двух положительных чисел, который называется умножением по-русски (multiplication a la russe), или русским крестьянским методом (Russian peasant method). Пусть п и га — натуральные числа, произведение которых мы хотим вычислить, а размером экземпляра задачи будем считать величину п. Теперь, если п четно, экземпляр задачи уменьшается вдвое, до п/2, и для решения большего экземпляра задачи мы используем решение меньшего экземпляра при помощи очевидной формулы

п ~

***п- т— —*** • **2**га.

Если п нечетно, требуется внесение в формулу небольшой поправки:

п - 1

п ' т — —-— • 2т + т.

Z

Используя приведенные формулы и тривиальный частный случай 1 • га = га для остановки, мы можем рекурсивно или итеративно вычислить произведение п • т. Пример вычисления 50 • 65 приведен на рис. 5.13. Обратите внимание, что дополнительные слагаемые (показанные в скобках) имеются в тех строках, где в первом столбце стоит нечетное значение. Таким образом, при умножении вруч­ную никакие скобки (которые для ясности показаны на рис. 5.13а) не требуются, и для получения результата надо просто просуммировать значения из столбца т, соответствующие нечетным значениям в столбце гг, что и сделано на рис. 5.136.

Заметим также, что в алгоритме используются только простые операции удво­ения, деления пополам и сложения — это должно показаться привлекательным тем, у кого проблемы с таблицей умножения. По крайней мере, как утверждают западные путешественники, русские крестьяне (именем которых и назван дан­ный метод) еще в XIX веке использовали итеративные алгоритмы, основанные на методе уменьшения размера задачи вдвое. (Впрочем, идея этого алгоритма бы­ла известна еще примерно в 1650 г. до н.э. египетским математикам — см. [26], стр. 16.) Этот алгоритм приводит к очень быстрой аппаратной реализации умно­жения, поскольку удвоение и деление пополам двоичных чисел реализуются при помощи сдвига, который представляет собой одну из базовых операций на аппа­ратном уровне.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| п | т |  | п | 771 |  |
| 50 | 65 |  | 50 | 65 |  |
| 25 | 130 |  | 25 | 130 | 130 |
| 12 | 260 | (+130) | 12 | 260 |  |
| 6 | 520 |  | 6 | 520 |  |
| 3 | 1040 |  | 3 | 1040 | 1040 |
| 1 | 2080 | (+1040) | 1 | 2080 | 2080 |
|  | 2080 | + (130 + 1040) = 3250 |  |  | 3250 |
|  |  | а) |  | б) |  |

Рис. 5.13. Вычисление 50 • 65 умножением по-русски

**Задача Иосифа**

Наш последний пример — задача Иосифа (Josephus problem), названная так по имени Флавия Иосифа (Flavius Josephus), известного еврейского историка, участ­ника и летописца восстания евреев против римлян в 66-70 гг. Иосиф в течение 47 дней командовал обороной крепости Иотапата (Jotapata), и после того, как она пала, укрылся с 40 фанатиками в пещере неподалеку. Здесь повстанцы решили по­кончить с собой, чтобы не сдаться римлянам. Иосиф предложил, чтобы все встали в круг и по очереди каждый убивал своего соседа (т.е. каждого второго)[[41]](#footnote-41), пока не останется один человек, который должен совершить самоубийство. Нечего и го­ворить, что решивший сдаться Иосиф поставил себя и своего единомышленника на такие места в круге, что-они остались последними.

Итак, пусть п человек, пронумерованных от 1 до гг, встали в круг. Начнем счет с человека под номером 1 и будем убирать из круга каждого второго, пока в круге не останется только один человек. Задача состоит в том, чтобы определить номер этого уцелевшего J (гг). Например, если п — 6 (см. рис. 5.14), то при первом проходе из круга будут удалены номера 2, 4 и 6, а на втором проходе — номера 3 и 1, так что решение задачи равно J (6) = 5. Если п = 7, то при первом проходе будут удалены номера 2, 4, 6 и 1 (оказывается удобным включить удаление 1 в первый проход), а на втором — 5 и 3 (удаление 3 удобнее отнести ко второму проходу), так что решением данного экземпляра задачи является J (7) = 7.

Удобно рассматривать четные и нечетные значения п отдельно. Если п четно, т.е. п = 2к, то после первого прохода мы получаем экземпляр той же задачи, но половинного размера. Единственное отличие — в нумерации: например, чело-

а) б)

Рис. 5.14. Экземпляры задачи Иосифа при а) п = 6иб)п = 7. Индексы ука­зывают номер прохода, во время которо­го устраняется данный человек. Решения показанных экземпляров равны J (б) = 5 и J (7) = 7

век номер 3 на втором проходе находится в позиции 2, номер 5 — в позиции 3 и т.д. Легко увидеть, что для того, чтобы получить начальную позицию человека, надо просто умножить его новую позицию на 2 и вычесть 1. Это соотношение выполняется, в частности, для уцелевшего человека, так что

***J (2к) = 2J (к) - h***

Давайте теперь рассмотрим случай нечетного п > 1, т.е. п = 2к + 1. При первом проходе удаляются все люди в четных позициях. Если мы добавим сюда и человека номер 1, то получим экземпляр задачи размера к. В этот раз, чтобы получить начальную позицию человека по его новой позиции, мы должны умножить новую позицию на 2 и прибавить 1 (см. рис. 5.146). Итак, для нечетных значения п получаем

J (2/с + 1) = 2J (к) + 1.

Можем ли мы получить решение этих двух рекуррентных соотношений в яв­ном виде (с учетом начального условия J( 1) = 1)? Ответ положителен, хотя решение и требует более сложных приемов, чем простая обратная подстанов­ка. Один из способов решения — применение прямой подстановки для получе­ния, скажем, пятнадцати первых значений J (гг), выявление закономерности и ее доказательство методом математической индукции. Выполнение этих действий оставлено читателю в качестве самостоятельного упражнения. Вы можете также обратиться к [45] и найти там все интересующие вас сведения (данный раздел изложен по материалам этой книги). Интересно, что наиболее элегантный вид ре­шения рекуррентных соотношений использует бинарное представление числа п: J (п) можно получить путем циклического сдвига п на один бит влево! Например, J (6) = J (1102) = Ю12 = 5 и J (7) = J (1112) = 1112 = 7.

1. Разработайте алгоритм вычисления [log2nJ, основанный на методе уменьшения размера задачи вдвое, и определите его временную эф­фективность.
2. Рассмотрим тернарный поиск (ternary search) — следующий алгоритм для поиска в отсортированном массиве Д [0..тг — 1]: если п = 1, мы просто сравниваем ключ К с единственным элементом массива; в про­тивном случае поиск рекурсивно сравнивает К с А [[п/3\] и, если К больше, сравнивает его с Л[|\_2тг/3\_|] для определения того, в какой трети массива следует продолжить поиск.

а) На каком методе основан данный алгоритм?

б) Запишите рекуррентное соотношение для количества сравнений ключей в наихудшем случае (можно считать, что п = 3fc).

в) Решите это рекуррентное соотношение при п = Зк.

image107

г) Сравните эффективность данного алгоритма с эффективностью би­нарного поиска.

1. а) Напишите псевдокод алгоритма поиска фальшивой монеты путем

деления на три кучи. (Убедитесь, что ваш алгоритм корректно ра­ботает для любого значения гг, а не только для кратного 3.)

б) Запишите рекуррентное соотношение для количества взвешиваний в алгоритме поиска фальшивой монеты путем деления на три кучи и решите его для п = Зк.

в) Во сколько раз этот алгоритм быстрее алгоритма поиска фальши­вой монеты путем деления на две кучи при больших п? (Ответ не должен зависеть от п.)

1. Примените метод умножения по-русски для вычисления 26 • 47.
2. а) Имеет ли значение с точки зрения эффективности, выполняем ли

мы умножение по-русски п • га или га • гг? б) Определите класс эффективности умножения по-русски.

1. Напишите псевдокод алгоритма умножения по-русски.
2. Найдите J (40) — решения задачи Иосифа для п = 40.
3. Докажите, что решение задачи Иосифа равно 1 для всех гг, являющихся степенью 2.
4. Для задачи Иосифа

а) Вычислите J (гг) для п — 1,2,..., 15.

б) Найдите закономерность в решениях задачи Иосифа для первых пят­надцати значений п и докажите ее справедливость в общем случае.

в) Докажите корректность получения значения J (п) путем цикличе­ского сдвига бинарного представления п.

1. Алгоритмы с переменным уменьшением размера

Как упоминалось во введении к данной главе, третьим основным вариантом метода уменьшения размера задачи является уменьшение, меняющееся от итера­ции к итерации. Хорошим примером такого алгоритма является алгоритм Евклида для вычисления наибольшего общего делителя. В этом разделе мы познакомимся и с другими представителями этого класса алгоритмов.

**Вычисление медианы и задача выбора**

Задача выбора (selection problem) заключается в поиске к-то наименьшего эле­мента в списке из п чисел. Это число называется к-ой порядковой статистикой (order statistic). Естественно, при k = 1 или к = п можно просто сканировать весь список в поисках наименьшего или наибольшего элемента. Более интересен случай, когда к = [гг/2], т.е. когда надо найти элемент, больший одной поло­вины элементов списка, и меньший — другой половины. Такое среднее значение, называемое медианой (median), является одной из наиболее важных величин в ма­тематической статистике. Очевидно, мы можем найти к-ый по величине элемент, отсортировав весь список и выбрав к-ый по порядку элемент из отсортированного списка. Время работы такого алгоритма определяется эффективностью использу­емого алгоритма сортировки. При использовании хорошего алгоритма типа сорти­ровки слиянием эффективность такого алгоритма поиска порядковой статистики равна О (гг log п).

Закрадывается подозрение, что сортировка всего списка — выполнение лиш­ней работы, поскольку в задаче не требуется сортировать список, а надо только указать один-единственный элемент. К счастью, у нас имеется очень эффектив­ный (в среднем) алгоритм для выполнения похожей задачи — разбиения элементов массива на два подмножества: одно из них содержит элементы, не превышающее некоторое опорное значение р, а второе — элементы, которые не меньше р:

Q'ii • • • O'ig-1 Р \*

image108

ч.

Такое разбиение является важной частью алгоритма быстрой сортировки, рас­сматривавшейся в главе 4.

Как воспользоваться преимуществами такого разбиения? Пусть s — позиция разбиения. Если s = к, то опорный элемент р и есть решением задачи выбора. (Ес­ли элементы в списке нумеруются начиная с 0, то, конечно, решение получается при s = к — 1.) Если s > к, то к-ый наименьший элемент находится в левой части списка, а если s < fc, то надо найти (к — з)-ый наименьший элемент из правой части списка. Итак, если мы не получаем решение задачи на данной итерации, то по крайней мере уменьшаем ее размер и получаем задачу, которая решается таким же методом, т.е. рекурсивно. На самом деле реализовать эту же идею можно и без применения рекурсии. При использовании нерекурсивной версии даже не надо изменять значение к — мы просто продолжаем выполнение вычислений, пока не получим s = к.

Пример 1. Найдем медиану списка из девяти элементов: 4, 1, 10, 9, 7, 12, 8, 2, 15. В этом случае к = [9/2] = 5, и наша задача состоит в поиске пятого по величине элемента массива. Как и ранее, мы предполагаем, что элементы в списке проиндексированы от 1 до 9.

Мы можем воспользоваться той же версией разбиения, которой пользовались при рассмотрении алгоритма быстрой сортировки в главе 4, т.е. выбирая в каче­стве опорного первый элемент и переставляя элементы в процессе двух противо­положно направленных сканирований массива:

4 1 10 9 7 12 8 2 15

2 1 4 9 7 12 8 10 15

Поскольку s = 3 < к = 5, продолжаем работу с правой частью списка:

9 7 12 8 10 15

8 7 9 12 10 15

Поскольку s = 6 > к = 5, мы работаем с левой частью списка:

8 7 7 8

Теперь s = к = 5, так что можно остановиться: мы нашли медиану, равную 8 — которая больше 2, 1, 4 и 7, но меньше 9, 12, 10 и 15. ■

Насколько эффективен этот алгоритм? Следует ожидать, что в среднем он бо­лее эффективен, чем быстрая сортировка, так как работает только с одной частью исходного массива после разбиения, в то время как быстрая сортировка должна обработать обе части. Если считать, что разбиение всегда выполняется посредине остающейся части массива, рекуррентное соотношение для количества сравнений будет иметь следующий вид:

С (п) — С (п,/2) + (тг + 1),

решение которого, согласно Основной теореме (см. главу 4), равно © (п). Хотя на самом деле размер массива уменьшается от одной итерации к другой непредсказу­емым образом (иногда это уменьшение меньше, чем в два раза, иногда — больше), тщательный математический анализ показывает, что эффективность в среднем случае будет такой же, как если бы уменьшение размера задачи всякий раз было ровно в два раза. Другими словами, в среднем мы получаем линейный алгоритм. В худшем же случае эффективность алгоритма падает до © (гг2). Хотя кибернети­ки и открыли алгоритм, который решает задачу выбора за линейное время даже в наихудшем случае [19], он слишком сложен для практического применения.

Заметим также, что на самом деле алгоритм на основе разбиения решает немного более общую задачу, а именно: определение к наименьших и п — к наибольших элементов данного списка, а не только поиск fc-го элемента в порядке возрастания.

**Интерполяционный поиск**

В качестве следующего примера алгоритма с переменным уменьшением раз­мера задачи мы рассмотрим еще один алгоритм поиска в отсортированном масси­ве, который называется интерполяционным поиском (interpolation search). В от­личие от бинарного поиска, который всегда сравнивает ключ поиска со средним значением отсортированного массива (а следовательно, всегда уменьшает размер задачи вдвое), интерполяционный поиск учитывает значение ключа поиска при определении элемента массива, который будет сравниваться с ключом. В опреде­ленном смысле алгоритм имитирует поиск имени в телефонной книге. Если мы ищем в телефонной книге, например, Горбенко — вряд ли мы будем открывать ее в средине или ближе к концу, как поступили бы при поиске Подгорного.

Говоря более строго, при выполнении итерации поиска между элементами А [/] (крайним слева) и А [г] (крайним справа), алгоритм предполагает, что значения в массиве растут линейно (отличие от линейности может влиять на эффектив­ность, но не на корректность данного алгоритма). В соответствии с этим предпо­ложением, значение v ключа поиска сравнивается с элементом, индекс которого вычисляется (с округлением) как координата х точки на прямой, проходящей через (/, А [/]) и (г, А [г]), координата у которой равна значению v (см. рис. 5.15).

Записав стандартное уравнение для прямой, проходящей через две точки (/, A [Z]) и (г, А [г]), заменив в нем у на г; и решая его относительно х, получим формулу

{v-A [Z]) (г - /)

(5.4)

X — I +

Л [г]-Л [(]

Логика, лежащая в основе этого метода, проста. Мы знаем, что значения мас­сива возрастают (точнее говоря, не убывают) от А [/] до А [г], но не знаем, как именно. Пусть это возрастание — линейное (простейшая из возможных функци-

Значение

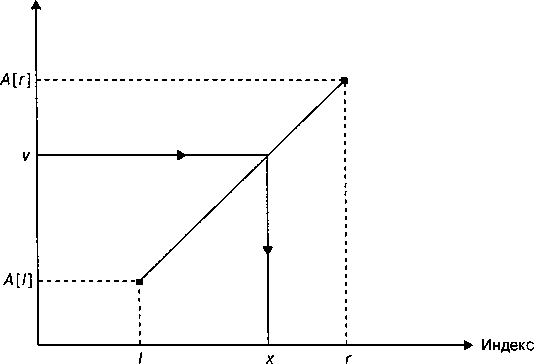


Рис. 5.15. Вычисление индекса при интерполяционном поиске

ональных зависимостей); в таком случае вычисленное по формуле (5.4) значение индекса — ожидаемая позиция элемента со значением, равным v. Конечно, если v не находится между А[1]и А [г], формулу (5.4) применять не следует (почему?).

После сравнения v с А[х] алгоритм либо прекращает работу (если они равны), либо продолжает поиск тем же способом среди элементов с индексами либо от I до х — 1, либо от х + 1 до г, в зависимости от того, меньше ли v значения А [х] или больше. Таким образом, на каждой итерации размер задачи уменьшается, но априори мы не знаем, насколько именно.

Анализ эффективности данного алгоритма показывает, что интерполяционный поиск в среднем требует менее log2 log2 п+1 сравнений ключей при поиске в спис­ке из п случайных значений. Эта функция растет настолько медленно, что для всех реальных практических значений п ее можно считать константой (см. упражне­ние 5.6.6). Однако в наихудшем случае интерполяционный поиск вырождается в линейный, который рассматривается как наихудший из возможных (почему?). В качестве последнего замечания по поводу сравнения интерполяционного по­иска с бинарным приведем мнение Седжвика (R. Sedgewick) [102], считающего, что бинарный поиск, вероятно, более выгоден для небольших входных данных, но для файлов большого размера и для приложений, в которых сравнение или обращение к данным — дорогостоящая операция, лучше использовать интерпо­ляционный поиск. Упомянем, что в разделе 11.4 будет рассмотрена непрерывная версия интерполяционного поиска, которую также можно рассматривать как еще один пример алгоритма, основанного на переменном уменьшении размера задачи.

**Поиск и вставка в бинарное дерево поиска**

В качестве последнего примера в данном разделе вернемся к бинарным де­ревьям поиска. Вспомним, что бинарное дерево поиска — это бинарное дерево, узлы которого содержат упорядочиваемые элементы, по одному в каждом узле, так что для каждого узла все элементы в левом поддереве меньше, а все элементы в правом поддереве больше элемента в корне поддерева. Когда надо найти эле­мент с заданным значением (скажем, v) в таком дереве, мы рекурсивно повторяем следующие действия. Если дерево пустое — поиск завершается неудачно. Если дерево не пустое, мы сравниваем значение v со значением в корне дерева К (г). Если они равны, искомый элемент найден, и поиск завершается успешно. Если же они не равны, то мы продолжаем поиск в левом поддереве, если v < К (г), и в правом, если v > К (г). Таким образом, на каждой итерации алгоритма задача поиска в бинарном дереве поиска сводится к задаче поиска в бинарном дереве меньшего размера. Наиболее разумной мерой размера бинарного дерева поиска является его высота; очевидно, что уменьшение высоты дерева может изменяться от итерации к итерации — это дает нам пример алгоритма с переменным умень­шением размера задачи.

В худшем случае бинарного дерева поиска оно строго линейное. Такое бывает, в частности, если дерево строится путем вставки возрастающей или убывающей последовательности ключей (рис. 5.16).

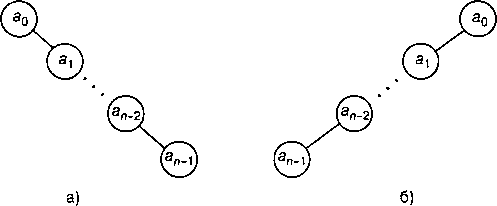


Рис. 5.16. Бинарные деревья поиска для а) возрастаю­щей последовательности ключей и б) убывающей по­следовательности ключей

Очевидно, что поиск an\_i в таком дереве требует п сравнений, что приво­дит к эффективности © (гг) в наихудшем случае. К счастью, эффективность по­иска в среднем случае — ©(logгг). Говоря более точно, количество сравнений ключей, необходимое для поиска в бинарном дереве поиска, построенном из п случайных ключей, приблизительно равно 2 In гг« 1.39 log2 гг. Поскольку вставка нового ключа в бинарное дерево поиска практически идентична поиску в нем, задача вставки в бинарное дерево поиска также является примером алгоритма

с переменным уменьшением размера задачи и имеет ту же эффективность, что и операция поиска.

**Упражнения 5.6**

image111

image112

1. Игра Ним с одной кучкой камней. Рассмотрим следующую игру Име­ется кучка из п камней. Два игрока по очереди берут из нее от 1 до 4 камней за ход. Побеждает тот, кто возьмет последний камень. Раз­работайте выигрышную стратегию для игрока, делающего первый ход (если таковая существует).
2. Переворачивающиеся блинчики. Имеется стопка из п блинчиков разно­го размера. Вы можете вставить лопатку под любой блинчик и пере­вернуть всю стопку, оказавшуюся на лопатке, “вверх ногами”. Ваша задача — при помощи последовательности таких операций разложить блинчики по размерам так, чтобы самый больший из них был в самом низу. (Визуализацию этой головоломки можно найти на узле Interac­tive Mathematics Miscellany and Puzzles [20]). Разработайте алгоритм для решения этой головоломки.

Резюме

* Метод уменьшения размера задачи для разработки алгоритмов основан на использовании соотношения между решением данного экземпляра задачи и решением меньшего экземпляра той же задачи. Если такое соотношение установлено, оно может использоваться либо сверху вниз (рекурсивно), либо снизу вверх (без рекурсии).
* Имеются три основные разновидности метода уменьшения размера задачи:
* уменьшение на постоянную величину, чаще всего на единицу (на­пример, сортировка вставкой);
* уменьшение на постоянный множитель, чаще всего на два (на­пример, бинарный поиск);
* переменное уменьшение размера (например, алгоритм Евклида).
* Сортировка вставкой представляет собой непосредственное примене­ние метода ^енынения размера на единицу к задаче сортировки. Этот алгоритм имеет эффективность 0(тг[[42]](#footnote-42)) как в среднем, так и в наихудшем случае, но в среднем он примерно вдвое быстрее, чем в наихудшем слу­чае. Главное преимущество алгоритма — высокая производительность для почти отсортированных массивов.
* Поиск в глубину и поиск в ширину — два основных алгоритма обхода графов. Представление графов в виде леса поиска в глубину или леса поиска в ширину позволяет изучить многие важные свойства графов. Оба алгоритма имеют одинаковую эффективность: 0(|У|2) для пред-

ставления с использованием матрицы смежности и © (|V"| + \Е\) для представления с использованием связанных списков смежности.

* Ориентированный граф — это граф, ребра которого имеют ориента­цию. Задача топологической сортировки состоит в перечислении вер­шин графа в таком порядке, что для каждого ребра графа его начальная вершина находится в списке до конечной вершины. Эта задача имеет решение тогда и только тогда, когда ориентированный граф является ориентированным ациклическим графом, т.е. не содержит ориентиро­ванных циклов.
* Имеется два алгоритма для решения задачи топологической сортиров­ки. Первый основан на поиске в глубину; второй представляет собой непосредственное применение метода уменьшения размера задачи на единицу.
* Метод уменьшения размера задачи на единицу является естественным подходом к разработке алгоритмов для генерации элементарных комби­наторных объектов. Наиболее эффективный класс таких алгоритмов — алгоритмы, удовлетворяющие требованию минимальных изменений. Однако количество комбинаторных объектов растет настолько быст­ро, что даже лучшие алгоритмы представляют практический интерес только для очень маленьких экземпляров таких задач.
* Поиск фальшивой монеты на рычажных весах, умножение по-рус­ски и задача Иосифа представляют собой примеры задач, решаемых с использованием алгоритмов уменьшения размера задачи на посто­янный множитель. Два других, более важных примера таких алгорит­мов — бинарный поиск и возведение в степень посредством возведения в квадрат.
* Для некоторых алгоритмов уменьшения размера задачи величина уменьшения варьируется от итерации к итерации. Примерами таких алгоритмов с переменным уменьшением размера задачи служат ал­горитм Евклида, алгоритм решения задачи выбора путем разбиения, интерполяционный поиск, а также поиск и вставка в бинарном дереве поиска.

**Глава**

Метод преобразования

Секрет жизни ... в замене одних беспокойств другими.

— Чарльз Шульц (Charles М. Schulz) (1922-2000), американский карикатурист

Э

та глава посвящена группе методов, основанных на идее преобразования. Мы называем эту общую технологию “преобразуй и властвуй”, поскольку такие методы работают в две стадии. Сначала, на стадии преобразования, экземпляр задачи преобразуется в другой, по той или иной причине легче поддающийся решению, после чего на стадии “властвования” решается полученный в результате преобразования экземпляр задачи.

Имеется три основных варианта этого метода, отличающихся способом пре­образования (рис. 6.1).

* Преобразование в более простой или более удобный для решения эк­земпляр той же задачи — упрощение экземпляра.
* Изменение представления имеющегося экземпляра задачи.
* Приведение задачи, т.е. преобразование к экземпляру другой задачи, для которой имеется алгоритм решения.

Более простой экземпляр или

Экземпляр

задачи

Иное представление ЯШФ> Решение или

Другой экземпляр задачи Рис. 6.1. Стратегия “преобразуй и властвуй”

В первых трех разделах данной главы мы встретимся с разными примера­ми упрощения экземпляра задачи. В разделе 6.1 мы познакомимся с простой, но плодотворной идеей предварительной сортировки. Многие задачи, связанные со списками, гораздо проще решить, если списки отсортированы. Естественно, преимущества от сортировки списков должны более чем компенсировать затра­ты времени на их сортировку; в противном случае лучше работать непосред­ственно с несортированными списками. В разделе 6.2 мы познакомимся с одним из важных алгоритмов прикладной математики — методом исключения Гаусса. Этот алгоритм предназначен для решения системы линейных уравнений путем их предварительного преобразования к другой системе линейных уравнений, обла­дающей специальными свойствами, позволяющими легко находить ее решение. В разделе 6.3 идея упрощения экземпляра и изменения представления применена к деревьям поиска. В результате мы получим AVL-деревья и многопутевые сба­лансированные деревья поиска; позже мы рассмотрим их простейший случай — 2-3-деревья.

В разделе 6.4 представлены пирамиды (кучи) и пирамидальная сортировка. Даже если вы уже знакомы с этой важной структурой данных и ее применением для сортировки, все равно стоит взглянуть на нее в новом свете метода преоб­разования. В разделе 6.5 мы обсудим схему Горнера — замечательный алгоритм для вычисления полиномов. Если бы существовал Зал славы алгоритмов, то схе­ма Горнера была бы одним из главных претендентов быть представленным там за свою элегантность и эффективность. Мы также рассмотрим два алгоритма для решения задачи возведения в степень, которые основаны на идее изменения представления.

Завершается глава обзором ряда применений третьего варианта метода преоб­разования — приведения задачи. Это наиболее радикальное преобразование, так как задача преобразуется в совершенно иную задачу. Это очень мощный метод, который активно используется в теории сложности (глава 10). Однако примене­ние этого метода для разработки практичных алгоритмов далеко не тривиально. Во-первых, мы должны определить новую задачу, в которую будет преобразо­вана исходная. Затем мы должны убедиться, что алгоритм преобразования, за которым следует алгоритм решения новой задачи, является более эффективным методом решения с точки зрения времени работы, чем другие алгоритмические альтернативы. Среди нескольких примеров мы рассмотрим важный частный слу­чай математического моделирования, т.е. выражения задачи в терминах чисто математических объектов — таких как переменные, функции и уравнения.

1. Предварительная сортировка

Предварительная сортировка — старая идея в кибернетике. На самом деле ин­терес к алгоритмам сортировки в значительной степени обусловлен именно тем, что ряд задач с участием списков решаются существенно проще, если списки от­сортированы. Понятно, что временная эффективность алгоритма, включающего в качестве этапа сортировку, может зависеть от эффективности использованно­го алгоритма сортировки. Для простоты в этом разделе мы полагаем, что все списки реализованы в виде массивов, поскольку многие алгоритмы сортировки реализуются проще при использовании этого представления.

Мы уже рассматривали три элементарных алгоритма сортировки — сортировку выбором, пузырьковую сортировку и сортировку вставкой, — которые квадратич­ны как в наихудшем, так и в среднем случае, и два более эффективных алгоритма — сортировку слиянием, эффективность которой в любом случае равна © (гг log гг), и быструю сортировку, эффективность которой в среднем случае также равна 0 (nlogn), но в худшем — квадратична. Имеются ли более быстрые алгоритмы сортировки? Как мы уже указывали в разделе 1.3 (см. также раздел 10.2), в общем случае ни один алгоритм сортировки, основанный на сравнении, не может иметь эффективность, превышающую nlogn в наихудшем или в среднем случае.[[43]](#footnote-43)

Далее приведены три примера использования предварительной сортировки. Дополнительные примеры можно найти в упражнениях к данному разделу.

Пример 1 (Проверка единственности элементов массива). Сама задача вам уже должна быть знакома — мы рассматривали алгоритм ее решения с приме­нением грубой силы в разделе 2.3 (пример 2). Алгоритм на основе грубой силы для проверки того, что все элементы массива различны, попарно сравнивает все элементы этого массива, пока не будут найдены два одинаковых либо пока не будут пересмотрены все возможные пары. В наихудшем случае эффективность такого алгоритма равна © (п2).

К решению задачи можно подойти и по-другому — сначала отсортировать массив, а затем сравнивать только последовательные элементы: если в массиве есть одинаковые элементы, то они должны следовать в отсортированном массиве один за другим.

Алгоритм *PresortElementUniqueness (А* [0..п — 1])

// Проверка единственности элементов массива

// Входные данные: Массив А[0..п — 1] упорядочиваемых элементов

// Выходные данные: true, если в А нет одинаковых элементов,

// и false, если есть

Сортировка массива **А for** г **<— 0 to п — 2 do if** А[г\ **=** А[г **+ 1] return false return true**

Время работы данного алгоритма представляет собой сумму времени, затра­ченного на сортировку, и времени на проверку соседних элементов. Поскольку для сортировки требуется как минимум п log п сравнений, а для проверки сосед­них элементов — не более п — 1, именно сортировка и определяет общую эф­фективность алгоритма. Так, если мы используем здесь квадратичный алгоритм сортировки, то алгоритм в целом окажется не эффективнее метода грубой силы. Но если воспользоваться хорошим алгоритмом сортировки, таким как сортиров­ка слиянием, эффективность которого в худшем случае составляет © (nlogn), то весь алгоритм проверки единственности элементов массива также будет иметь эффективность © (n log v):

Т (п) = Tsort (п) + Tscan (п) Е © (n log п) + © (п) = © (n log п). ш

Пример 2 (Вычисление моды). Модой (mode) называется значение, которое встречается в данном списке чаще других. Например, в случае значений 5, 1,5, 7, 6, 5, 7 модой является значение 5 (если одинаково часто встречается несколько значений, модой может быть выбрано любое из них). Алгоритм на основе грубой силы сканирует весь список и вычисляет количество появлений в списке каждого из различных значений, после чего ищется наибольшее из найденных количеств появлений в списке. При реализации такого подхода встреченные значения и ко­личество их появлений можно хранить в отдельном списке. При каждой итерации г-ый элемент исходного списка сравнивается со значениями уже встречавших­ся элементов путем сканирования вспомогательного списка. Если значение г-го элемента имеется во вспомогательном списке, увеличивается счетчик количества элементов; если — нет, элемент добавляется во вспомогательный список, а его счетчику присваивается значение 1.

Нетрудно увидеть, что в наихудшем случае входные данные представляют собой список из неповторяющихся элементов. В таком списке его г-ый элемент сравнивается с г — 1 различными элементами вспомогательного списка, перед тем как быть добавленным к этому списку. В результате количество сравнений, выполняемых алгоритмом в наихудшем случае, при создании вспомогательного списка составляет

С (п) = ^2 (г — 1) = 0 + Ц Ь (п - 1) = Е © (п2) .

г**=1**

Кроме того, для выявления элемента с наибольшим значением счетчика во вспо­могательном списке требуется выполнить п — 1 сравнений, но они не влияют на квадратичность рассмотренного алгоритма в наихудшем случае.

Рассмотрим альтернативный вариант, начинающийся с сортировки списка. В таком случае все равные значения будут соседствовать друг с другом, и для вычисления моды надо только найти наибольшую подпоследовательность одина­ковых соседних значений в отсортированном списке.

Алгоритм PresortMode (А [0..п - 1])

// Вычисляет моду массива с использованием // предварительной сортировки

// Входные данные: Массив А[0..п — 1] упорядочиваемых элементов

// Выходные данные: Мода массива Сортировка массива А

г <— О II Текущее сканирование начинается с позиции г

mode frequency <—0 // Максимальное количество одинаковых элементов

while г ^ п — 1 do

***runlength*** <— **1**; ***runvalue*** <— ***А[г]***

while г + ***runlength*** ^ n — **1** and ***A[i*** + ***runlength] = runvalue*** do ***runlength = runlength*** + **1** if ***runlength*** > ***modefrequency***

***modefrequency*** <— ***runlength\ modevalue*** <— ***runvalue i <— i + runlength*** return ***modevalue***

Анализ этого алгоритма аналогичен анализу алгоритма из примера 1: время ра­боты алгоритма определяется временем сортировки, поскольку время выполнения остальной части алгоритма — линейное (почему?). Следовательно, при использо­вании сортировки, принадлежащей классу эффективности п log п, эффективность описанного алгоритма в наихудшем случае будет выше эффективности в наихуд­шем случае алгоритма с использованием грубой силы. ■

Пример 3 (Поиск). Рассмотрим поиск данного значения v в массиве из п упо­рядочиваемых элементов. Решение с использованием грубой силы — последова­тельный поиск (см. раздел 3.1) — требует в наихудшем случае п сравнений. Если массив предварительно отсортировать, то применение бинарного поиска приведет к [log2 пJ + 1 сравнений в наихудшем случае. Даже при использовании макси­мально эффективной сортировки класса nlogn общее время работы алгоритма в наихудшем случае составит

Т (п) = Tsort (п) + Tsearch (п) = 0 (n log п) + © (log п) = 0 (n log п),

что хуже, чем в случае последовательного поиска. То же самое можно сказать и об эффективности в среднем случае. Конечно, если поиск требуется выполнять неоднократно, то затраты времени на сортировку могут оказаться оправданны­ми (в упражнении 6.1.6 требуется оценить наименьшее количество поисков, при котором окупается предварительная сортировка). ■

Перед тем как завершить обсуждение предварительной сортировки, мы долж­ны упомянуть, что многие (если не большинство) геометрические алгоритмы работают с множествами точек, отсортированных тем или иным образом. Точки могут быть отсортированы по одной из координат, их расстоянию от некоторой линии, некоторому углу и т.д. Например, предварительная сортировка использу­ется в алгоритме декомпозиции для задачи пар ближайших точек и в алгоритме вычисления выпуклой оболочки, рассматривавшихся в разделе 4.6.

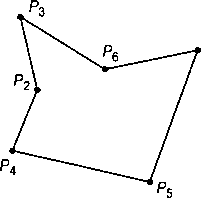
**Упражнения 6.1**

1. Покажите, что эффективность в среднем случае при однократном по­иске при помощи алгоритма, состоящего из наиболее эффективного алгоритма сортировки на основе сравнений, за которым следует би­нарный поиск, оказывается ниже эффективности последовательного поиска в среднем случае.
2. Оцените, какое количество поисков следует выполнить, чтобы оправ­дать время, затраченное на предварительную сортировку массива из 103 элементов, если она выполняется при помощи сортировки слия­нием, а поиск — с использованием бинарного поиска (для простоты считаем, что выполняется поиск элементов, о которых заведомо из­вестно их наличие в массиве). А в случае массива из 10[[44]](#footnote-44) элементов?
3. Сортировать или не сортировать? Разработайте эффективные алгорит­мы для решения следующих задач и определите их классы эффектив­ности.

а) У вас имеется п телефонных счетов и т чеков для их оплаты п ^ га). Считая, что номера телефонов указаны на чеках, надо найти всех должников (для простоты считаем, что для каждого счета выписано не более одного чека и что выписанный чек полностью оплачивает счет.)

б) Имеется файл с п записями о студентах, в которых указаны номер, имя, адрес и дата рождения студента. Требуется найти количество студентов из каждого штата.

1. Дано множество из п ^ 3 точек на плоскости х — у. Требуется соеди­нить их замкнутой ломаной линией без самопересечений так, чтобы получился многоугольник, например:



5

•р,

а) Всегда ли поставленная задача имеет решение? Всегда ли это реше­ние единственно?

б) Разработайте эффективный алгоритм для решения поставленной за­дачи и определите его класс эффективности.

1. У вас есть массив из п чисел и целое число s. Определите, имеются ли в массиве два числа, сумма которых равна s. (Например, в случае массива 5, 9, 1, 3 и s = б ответ — “да”, но если s = 7, ответ — “нет”.) Разработайте алгоритм для решения поставленной задачи, эффектив­ность которого превышает квадратичную.

image114

1. а) Разработайте эффективный алгоритм для поиска всех множеств ана­грамм в большом файле, как, например, словарь английских слов [15]. Например, eat, ate и tea принадлежат к одному такому мно­жеству.

б) Напишите программу, реализующую ваш алгоритм.

1. Метод исключения Гаусса

Вы наверняка знакомы с системой из двух линейных уравнений с двумя неиз­вестными:

ОцЖ + 0122/ = £>1

**С21Х +** а22У **=** ь2

Вспомним, что если только коэффициенты одного уравнения не пропорциональны коэффициентам другого, то система имеет единственное решение. Стандартный метод поиска этого решения состоит в использовании одного из уравнений для того, чтобы выразить одну переменную как функцию другой, а затем подставить результат в другое уравнение. Это дает линейное уравнение относительно одной переменной, решение которого позволяет найти значение второй переменной.

Во многих приложениях требуется решить систему из п уравнений с п неиз­вестными, где п — большое число:

а11х1 + &12х2 + \* \* \* + СЬ\пхп — Ь\ а21х1 + а22х2 + \* \* \* + П-2пхп — Ь*'2*

***TL***

**anlxl** "h **dn2x2** ”h \* \* ’ **0"ппхп** —

Теоретически такую систему уравнений можно решить, обобщив метод подста­новки, примененный для решения системы из двух линейных уравнений (на каком методе разработки алгоритмов основан этот способ?), но полученный в результате алгоритм будет слишком громоздким.

К счастью, имеется гораздо более элегантный алгоритм решения систем ли­нейных уравнений, который называется методом исключения Гаусса (Gauss eli­mination)[[45]](#footnote-45). Идея метода заключается в преобразовании системы п линейных урав­нений с п неизвестными в эквивалентную систему (т.е. систему с тем же решени­ем, что и у исходной) с верхнетреугольной матрицей коэффициентов, т.е. такой, у которой все элементы ниже главной диагонали равны нулю:

**а\\Х\** + 0-12X2 + \* • \* + **Q'inXn** — **Ь\ а!цХ1** + 0-12^2 + \* \* \* + **ainxn =** ^1

0-21XI + 022X2 + \* \* \* + 02п^п — &2 а22х2 + ’ ’ \* + а2пхп = ^2

**Onl^l “Г Оп2^2 “Ь \* \* \* “Г** апп%п = Ьп °'ппхп **=**

Используя матричные обозначения, можно записать это как

***Ах — b => А! х*** = ***У***

где

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Оц | 012 • • | . . CLln |  | V |  | °п | 012 • 1 | ■ • °'\п |  | X |
| А = | 021 | 022 • | ■ ■ 02п | Ъ = | ъ2 | А' = | 0 | °22 • | •• а2п | ь' = | ъ'2 |
|  | \_Оп1 | О 7г 2 • | • • опп\_ |  | Ьп\_ |  | \_ 0 | 0 . | ■ ■ апп. |  | X |

(Мы добавили штрихи к элементам матрицы и свободным членам новой систе­мы линейных уравнений для того, чтобы подчеркнуть отличие этих значений от значений их аналогов в исходной системе линейных уравнений.)

Чем же система линейных уравнений с верхнетреугольной матрицей коэффи­циентов лучше системы линейных уравнений с произвольной матрицей? Дело в том, что систему линейных уравнений с верхнетреугольной матрицей легко решить методом обратной подстановки следующим образом. Сначала мы вы­числяем значение хп из последнего уравнения; затем подставляем полученное значение в предпоследнее уравнение и получаем значение xn\_i. Продолжая вы­полнять подстановки вычисленных значений переменных в очередные уравнения, мы получим значения всех п переменных — от хп до х\.

Итак, каким же образом можно получить из системы линейных уравнений с произвольной матрицей коэффициентов А эквивалентную систему линейных уравнений с верхнетреугольной матрицей А'1 Это можно сделать при помощи последовательности так называемых элементарных операций:

* обмена двух уравнений системы линейных уравнений;
* умножения уравнения на ненулевую величину;
* замены уравнения на сумму или разность этого уравнения и другого уравнения, умноженного на некоторую величину.

Поскольку ни одна из элементарных операций не изменяет решение системы линейных уравнений, любая система линейных уравнений, полученная из исход­ной при помощи серии элементарных операций, будет иметь то же решение, что и исходная система линейных уравнений. Теперь посмотрим, как получить систе­му линейных уравнений с верхнетреугольной матрицей. Для начала используем в качестве опорного элемента ац для того, чтобы сделать все коэффициенты при х\ в строках ниже первой нулевыми. В частности, заменим второе уравнение раз­ностью между ним и первым уравнением, умноженным на a2i/an для того, чтобы получить нулевой коэффициент при х\. Выполняя то же для третьей, четвертой и далее строк и умножая первое уравнение, соответственно, на a3i/an, ац/ац, ..., ani/an, сделаем все коэффициенты при х\ в уравнениях ниже первого рав­ными 0. Затем обнулим все коэффициенты при Х2 в уравнениях ниже второго, вычитая из каждого из этих уравнений второе, умноженное на соответствующий коэффициент. Повторяя эти действия для каждой из первых п — 1 строк, получим систему линейных уравнений с верхнетреугольной матрицей коэффициентов.

Перед тем как рассмотреть конкретный пример использования метода Гаусса, заметим, что можно работать только с матрицей коэффициентов, к которой в каче­стве п + 1-го столбца добавлены свободные члены системы линейных уравнений. Другими словами, нет необходимости явно использовать имена переменных си­стемы линейных уравнений или знаки + и =.

**Пример 1. (Решение системы линейных уравнений методом исключения Гаусса).**

2xi - х2 + х3 = 1 4xi + Х2 - х3 = 5



2-111 4 1-15

**1110**

Строка 2 — | • Строка 1 Строка 3 — \ • Строка 1

XI + Х2 + хз = О

**2-111 0** 3-33

0 0 2 -2

Теперь при помощи обратной подстановки легко получить хз = (—2) /2 = — 1, = (3 — (-3) хз)/3 = 0 и xi = (1 - хз - (-1) х2)/2 = 1. ■

Далее приведен псевдокод этапа исключения алгоритма решения систем ли­нейных уравнений методом исключения Гаусса.

АЛГОРИТМ Gauss Elimination (A [l..n, l..n], b [l..n])

// Применение метода исключения Гаусса к матрице // коэффициентов системы линейных уравнений А, объединяемой // со столбцом свободных членов b

II Входные данные: Матрица A[l..n, 1..п] и вектор Ь[1..п]

// Выходные данные: Эквивалентная верхнетреугольная матрица // на месте матрицы А со значениями

// в п + 1-ом столбце, соответствующим

// свободным членам новой системы линейных

// уравнений

for г <— 1 to п do

A[i, п + 1] <— 6[г] // Расширение матрицы

for г <— 1 to п — 1 do for j <— г + 1 to гг do for к <— г to п + 1 do

***A\j,*** /с] <- ***A[j, к] - A[i, к] \* A\j,*** г]/А[г, г]

По поводу приведенного псевдокода следует сделать два важных замечания. Во-первых, он не всегда корректен: если А [г, г] = 0, нельзя выполнить деление на этот элемент и, следовательно, использовать г-ую строку в качестве опорной на г-ой итерации алгоритма. В этом случае мы должны воспользоваться пер­вой из элементарных операций и обменять г-ую строку с одной из строк ниже ее, у которой в г-ом столбце находится ненулевой элемент (если система линей­ных уравнений имеет единственное решение (что является нормальной ситуа­цией при рассмотрении систем линейных уравнений), то такая строка должна существовать).

Поскольку мы все равно должны быть готовы к возможному обмену строк, сле­дует позаботиться и о другой потенциальной сложности: возможности того, что величина А [г, г] будет столь мала (и, соответственно, столь велик коэффициент A [j, г]/А [г, г]), что новое значение A [j, к] может оказаться искаженным ошибкой округления, связанной с вычитанием двух сильно отличающихся чисел.[[46]](#footnote-46) Чтобы избежать этой проблемы, можно всегда выбирать строку с наибольшим абсолют­ным значением коэффициента в г-ом столбце для обмена с г-ой строкой, а затем использовать ее в качестве опорной на г-ой итерации. Такая модификация алго­ритма, называющаяся выбором ведущего элемента (partial pivoting), гарантирует, что значение масштабирующего множителя никогда не превысит 1.

Во-вторых, заметим, что внутренний цикл написан с вопиющей неэффектив­ностью. Можете ли вы, не обращаясь к приведенному далее псевдокоду, сказать, в чем именно заключается эта неэффективность и как ее избежать?

**АЛГОРИТМ *BetterGaussElimination*** **(*A*** **[l..n, l..n]**, ***Ъ*** [1..п])

// Реализует метод исключения Гаусса с выбором ведущего // элемента

// Входные данные: Матрица А[1..п, 1..п] и вектор Ь[1..п]

// Выходные данные: Эквивалентная верхнетреугольная матрица // на месте матрицы А со значениями

// вп + 1-ом столбце, соответствующим

// свободным членам новой системы линейных

// уравнений

for г <— 1 to п do

А [г, п 1] <— 6 [г] // Расширение матрицы

for г <— 1 to п — 1 do pivotrow <— г for j <— г + 1 to п do

>| A\pivotrow, г] | pivotrow <— j for к <— г to n + 1 do

swap(A[i, k}. A\pivotrow, k]) for j <— г + 1 to n do temp <— A\j, г]/А[г, г] for к <— г to n + 1 do

A\j, fc] <— A\j, A:] — A[i, fc] \* temp

Давайте определим временную эффективность этого алгоритма. Наиболее глу­боко вложенный цикл состоит из одной строки A[j, к] <— A[j. к] — A[i, fc] \* temp которая содержит одну операцию умножения и одну — вычитания. На большин­стве компьютеров умножение, несомненно, более дорогостоящая операция, чем сложение и вычитание, так что именно умножение рассматривается как базовая операция данного алгоритма.[[47]](#footnote-47) Напомним, что в разделе 2.3 (см. также прило­жение А) приведены стандартные формулы суммирования, которые будут очень полезны для понимания приведенных далее выкладок:

71 — 1 71 71+1 71 — 1 71 71—1 71

C(n) = Yl £ = £ 2 (rc + l-\* + l) = £ ^ (п + 2 — г) =

г=1 j=i+l k=i i=1 j = г-h 1 г=1 j=i+l

71 — 1 71—1

= (n + 2 - г) (n — (г + 1) + 1) = (n + 2 — г) (n — г) =

г=1 г=1

71 — 1

= (n + 1) (гг — 1) + гг (гг — 2) + • • • + 3 • 1 = ^ ^ (j + 2) j —

з=i

-2 (гг — 1) тг (2тг — 1) (гг —1)тг

= Е-> +Е2.> = -—^2 =

j=i j=i

= п(п-1)(2п + 5)и1пз€в(п,^

Поскольку временная эффективность второй стадии (обратной подстановки) алгоритма исключения Гаусса равна © (тг2) (что требуется самостоятельно пока­зать в упражнении 6.2.5), общее время работы алгоритма определяется домини­рующим кубическим временем стадии исключения, так что алгоритм исключения Гаусса — кубический.

Теоретически метод исключения Гаусса всегда либо дает точное решение си­стемы линейных уравнений (если она имеет единственное решение), либо выяс­няет, что такого решения -не существует. В последнем случае система линейных уравнений может либо не иметь решения вовсе, либо иметь бесконечно много решений. На практике решение систем большого размера данным методом на­талкивается на трудности, в первую очередь связанные с накоплением ошибок округления (см. раздел 10.4). Обратитесь к учебникам по численному анализу, где этот вопрос рассматривается более подробно, как для данного метода решения систем линейных уравнений, так и для других реализаций.

**LU-разложение и другие приложения**

Метод исключения Гаусса имеет интересный и очень полезный побочный ре­зультат, именуемый LU-разложением, или LU-декампозицией (LU-decomposition) матрицы коэффициентов. На деле современные коммерческие реализации мето­да исключения Гаусса основаны именно на этом разложении, а не на описанном ранее алгоритме.

**значения, поскольку нас интересует количество выполнений внутреннего цикла (которое, конечно же, в данном случае совпадает с количеством выполняемых умножений и вычитаний).**

Пример 2. Вернемся к примеру в начале этого раздела, когда мы применили метод исключения Гаусса к матрице

***А =***

2-11 4 1-1

**111**

Рассмотрим нижнетреугольную матрицу L, образованную единицами на главной диагонали и множителями, вычисляемыми в процессе исключения Гаусса для обнуления соответствующих коэффициентов в строках матрицы:

***L —***

**1** **0** **0 2** **1** **0** 1/2 1/2 1

и верхнетреугольную матрицу U, представляющую собой результат исключения

U =

**2-11 0** 3-3

0 0 2

Оказывается, произведение LU этих матриц равно исходной матрице А (для рас­сматриваемых матриц L и 27 это можно проверить непосредственным умножени­ем, но в общем случае этот факт, естественно, требует доказательства, которое мы опускаем).

Следовательно, решение системы линейных уравнений Ах = b эквивалентно решению системы LUx = Ъ. Решить ее можно следующим образом. Обозначим у = Uх, тогда Ly = b. Сначала решим систему Ly = b, что очень просто сделать, поскольку L — нижнетреугольная матрица. Затем решим систему Ux = у, что опять же несложно, поскольку U — верхнетреугольная матрица. Так, для системы в начале данного раздела мы сначала решаем уравнение Ly = Ь:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 2/1 |  | Y |
|  | 2/2 | = | 5 |
|  | .2/3. |  | 0 |

**1** **0** **0 2** **10** 1/2 1/2 1

Ее решение —

2/т = 1, У2 = 5 - 2yi = 3, уз = 0 - ~j/г - iy2 = -2.

Затем надо решить уравнение Uу = х, т.е. матричное уравнение

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1  г—1  г—1 1  tCN |  | Xl |  | ' 1 ' |
| 0 3-3 |  | Х2 | = | 3 |
| 00 2 |  | ,хз\_ |  | -2 |

Его решение —

х3 = (—2)/2 = -1, х2 = (3 - (-3) х3)/3 = О,

**XI = (1 -х3- (-1)х2)/2 = 1. в**

Заметим, что получив LU-разложение матрицы А, мы можем решать системы линейных уравнений Ах = Ь для разных векторов свободных членов Ь. Это глав­ное преимущество метода LU-разложения по сравнению с классическим методом исключения Гаусса, описанным ранее. Заметим также, что L[/-разложение не тре­бует дополнительной памяти, поскольку ненулевую часть матрицы U мы можем хранить в верхнетреугольной части матрицы А (включая главную диагональ), а нетривиальную часть матрицы L — ниже главной диагонали А.

**Вычисление обратной матрицы**

Метод исключения Гаусса — очень полезный алгоритм, с помощью которого можно решить одну из наиболее важных задач прикладной математики — систе­мы линейных уравнений. Метод исключения Гаусса может быть также применен и к некоторым другим задачам линейной алгебры, таким как вычисление обрат­ной матрицы (matrix inverse). Обратная к матрице А размером пхп матрица также имеет размер пхп и обозначается как А~1:

***AA~l*** = ***I,***

где I — единичная матрица размером пхп (т.е. матрица, все элементы которой равны 0, за исключением элементов на главной диагонали, которые равны 1). Не каждая квадратная матрица имеет обратную, но если у данной матрицы есть обратная, то она — единственная. Матрица А, не имеющая обратной матрицы, на­зывается сингулярной (singular). Можно доказать, что матрица сингулярна тогда и только тогда, когда одна из ее строк представляет собой линейную комбинацию (сумму умноженных на некоторые величины) других строк. Удобным способом проверить, является ли данная матрица не сингулярной, — применить к ней ме­тод исключения Гаусса: если он даст верхнетреугольную матрицу с ненулевыми элементами на главной диагонали, матрица не сингулярна; в противном случае исходная матрица сингулярна. Сингулярные матрицы — очень специфичный част­ный случай, и большинство квадратных матриц имеют обратные.

Теоретически обратные матрицы очень важны, поскольку играют роль об­ратных величин в матричной алгебре, тем самым преодолевая отсутствие явной операции деления матриц. Например, аналогично линейному уравнению с одним неизвестным ах = Ъ, решение которого х = а~гЬ (если а не равно 0), мы мо­жем записать решение системы п линейных уравнений с п неизвестными Ах = b как х = А~1Ь (если А не сингулярная матрица), где Ь, разумеется, — не число, а вектор.

В соответствии с определением обратной матрицы, для того, чтобы найти ее для не сингулярной матрицы А размером п х тг, требуется найти п2 чисел 1 таких, что

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ац | а\2 .. | «1 п |  | Хц | Xu • - | Х\п |  | "1 | 0 .. | .. о' |
| а 21 | «22 • - | «2 п |  | Х21 | Х22 • | Х2 п | = | 0 | 1 .. | .. 0 |
| \_Р"п1 | ап 2 | «пп\_ |  | Хп1 | Хп2 | Хпп\_ |  | 0 | 0 . | .. 1\_ |

Мы можем найти неизвестные числа, решая п систем линейных уравнений с од­ной и той же матрицей коэффициентов А, у которых вектор неизвестных х-7 представляет собой j-ый столбец обратной матрицы, а вектор свободных членов eJ — j-ый столбец единичной матрицы (1 ^ j ^ тг):

***Ахi — е?***.

Эти системы линейных уравнений можно решить, применяя метод исключения Гаусса к матрице А, расширенной добавлением к ней единичной матрицы разме­ром тг х тг. Еще лучший способ — использовать метод исключения Гаусса для поис­ка LU-разложения матрицы А и решать системы линейных уравнений LUxi = е-7, j = 1,2,..., тг, как описывалось ранее.

**Вычисление определителя**

Еще одна задача, которая может быть решена при помощи метода исключе­ния Гаусса, — это вычисление определителя, или детерминанта (determinant) матрицы. Определителем матрицы А размером тг х тг, обозначаемым det А или \А\, является число, которое можно рекурсивно определить следующим образом. Если тг = 1, т.е. А состоит из единственного элемента ац, то det А = ац. Если тг > 1, то det А вычисляется по рекурсивной формуле

**гг**

det А = Sja\j det Aj, з=i

где Sj равно +1, если j нечетно, и —1, если j четно (т.е. Sj = (—1)J'+1), aij — элемент на пересечении первой строки и j-ro столбца матрицы, а Aj — матрицаразмером (гг — 1) х (гг — 1), полученная из матрицы А удалением первой строки и j-ro столбца.

В частности, для матрицы размером 2 х 2 из определения вытекает следующая легко запоминаемая формула:

**ап а**\2 **«21** **«22**

det

= ац det [0-22] — «12 det [«21] — «п«22 — «12«21-

Другими словами, определитель матрицы размером 2x2 равен разности произ­ведений ее диагональных элементов.

В случае матрицы размером 3x3 получаем

«22 «23 «32 «зз

**«21** **«22**

«31 «32

«21 «23 «31 «33

det

— d\\ det

— &\2 det

+ ахз det

**«11** **«12** «13 **«21** **«22** **«23**

«31 «32 «зз

— «11«22«33 + «12«23«31 + «21«32«13 “ «31«22«13~“

— «21«12«33 ~ «32«23«11 •

Кстати, эта формула очень удобна для применения в ряде приложений; в частно­сти, мы уже воспользовались ею в разделе 4.6 в алгоритме быстрой оболочки.

Но что если требуется вычислить определитель большой матрицы? (Хотя на практике это требуется не так часто, тем не менее эта задача стоит того, чтобы ее рассмотреть.) Рекурсивное определение мало чем может помочь, поскольку оно приводит к вычислению суммы п\ членов. И здесь опять нас спасает метод исключения Гаусса. Основная идея заключается в том, что определитель верхне­треугольной матрицы равен произведению ее элементов на главной диагонали, и легко понять, как именно влияют на значение определителя элементарные опе­рации, выполняемые алгоритмом (они либо оставляют его значение неизменным, либо меняют знак, либо приводят к умножению его на константу, использующу­юся в алгоритме исключения Гаусса). В результате мы можем вычислить опреде­литель матрицы размером п х п за кубическое время.

det А,

Определители играют важную роль в теории систем линейных уравнений. В частности, система из п линейных уравнений с п неизвестными Ах = b имеет единственное решение тогда и только тогда, когда определитель матрицы коэффи­циентов det А не равен 0. Более того, решение можно найти по формуле, носящей название правила Крамера (Cramer’s rule):

det Ап

det А\

***xi =***

хп —

det A,"',Xj det a’" ' ,^n det^ ’ где det A3 — определитель матрицы, полученной путем замены j-ro столбца мат­рицы А столбцом Ь. (В упражнении 6.2.10 требуется выяснить, насколько хорошим алгоритмом для решения систем линейных уравнений является правило Крамера.)

1. Решите методом исключения Гаусса следующую систему линейных уравнений:

х\ + Х2 + хг = 2 2xi + Х2 + хз = 3 XI- Х2 + Зх3 = 8

1. а) Решите систему линейных уравнений из упражнения 1 методом LU-

разложения.

б) Как можно классифицировать метод LU-разложения с точки зрения методов разработки алгоритмов?

1. Решите систему линейных уравнений из упражнения 1 путем обраще­ния матрицы коэффициентов с последующим умножением на вектор свободных членов.
2. Насколько корректен следующий вывод класса эффективности стадии исключения метода исключения Гаусса?

**п—1** **п** п+1 **п—1**

[с (п) = Л 2 £1 = 2 (п+2 \_ - \*)=](#bookmark196)

**г=1 *j=i+l k=i*** **г=1**

**гг—1**

= ((п + 2) п — г (2п + 2) + г2) =

**1=1**

**гг—1 гг—1 гг—1**

= (п + 2) п — (2тг + 2) г + г2.

**г= 1 г=1 г=1**

Поскольку 51 (n) = (п + 2) ne© (n3)> 52 (п) = Yli=l (2п + 2) ^

Е © (гг3) и 53 (гг) = е ® (n3) > то 5i (гг) - 52 (гг) + 53 (гг) Е © (гг3).

1. Напишите псевдокод стадии обратной подстановки метода исключения Гаусса и покажите, что время его работы равно © (гг2).
2. Предположим, деление двух чисел выполняется в три раза дольше их умножения. Оцените, насколько в таком случае алгоритм BetterGauss- Elimination работает быстрее алгоритма Gauss Elimination (есте­ственно, мы считаем, что компилятор не слишком интеллектуален и не устраняет неэффективность в GaussElimination самостоятельно).
3. а) Приведите пример системы двух линейных уравнений с двумя неиз­

вестными, которая имеет единственное решение, и решите ее мето­дом исключения Гаусса.

б) Приведите пример системы двух линейных уравнений с двумя неиз­вестными, которая не имеет решения, и примените к ней метод ис­ключения Гаусса.

в) Приведите пример системы двух линейных уравнений с двумя неиз­вестными, которая имеет бесконечное количество решений, и при­мените к ней метод исключения Гаусса.

1. Метод исключения Гаусса-Джордана (Gauss-Jordan elimination) от­личается от метода исключения Гаусса тем, что все элементы над глав­ной диагональю делаются нулевыми в то же время и с использованием той же опорной строки, что и элементы под главной диагональю.

а) Примените метод исключения Гаусса-Джордана к системе линей­ных уравнений из упражнения 1.

б) На какой общей стратегии разработки основан этот алгоритм?

в) Сколько умножений в общем случае выполняет данный алгоритм при решении системы п линейных уравнений с п неизвестными? Сравните это количество с числом умножений, выполняющихся при использовании метода исключения Гаусса (как на стадии исключе­ния, так и на стадии обратной подстановки).

1. Система п линейных уравнений с п неизвестными Ах = b имеет един­ственное решение тогда и только тогда, когда det А ф 0. Имеет ли смысл проверять выполнимость этого условия перед применением ме­тода исключения Гаусса?
2. а) Примените правило Крамера для решения системы линейных урав­нений из упражнения 1.

б) Оцените, во сколько раз дольше решается система из п линейных уравнений с п неизвестными по правилу Крамера по сравнению с методом исключения Гаусса (считаем, что все определители в фор­муле Крамера вычисляются независимо с применением метода ис­ключения Гаусса).

1. Сбалансированные деревья поиска

В разделах 1.4 и 4.4 мы рассматривали бинарные деревья поиска — одну из важнейших структур данных для реализации словарей. Бинарное дерево поиска — это бинарное дерево, узлы которого содержат элементы множества упорядочивае­мых элементов, по одному элементу в узле, причем все элементы в левом поддере­ве меньше элемента в корне поддерева, а элементы в правом поддереве — больше него. Отметим, что такое преобразование множества в бинарное дерево поиска представляет собой пример метода изменения представления. Чего мы достигаем таким преобразованием по сравнению с простой реализацией словаря, например, при помощи массива? Мы получаем более высокую эффективность поиска, встав­ки и удаления — время выполнения всех этих операций равно © (log тг), но только в среднем случае. В наихудшем случае эти операции выполняются за время © (тг), поскольку дерево может выродиться в полностью несбалансированное, с высотой, равной тг — 1.

Ученые в области кибернетики затратили массу усилий в попытках найти структуру, которая сохраняет важные свойства классических бинарных деревьев поиска, — в первую очередь логарифмическую эффективность словарных опе­раций и отсортированность элементов, — но при этом избегает вырожденное™ в наихудшем случае. Для этого используются два подхода.

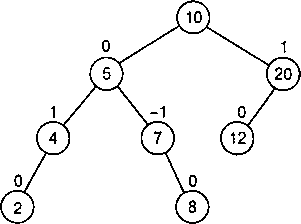
* Первый подход представляет собой вариант упрощения экземпляра за­дачи — несбалансированное бинарное дерево поиска преобразуется в сбалансированное. Конкретные реализации этой идеи различаются по их определениям того, что такое сбалансированность. AVL-depeeo (AVL tree) требует, чтобы разница высот левого и правого поддеревьев каждого узла не превышала 1. Красно-черное дерево (red-black tree) допускает, чтобы высота одного поддерева была в два раза больше высоты другого поддерева того же самого узла. Если вставка нового узла или удаление имеющегося приводит к тому, что нарушается усло­вие сбалансированности, такое дерево перестраивается при помощи одного из семейства специальных преобразований, которые называют­ся поворотами (rotation) и которые восстанавливают условия сбалан­сированности. (В этом разделе мы рассмотрим только AVL-деревья. Информацию о других типах бинарных деревьев поиска, которые ис­пользуют идею балансировки посредством поворотов, включая красно­черные деревья и так называемые косые деревья (splay tree), можно найти в соответствующей литературе.)
* Второй подход представляет собой вариант изменения представления: допускается наличие более чем одного элемента в узле дерева поис­ка. Частными случаями таких деревьев являются 2-3-деревья, 2-3-4- деревья и более общий и важный случай — В-деревья. Они различа­ются количеством элементов, которые допустимы в одном узле дерева поиска, но все они являются идеально сбалансированными. (Здесь мы рассмотрим простейший вид таких деревьев, а именно 2-3-деревья, отложив рассмотрение В-деревьев до главы 7.)

**AVL-деревья**

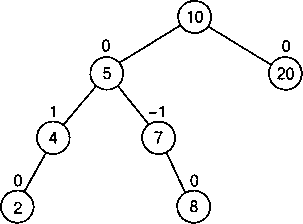
AVL-деревья были открыты в 1962 г. двумя советскими математиками — Г.М. Адельсон-Вельским и Е.М. Ландисом [1]; изобретенная структура получила название по первым буквам их фамилий.

Определение 1. AVL-дерево представляет собой бинарное дерево поиска, в ко­тором показатель сбалансированности (balance factor) каждого узла, определя­емый как разность высот левого и правого поддеревьев узла, равен 0, +1 или —1 (высота пустого дерева считается равной —1). ■

Например, бинарное дерево поиска на рис. 6.2а — AVL-дерево, в то время как бинарное дерево поиска на рис. 6.2б таковым не является.



1 2



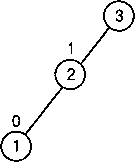
а) б)

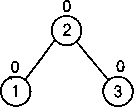
Рис. 6.2. а) AVL-дерево. б) Бинарное дерево поиска, не являющееся AVL-дере-

вом. Показатели сбалансированности показаны над узлами деревьев

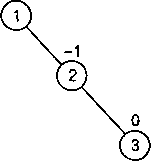
Если вставка нового узла делает AVL-дерево несбалансированным, оно пре­образуется при помощи поворота. Поворот в AVL-дереве представляет собой локальное преобразование поддерева, корень которого имеет показатель сбалан­сированности, равный +2 или —2; если таких узлов несколько, мы поворачиваем дерево с несбалансированным корнем, который наиболее близок к вновь встав­ленному листу. Всего имеется только четыре типа поворотов, причем два из них представляют собой зеркальное отражение двух других. В простейшей форме четыре возможных поворота показаны на рис. 6.3.

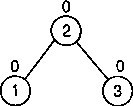
Первый тип поворота — одиночный правый поворот (single right rotation), или R-поворот (Д-rotation) (представьте поворот ребра, связывающего корень и его левый дочерний узел в бинарном дереве на рис. 6.3а, вправо). На рис. 6.4 одиночный Д-поворот показан в наиболее общем виде. Обратите внимание, что такой поворот выполняется после вставки нового ключа в левое поддерево левого дочернего узла корня, который перед вставкой имел показатель сбалансированно­сти +1.



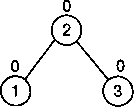


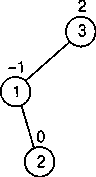
а)



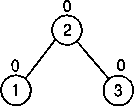


б)





в)



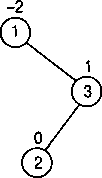


Рис. 6.3. Четыре типа поворотов AVL-дере- вьев с тремя узлами, а) Одиночный Я-поворот.

б) Одиночный L-поворот. в) Двойной LR-пово­рот. г) Двойной ЯЯ-поворот

Симметричный ему одиночный левый поворот (single left rotation), или L- поворот (L-rotation), представляет собой зеркальное отражение одиночного R- поворота. Он выполняется после вставки нового ключа в правое поддерево пра­вого дочернего узла корня, который перед вставкой имел показатель сбаланси-

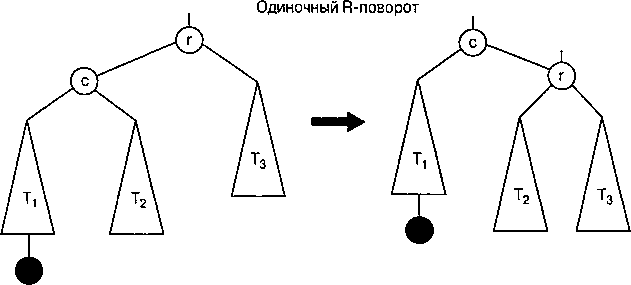


Рис. 6.4. Общий вид Я-поворота в AVL-дереве. Последний вставлен­ный узел выделен штриховкой

рованности —1 (в упражнениях к данному разделу имеется задание изобразить диаграмму одиночного L-поворота в общем виде).

Второй тип поворота — двойной лево-правый поворот (double left-right ro­tation), или LR-поворот (LjR-rotation). Он представляет собой объединение двух поворотов: выполняется L-поворот левого поддерева корня г, за которым следует Я-поворот нового поддерева, корнем которого является г (рис. 6.5). Он выполняет­ся после вставки нового ключа в правое поддерево левого дочернего узла дерева, корень которого перед вставкой имеет показатель сбалансированности +1.

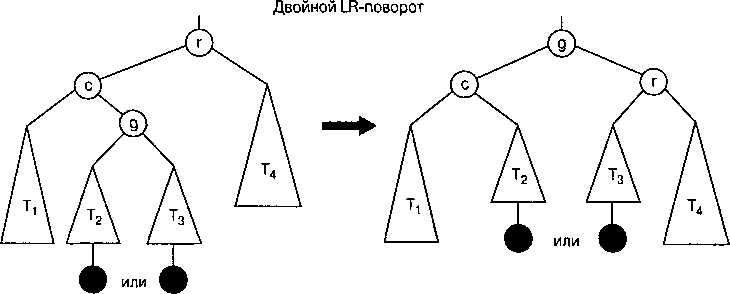


Рис. 6.5. Общий вид двойного ЯЯ-поворота в AVL-дереве. Последний вставлен­ный узел выделен штриховкой. Он может быть либо в левом, либо в правом поддереве “внука” корня

Двойной право-левый поворот (double right-left rotation), или RL-поворот ^L-rotation), представляет собой зеркальное отражение двойного ЯЯ-поворота и оставлен читателю в качестве самостоятельного упражнения.

Заметим, что повороты не являются тривиальными преобразованиями, хотя, к счастью, могут быть выполнены за постоянное время. Они должны не только гарантировать сбалансированность получающихся в результате поворотов дере­вьев, но и сохранять базовые требования к бинарным деревьям поиска. Например, в исходном дереве на рис. 6.4 все ключи поддерева Т\ меньше с, который меньше всех ключей поддерева Тг, которые, в свою очередь, меньше г, а тот — меньше всех ключей в поддереве Тз. То же отношение значений ключей сохраняется, как и требуется, и в сбалансированном дереве после выполнения поворота.

В качестве примера на рис. 6.6 показано построение AVL-дерева для задан­ного списка чисел. При отслеживании операций, выполняемых алгоритмом, не забывайте о том, что, если имеется несколько узлов с показателем баланса ±2, поворот выполняется для дерева, корнем которого является ближайший к вновь вставленному листу несбалансированный узел.

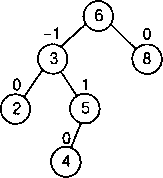
Насколько эффективны AVL-деревья? Как и для любого дерева поиска, кри­тической характеристикой является его высота. Можно вывести, что высота AVL- дерева и сверху, и снизу ограничена логарифмической функцией. В частности, высота h любого AVL-дерева с п узлами удовлетворяет неравенствам

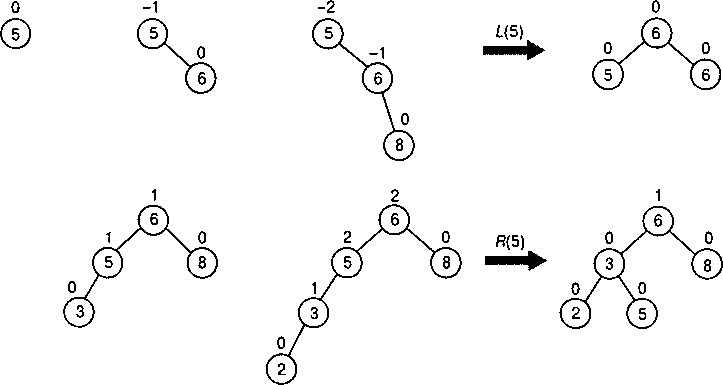
[log2 nj < h < 1.4405 log2 (n + 2) - 1.3277.

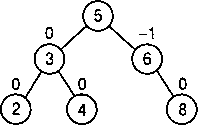
(Использованные в приведенной формуле константы — округления иррациональ­ных значений, связанных с числами Фибоначчи и золотым сечением — см. раз­дел 2.5.)

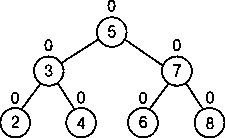
Непосредственно из приведенных неравенств следует, что операции поиска и вставки в худшем случае выполняются за время ©(logn). Получить точную формулу для средней высоты AVL-дерева, построенного для случайного набора ключей, достаточно сложно, но из многочисленных экспериментов известно, что средняя высота AVL-дерева составляет примерно 1.011 log2 п+0.1, за исключени­ем малых значений п [67]. Таким образом, поиск в AVL-дереве требует в среднем того же количества сравнений, что и поиск в отсортированном массиве при би­нарном поиске. Операция удаления ключа из AVL-дерева значительно сложнее, чем вставка, но, к счастью, она принадлежит к тому же классу эффективности, что и вставка, — т.е. к логарифмическому.

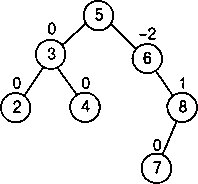
Однако эти впечатляющие характеристики эффективности достаются не за­даром. Недостатками AVL-деревьев являются частые повороты, необходимость поддержания сбалансированности узлов дерева и сложность, в особенности опе­рации удаления. Эти недостатки не позволили стать AVL-деревьям стандартной структурой данных для реализации словарей. В то же время лежащая в их основе идея о балансировке бинарного дерева поиска при помощи поворотов оказалась очень плодотворной и привела к открытию других интересных вариаций класси­ческих бинарных деревьев поиска.









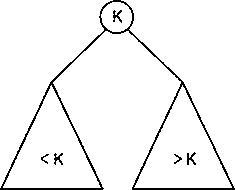


*RL(* 6

Рис. 6.6. Построение AVL-дерева путем последовательной вставки чисел из спис­ка 5, 6, 8, 3, 2, 4, 7. Число в скобках после указания типа поворота — корень перестраиваемого дерева

**2-3-деревья**

Как мы упоминали в начале этого раздела, вторая идея балансировки дере­вьев поиска заключается в том, чтобы позволить узлу одновременно содержать несколько ключей. Простейшей реализацией этой идеи являются 2-3-деревья, раз­работанные в 1970 г. американским кибернетиком Дж. Хопкрофтом (John Нор-croft) [4]. 2-3-дерево представляет собой дерево, которое может иметь узлы двух видов — 2-узлы и 3-узлы. 2-узел содержит единственный ключ К и имеет два потомка: левый дочерний узел служит корнем поддерева, все ключи в котором меньше К, а правый — корнем поддерева, все ключи в котором больше К. (Дру­гими словами, 2-узел точно такой же, как и узел в классическом бинарном дереве поиска.) 3-узел содержит два упорядоченных ключа К\ и Ко {К\ < К2) и имеет три дочерних узла. Левый дочерний узел служит корнем поддерева, ключи в ко­тором меньше К\, средний — корнем поддерева, ключи в котором больше К\ и меньше К2, а правый — корнем поддерева, все ключи в котором больше К2 (рис. 6.7).



1. узел 3-узел

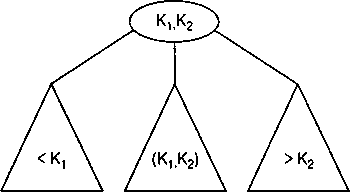


Рис. 6.7. Два вида узлов в 2-3-дереве

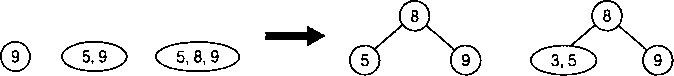
Последнее требование к 2-3-дереву заключается в том, что все его листья должны находиться на одном уровне, т.е. 2-3-дерево всегда сбалансировано по высоте (height-balanced): длина пути от корня дерева к листу должна быть одина­кова для всех листьев дерева. Это свойство достигается ценой разрешения иметь узлы с тремя дочерними узлами.

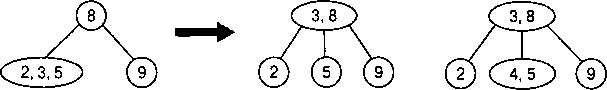
Поиск заданного ключа К в 2-3-дереве достаточно прост. Он начинается с кор­ня. Если корень представляет собой 2-узел, то мы действуем так же, как и в слу­чае бинарного дерева поиска, — либо прекращаем поиск, если значение К равно значению ключа корня, либо продолжаем поиск в левом или правом поддереве, в зависимости от того, меньше ли значение К, чем ключ корня, или больше. Если же корень представляет собой 3-узел, то после не более чем двух сравнений мы знаем, следует ли прекратить поиск (если К равно одному из ключей 3-узла) или в каком из трех поддеревьев он должен быть продолжен.

Вставка нового ключа в 2-3-дерево выполняется следующим образом. Новый ключ К всегда вставляется в лист, за исключением случая пустого дерева. Соот­ветствующий лист мы находим, выполняя поиск ключа К. Если искомый лист —

1. узел, мы вставляем К либо как первый, либо как второй ключ — в зависимости от того, меньше ли К, чем старый ключ, или больше. Если же искомый лист —
2. узел, то мы разделяем его на два: наименьший из трех ключей (двух старых и но­вого) помещается в первый лист, наибольший — во второй лист, а средний ключ

переносится в узел, родительский по отношению к старому листу (если лист — корень дерева, то для вставки нового ключа создается новый корень). Заметим, что перемещение среднего ключа в родительский узел может привести к перепол­нению родительского узла (если он был 3-узлом) и, следовательно, вызвать ряд разделений узлов вдоль цепочки предков листа.





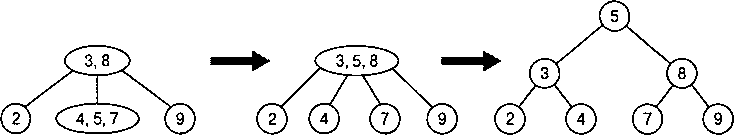


Рис. 6.8. Построение 2-3-дерева путем внесения ключей 9, 5, 8, 3, 2, 4, 7 Пример 2-3-дерева приведен на рис. 6.8.

Как и в любом дереве поиска эффективность словарных операций зависит от его высоты. Давайте сперва найдем ее верхнюю границу. 2-3-дерево высо­той h с минимальным количеством ключей представляет собой полное дерево с 2-узлами (такое, как последнее дерево на рис. 6.8 для h = 2). Следовательно, для любого 2-3-дерева высотой hen узлами мы получаем неравенство

га ^ ***1 + 2 + ■■■ + 2h = 2h+l -*** **1**,

следовательно,

h ^ log2 (га + 1) - 1.

С другой стороны, 2-3-дерево высотой h с максимальным количеством узлов представляет собой полное дерево, все узлы которого — 3-узлы, с двумя ключами и тремя потомками. Следовательно, для любого 2-3-дерева с п узлами

га^2-1 + 2- 3 + -- - + 2-3Л = 2

следовательно,

(l + 3 + • • • + 3Л) = 3Л+1 - 1

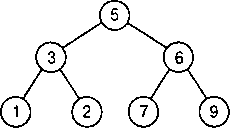
h ^ log3 (га + 1) - 1.

Из полученных таким образом верхней и нижней границ высоты h

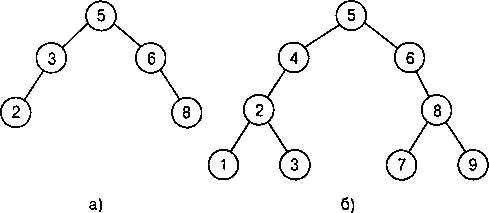
log3 (n + 1) - 1 < h ^ log2 (n + 1) — 1

вытекает, что временная эффективность поиска, вставки и удаления как в наи­худшем, так и в среднем случае — 0 (logn). В разделе 7.4 мы рассмотрим очень важное обобщение 2-3-деревьев — В-деревья.

**Упражнения 6.3**



1. Какие из представленных на рисунке деревьев являются AVL-деревья- ми?



1. а) Нарисуйте все бинарные деревья с п узлами (где п = 1,2,3,4,5),

которые удовлетворяют требованию сбалансированности AVL-дере- вьев.

б) Изобразите бинарное дерево высотой 4, которое представляет собой AVL-дерево и содержит наименьшее количество узлов среди всех таких деревьев.

1. Изобразите диаграмму одиночного L-поворота и двойного .RL-поворо- та в общем виде.
2. Для каждого из представленных наборов чисел постройте AVL-дерево путем последовательного внесения чисел в изначально пустое дерево.

а) 1, 2, 3, 4, 5, 6

б) 6, 5, 4, 3, 2, 1

в) 3, 6, 5, 1, 2, 4

1. а) Разработайте алгоритм вычисления диапазона AVL-дерева, содер­

жащего действительные числа (т.е. разность между наибольшим и наименьшим числами в дереве) и определите его эффективность в наихудшем случае.

б) Истинно или ложно следующее утверждение: наименьший и наи­больший ключи в AVL-дереве всегда находятся на последнем или предпоследнем уровне?

1. Напишите программу для построения AVL-дерева для списка из п раз­личных целых чисел.
2. а) Постройте 2-3-дерево для следующего множества символов: С, О,

М, Р, U, Т, I, N, G (используйте при этом алфавитное упорядочение букв и их последовательную вставку в изначально пустое дерево),

б) Найдите наибольшее и среднее количество сравнений ключей при успешном поиске в получившемся дереве в предположении равно­вероятного поиска ключей.

1. Пусть Тв и Т2\_з представляют собой, соответственно, классическое бинарное дерево поиска и 2-3-дерево, построенные для одного и того же набора ключей, вставляемых в деревья в одном и том же поряд­ке. Истинно или ложно следующее утверждение: поиск одного и того же ключа в Т2\_з всегда требует меньшего или такого же количества сравнений ключей, что и поиск в Тв?
2. Разработайте для 2-3-дерева, содержащего действительные числа, ал­горитм для вычисления диапазона (т.е. разности между наибольшим и наименьшим числом) дерева и определите его эффективность в наи­худшем случае.
3. Напишите программу для построения 2-3-дерева для данного списка п различных целых чисел.
4. Пирамиды и пирамидальная сортировка

Структура данных под названием пирамида (heap) представляет собой хитро­умную частично упорядоченную структуру данных, которая в особенности хоро­шо подходит для реализации очередей с приоритетами. Вспомним, что очередь с приоритетами (priority queue) представляет собой множество элементов с упо­рядочиваемой характеристикой, называющейся приоритетом (priority) элемента, и обеспечивающее выполнение следующих операций:

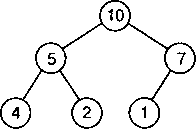
* поиск элемента с наивысшим (т.е. с наибольшим) приоритетом;
* удаление элемента с наибольшим приоритетом;
* добавление нового элемента в множество.

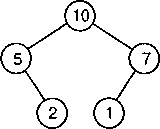
Пирамиды представляют особый интерес в первую очередь благодаря эффек­тивной реализации перечисленных операций. Пирамида также является структу­рой данных, которая служит краеугольным камнем теоретически важного алгорит­ма — пирамидальной сортировки (heapsort). Мы рассмотрим этот алгоритм позже, после того как дадим определение пирамиды и изучим ее основные свойства.

**Понятие пирамиды**

Определение 1. Пирамида (heap) может быть определена как бинарное дерево с ключами, назначенными ее узлам (по одному ключу на узел), для которого выполняются два следующих условия.

1. Требование к форме дерева. Бинарное дерево практически полное (essentially complete) или просто полное (complete), т.е. все его уровни заполнены, за исключением, возможно, последнего уровня, в котором могут отсутствовать некоторые крайние справа листья.
2. Требование доминирования родительских узлов. Ключ в каждом узле не меньше ключей в его дочерних узлах (условие считается автоматически выполняющимся для всех листьев).[[48]](#footnote-48) ■





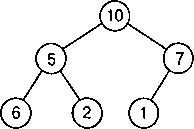


Рис. 6.9. Иллюстрация к определению пирамиды: таковой является только крайнее слева дерево

Рассмотрим, например, деревья на рис. 6.9. Первое дерево является пирами­дой. Второе дерево — не пирамида, поскольку нарушено требование к форме дерева. Третье дерево также не является пирамидой, поскольку в нем нарушено требование доминирования родительских узлов для узла с ключом 5.

Обратите внимание на упорядоченность значений в пирамиде сверху вниз — т.е. последовательность значений на любом пути от корня к листу убывающая (невозрастающая, если допускается наличие одинаковых ключей). Однако упо­рядоченности ключей слева направо нет, т.е. нет никаких соотношений между значениями ключей в узлах на одном уровне дерева или, в общем случае, в левом и правом поддеревьях одного узла.

Вот список важных свойств пирамид, которые несложно доказать (в качестве примера проверьте их выполнение для пирамиды, показанной на рис. 6.10.

1. Имеется ровно одно практически полное бинарное дерево с п узлами.

Его высота равна [log2 п\.

1. Корень пирамиды всегда содержит ее наибольший элемент.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Представление в виде массива | | | | | | |
| Индекс | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Значение |  | 10 | 5 | 7 | 4 | 2 | 1 |

Родительские Листья узлы

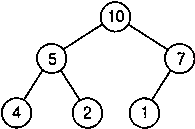


Рис. 6.10. Пирамида и ее представление в виде массива

1. Любой узел пирамиды со всеми его потомками также является пира­мидой.
2. Пирамида может быть реализована в виде массива путем записи ее элементов сверху вниз слева направо. Удобно хранить элементы пира­миды в позициях такого массива с 1 по гг, оставляя Н [0] либо неис­пользуемым, либо размещая в нем ограничитель, значение которого превышает значение любого элемента пирамиды. При использовании такого представления

а) ключи родительских узлов занимают первые [п/2J позиций в мас­сиве, а ключи листьев — последние [гг/2] позиций.

б) дочерние ключи по отношению к родительскому в позиции г (1 ^ г ^ [гг/2J) находятся в позициях 2г и 2г + 1, соответственно; родительский ключ для ключа в позиции г (2 ^ г ^ п) находится в позиции [\_г/2].

Таким образом, мы можем определить пирамиду как массив Н[1..п]9 в кото­ром каждый элемент в позиции г в первой половине массива больше или равен элементам в позициях 2 г и 2г + 1, т.е.

Н [г] ^ max {Н [2г], Н [2г + 1]} для г = 1,2,..., [гг/2] .

(Конечно, если 2г + 1 > гг, то выполняться должно только неравенство Н [г] ^ > Н [2г].) В то время как идеи, лежащие в основе алгоритмов с использованием пирамид, проще для понимания при представлении пирамид в виде бинарных деревьев, реальные реализации эти алгоритмов обычно существенно проще и эф­фективнее при использовании представления в виде массива.

Как же построить пирамиду для заданного множества ключей? Имеется два основных метода выполнить эту работу. Первый называется восходящим постро­ением пирамиды (bottom-up heap construction) и проиллюстрировано на рис. 6.11. При этом практически полное бинарное дерево инициализируется путем разме­щения п ключей в заданном порядке, а затем дерево “пирамидизируется” следу­ющим образом. Начиная с последнего родительского узла и заканчивая корнем алгоритм проверяет, выполняется ли для рассматриваемого узла требование до­минирования родительского узла. Если нет, то алгоритм обменивает ключ узла

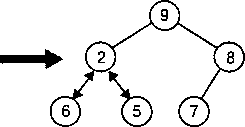
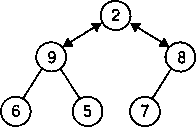
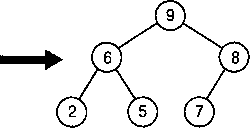
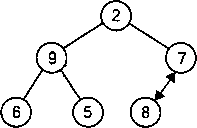
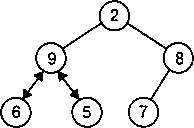


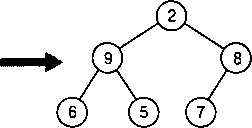
Рис. 6.11. Восходящее построение пирамиды для множества ключей 2, 9, 7, 6, 5, 8











К с наибольшим ключом среди его дочерних узлов и проверяет выполнение тре­бования доминирования родительского узла для ключа К в новой позиции. Этот процесс продолжается до тех пор, пока для ключа К не будет выполнено тре­бование доминирования родительского узла (в конечном итоге это требование будет выполнено, так как оно всегда выполняется для ключей в листьях). После завершения “пирамидизации” поддерева, корнем которого является данный роди­тельский узел, алгоритм выполняет те же действия с непосредственным предком этого узла. Алгоритм завершает свою работу после обработки корня дерева.

Перед тем как будет приведен псевдокод восходящего построения пирамиду, следует сделать одно замечание. Поскольку значение ключа не изменяется при его перемещении вниз по дереву, нет необходимости выполнять промежуточные обмены. Такое улучшение алгоритма можно представить как обмен пустого узла с большими ключами среди потомков (или как перемещение пустого узла вниз по дереву — прим. перев.) до достижения конечной позиции, куда и вставляется ранее сохраненный ключ.

**Алгоритм HeapBottomUp(H** **[l..n])**

// Построение пирамиды из элементов заданного массива при // помощи восходящего алгоритма

// Входные данные: Массив Н[1..п] упорядочиваемых элементов

// Выходные данные: Пирамида Н[1..п] for г <— |n/2j downto 1 do

***к*** <— ***г; v*** <— ***Н[к] heap —*** false

while not heap and 2 \* к ^ n do

j <— 2 \* к

if j < n II Имеется два дочерних узла if H\j\ < H[j + 1]

j <- 3 + 1 if v > H\j] heap <— true else

***H[k]*** <- ***H\j}; k^j***

***H[k\*** <- ***v***

Насколько эффективен данный алгоритм в наихудшем случае? Предположим для простоты, что п = 2к — 1, так что дерево пирамиды заполнено, т.е. на каждом уровне находится максимально возможное количество узлов. Пусть h — высота пирамиды; согласно первому свойству пирамид из списка в начале этого разде­ла, h = [log2nJ (или просто [log2(n + 1)]—l = fc — 1 для рассматриваемого нами значения п). В наихудшем случае при выполнении алгоритма построения пирамиды каждый ключ на уровне г дерева будет перемещаться до уровня листа h. Поскольку перемещение на один уровень вниз требует двух сравнений (одно­го для поиска дочернего узла с наибольшим ключом, и второго для выяснения, требуется ли обмен ключами), общее количество сравнений ключей, выполняе­мых при перемещении ключа на уровне г, равно 2 (/г — г). Таким образом, общее количество сравнений ключей в наихудшем случае составляет

/i-i /i-i

Cworst (п) = 2 {h — г) = 2 (h — г) 21 = 2 (n — log2 (п + 1)),

**г=0 Ключи на г=0**

**уровне г**

где корректность последнего равенства может быть доказана либо с использо­ванием формулы для суммПы Yli=1 (см. приложение А), либо при помощи математической индукции по h. Таким образом, при использовании восходящего алгоритма пирамида размером п может быть построена с выполнением менее чем 2п сравнений.

Альтернативный (и менее эффективный) алгоритм строит пирамиду путем последовательных вставок нового ключа в ранее построенную; некоторые авторы называют этот алгоритм нисходящим построением пирамиды (top-down heap construction). Каким же образом можно добавить новый ключ в пирамиду? Нач­нем с добавления нового узла с ключом К после последнего листа имеющейся пирамиды, а затем переместим К в соответствующее его значению место в но­вой пирамиде следующим образом. Сравним К с родительским ключом: если он не меньше К, алгоритм прекращает работу (полученная структура является пирамидой). В противном случае обменяем эти два ключа и будем сравнивать К с новым родителем. Этот процесс продолжается до тех пор, пока К не перестанет превышать значение ключа в родительском узле или не достигнет корня (этот процесс проиллюстрирован на рис. 6.12). В этом алгоритме также можно переме­щать пустой узел до достижения им корректной позиции, а затем присвоить ему значение К.

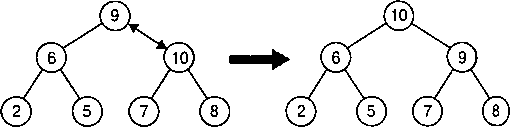
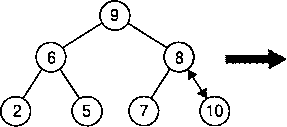
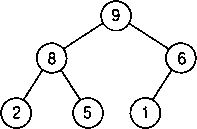


Рис. 6.12. Вставка ключа 10 в пирамиду, построенную на рис. 6.11



Очевидно, что такая вставка не может требовать большего количества срав­нений ключей, чем высота пирамиды. Поскольку высота пирамиды с п узлами около log2 гг, временная эффективность вставки составляет О (log гг).



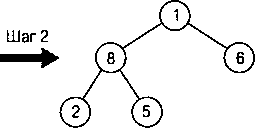
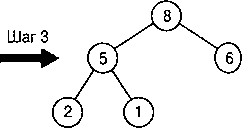
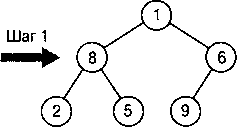


Рис. 6.13. Удаление корневого ключа из пирамиды





Как выполняется удаление узла из пирамиды? Мы рассмотрим здесь только наиболее важный случай удаления ключа из корня, оставляя вопрос об удале­нии произвольного ключа в качестве самостоятельного упражнения для читателя (авторы учебников вообще склонны перекладывать массу работы на читателей, правда?). Итак, удаление корневого ключа из пирамиды можно выполнить при помощи следующего алгоритма (проиллюстрированного на рис. 6.13).

Шаг 1. Обменять ключ в корне с последним ключом пирамиды.

Шаг 2. Уменьшить размер пирамиды на 1.

Шаг 3. “Пирамидизировать” уменьшенное дерево путем перемещения К вниз по дереву так же, как мы делали это в восходящем алгоритме построения пирамиды, — т.е. проверяя выполнение требования доминирования ро­дительских узлов: если оно выполняется, алгоритм завершает работу, если нет — обмениваем К с наибольшим из дочерних узлов и повторя­ем данную операцию до тех пор, пока в очередной позиции К требо­вания доминирования родительских узлов не окажется выполненным.

Эффективность операции удаления определяется количеством выполняемых сравнений ключей, необходимых для “пирамидизации” дерева после того, как был сделан обмен и размер пирамиды был уменьшен на 1. Поскольку не может потребоваться сравнений больше, чем удвоенная высота пирамиды, временная эффективность удаления из пирамиды — О (log гг).

**Пирамидальная сортировка**

Теперь мы можем описать пирамидальную сортировку (heapsort) — интерес­ный алгоритм сортировки, открытый Дж. Вильямсом (J. W. J. Williams) [122]. Этот двухэтапный алгоритм работает следующим образом.

Этап 1 (построение пирамиды). Строим пирамиду для заданного массива.

Этап 2 (удаление наибольших элементов). Применяем операцию удаления корня тг — 1 раз.

В результате элементы массива удаляются в порядке уменьшения. Но посколь­ку при реализации пирамиды с использованием массива удаляемый элемент рас­полагается последним, массив, получающийся в результате пирамидальной сор­тировки, оказывается отсортирован в порядке возрастания. Пример пошагового выполнения пирамидальной сортировки приведен на рис. 6.14 (на рис. 6.11 наме­ренно использовались те же входные данные, для того чтобы вы могли сравнить восходящее построение пирамиды при ее реализации в виде дерева и с использо­ванием массива).

Поскольку мы уже знаем, что этап построения пирамиды алгоритма пирами­дальной сортировки занимает время О (тг), осталось выяснить временную эффек­тивность второго этапа. Для количества сравнений ключей С(тг), необходимого для удаления корневых ключей из пирамид уменьшающегося от тг до 2 размера, получаем следующее неравенство:

**гг—1**

С (тг) < 2 [\_log2 (тг - 1)J + 2 |\_log2 (гг — 2)J Ч + 2 [log2 lj ^ 2 ^ log2 г ^

**г=1**

**гг—1**

^ 2 ^ log2 (тг — 1) = 2 (п — 1) log2 (тг — 1) ^ 2тг log2 тг.

**г=1**

Это означает, что для второго этапа пирамидальной сортировки С(тг) € еО (nlogn). Более подробный анализ показывает, что в действительности вре­менная эффективность пирамидальной сортировки равна © (тг log тг) как в сред­нем, так и в наихудшем случаях. Таким образом, временная эффективность пира­мидальной сортировки попадает в тот же класс, что и сортировка слиянием, но,

Этап 2. Удаление наибольших элементов 9 6 8 2 5 7

Этап 1. Построение пирамиды 2 9 7 6 5 8

2 9 8 6 5 7

2 9 8 6 5 7

9 2 8 6 5 7

9 6 8 2 5 7

1. 6 8 2 5 19
2. 6 7 2 5
3. 6 7 2 | 8

7 6 5 2

2 6 5 | 7

1. 2 5

5 2 I 6

5 2

2 I 5 2

Рис. 6.14. Пирамидальная сортировка массива 2, 9, 7, 6, 5, 8

в отличие от последней, выполняется “на месте”, без привлечения дополнитель­ной памяти. Эксперименты, проведенные над случайными файлами, показывают, что пирамидальная сортировка работает медленнее быстрой, однако вполне может соперничать с сортировкой слиянием.

**Упражнения 6.4**

1. Докажите следующее неравенство, использованное в тексте раздела:

/i-i

У" ***2 (h*** — ***г) 2г — 2 (п —*** log**2** ***(п*** + **1**)), где ***п = 2h+l*** — **1**.

i=0

1. а) Разработайте эффективный алгоритм для поиска и удаления из пи­

рамиды элемента с наименьшим значением и определите его вре­менную эффективность,

б) Разработайте эффективный алгоритм для поиска и удаления из пира­миды элемента с заданным значением v и определите его временную эффективность.

1. Отсортируйте следующие списки с помощью пирамидальной сорти­ровки с использованием представления пирамид в виде массивов:

а) 1, 2, 3, 4, 5 (в возрастающем порядке)

б) 5, 4, 3, 2, 1 (в убывающем порядке)

в) S, О, R, Т, I, N, G (в алфавитном порядке)

1. Является ли пирамидальная сортировка устойчивой?
2. Какую разновидность метода “преобразуй и властвуй” представляет собой пирамидальная сортировка?
3. Реализуйте три алгоритма сортировки — слиянием, быструю и пирами­дальную — на языке программирования по вашему выбору и исследуй­те их производительность для массивов размером п = 10[[49]](#footnote-49), 10[[50]](#footnote-50), 104, 105, 10е. Для каждого размера рассмотрите

а) случайно сгенерированные числа в диапазоне [1..п];

б) последовательность чисел 1,2,... ,п;

image156

в) последовательность чисел п,п — 1,... ,1.

1. Представьте горку спагетти, длины которых представляют числа, кото­рые требуется отсортировать.

а) Разработайте “сортировку спагетти” — алгоритм сортировки, ис­пользующий преимущества такого неортодоксального представле­ния.

б) Какое отношение этот пример компьютерного фольклора (см. [34]) имеет к теме данной главы вообще и к пирамидальной сортировке в частности?

1. Схема Горнера и возведение в степень

В этом разделе мы рассмотрим задачу вычисления значения полинома

р{х) = апхп + an-ixn~l Н Ь а\х + а0 (6.1)

в заданной точке х и ее важный частный случай — вычисление хп. Полиномы об­разуют наиболее важный класс функций благодаря множеству хороших свойств, с одной стороны, и возможности их применения для аппроксимации других ти­пов функций, — с другой. Задача эффективной работы с полиномами актуальна несколько столетий, и за последние 50 лет были сделаны новые открытия в этой области. Несомненно, наиболее важным из них было быстрое преобразование Фурье (fast Fourier transform, FFT). Практическая важность этого замечательного алгоритма, основанного на представлении алгоритмов при помощи их значений в специально выбранных точках, настолько велика, что некоторые ученые счита­ют его наиболее важным алгоритмическим открытием всех времен. Вследствие его относительной сложности мы не станем рассматривать быстрое преобразова­ние Фурье и порекомендуем читателю обратиться к соответствующим учебникам, например к [102] или [32].

**Схема Горнера**

Схема Горнера (Homer’s rule) — один из старых, но очень элегантных и эффек­тивных алгоритмов для вычисления полиномов. Он назван по имени британского математика В. Горнера (W. G. Homer), который опубликовал этот алгоритм в на­чале 19 века. Однако, как утверждает Д. Кнут (D. Knuth) ([66]), еще за 150 лет до Горнера данный метод использовался И. Ньютоном (I. Newton). Вы оцените этот алгоритм еще больше, если сначала самостоятельно разработаете и исследуете эффективность алгоритма вычисления значения полинома (см. упражнения 1 и 2 к данному разделу).

Схема Горнера — хороший пример использования метода изменения пред­ставления, поскольку он основан на представлении р (х) при помощи формулы,

отличающейся от (6.1). Эта новая формула получается из (6.1) путем последова­

тельного вынесения х за скобки с образованием полиномов с уменьшающимися степенями:

р (х) = (... (апх + an-i) х + ...) х + ао- (6.2)

Например, для полинома р (х) = 2х4 — х3 + Зх2 + х — 5 получим:

р (х) = 2х4 — х3 + Зх2 + х — 5 =

= х (2х3 — х2 + Зх + 1) — 5 =

= х (х (2х2 — x + 3) + l)—5 =

= х (х (х (2х — 1) + 3) + 1) — 5.

Для вычисления значения полинома в точке х это значение надо просто подста­вить в формулу (6.2). Глядя на нее, трудно поверить, что это и есть эффективный алгоритм вычисления значения полинома, однако ее неприглядность — не более чем внешний вид, и, как вы увидите, нет необходимости в такой записи полинома: все, что нам надо, — это список коэффициентов полинома.

Вычисления вручную проще организовать при помощи таблицы, состоящей из двух строк. Первая содержит коэффициенты полинома (включая те из них, которые равны 0, — если таковые имеются), перечисленные в порядке от старшего ап к младшему ао- Вторая строка используется для хранения промежуточных результатов (за исключением первой записи, которая просто равна ап). После такой инициализации очередная запись в таблице вычисляется как последнее значение, умноженное на х, плюс коэффициент из первой строки. Последняя запись таблицы, вычисленная таким способом, и есть искомое значение полинома.

Пример 1. Вычисление р (х) = 2х4 — х3 + 3.x2 + х — 5 в точке х = 3.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Коэффициенты | 2 | -1 | 3 | 1 | -5 |
| х = 3 | 2 | 3 • 2 + (—1) = 5 | 3 • 5 + 3 = 18 | 3 • 18 + 1 = 55 | 3 • 55 + (-5) = 160 |

Итак, р (3) = 160. (Сравнивая записи таблицы с формулой (6.3), можно видеть, что 3 • 2(—1) = 5 — это значение 2х — 1 в точке х = 3; 3-5 + 3 = 18 — значение х (2х — 1) + 3 в точке х = 3; 3-18+1 = 55 — значение х (х (2х — 1) + 3) +1 в точке х = 3 и, наконец, 3 • 55 + (—5) = 160 — значение х (х (х (2х — 1) + 3) + 1) — 5 = = р (х) в точке х = 3.) ■

Псевдокод схемы Горнера короче, чем мы себе представляем псевдокод для нетривиального алгоритма.

**Алгоритм Horner** (Р **[0..п], х)**

// Вычисление значения полинома в данной точке // по схеме Горнера

// Входные данные: Массив Р[0..п] коэффициентов полинома

// степени п (хранятся от младшего

// к старшему) и число х

// Выходные данные: Значение полинома в точке х

***р*** «- ***Р[п\***

for г <— п — 1 downto 0 do

р <— х \* р + Р[г\ return р

Количество умножений и сложений определяется одной и той же формулой:

**тг—1**

М (тг) = A (тг) = ^ 1 = тг.

**г=0**

Чтобы оценить, насколько эффективен алгоритм схемы Горнера, рассмотрим только первый член полинома степени тг: апхп. Вычисление только одного этого члена методом грубой силы потребует тг умножений, в то время как схема Горне­ра, кроме этого члена, вычисляет еще тг — 1 других членов и при этом использует то же количество умножений! Неудивительно, что схема Горнера — оптимальный алгоритм для вычисления полиномов без предварительной обработки полиноми­альных коэффициентов.

Схема Горнера имеет несколько полезных побочных результатов. Промежу­точные числа, генерируемые алгоритмом в процессе вычисления р (ж) в точке жо, представляют собой коэффициенты частного от деления р (ж) на х — жо, а конеч­ный результат, равный значению р(хо), представляет собой остаток от деления р (х) нах — хо. Так, в соответствии с рассмотренным примером, частное и остаток от деления 2х4—х3+Зх2+х—5 на х—3 равны, соответственно, 2ж3+5ж2 + 18ж+55 и 160. Такой алгоритм деления, известный как синтетическое деление (synthetic division), более удобен, чем так называемое “длинное деление” (но в отличие от длинного деления он применим только для деления на ж — с, где с — константа).

**Бинарное возведение в степень**

Поразительная эффективность схемы Горнера сводится на нет, будучи приме­ненной к вычислению ап, т.е. значению хп при х — а. Фактически в этом частном случае алгоритм вырождается в алгоритм, основанный на применении грубой си­лы, — в умножение значения а само на себя, с бессмысленными прибавлениями О между умножениями. Поскольку вычисление ап (на самом деле ап mod га) явля­ется основной операцией в ряде важных алгоритмов проверки чисел на простоту и алгоритмов шифрования, мы рассмотрим два важных алгоритма вычисления ап, которые основаны на идее изменения представления. Оба они используют бинар­ное представление показателя степени тг, но один из них обрабатывает бинарную строку слева направо, а другой — Справа налево.

Пусть п = й/ ... bi... bo — битовая строка, представляющая положительное целое п в двоичной системе счисления. Это означает, что значение п может быть вычислено как значение полинома

р (ж) = й/ж7 н Ь Ыхг н Ь йо (6.4)

при ж = 2. Например, если п = 13, его бинарное представление имеет вид 1101 и

13 = 1 • 23 + 1 • 22 + 0 • 21 + 1 • 2°.

Давайте вычислим значение этого полинома с использованием схемы Горне­ра — мы увидим, как из него вытекает способ вычисления степени

ап = а?= abixI+"+bixi+-+bo'

|  |  |
| --- | --- |
| Схема Горнера для вычисления бинарного поли­нома р (2) | Следствие для вычис­ления an = ap№ |
| р <— 1 // Старший разряд всегда 1 при п ^ 1 | oP <— a1 |
| for i«— I — 1 downto 0 | for i «— / — 1 downto 0 |
| p «— 2p + bi | oP <— oP «— a2p+bi |
| Однако | |
| a2p+bi = a2p ■ abi = (ap)2 ■ abi = {  1K)2' | если bi = 0, |
| ■ a если bi = 1. |

Таким образом, после инициализации переменной-аккумулятора значением а можно сканировать битовую строку, представляющую показатель степени, при каждом сканировании нового бита возводя значение аккумулятора в квадрат и, ес­ли сканируемый бит равен 1, дополнительно умножая полученный квадрат на ве­личину а. Это наблюдение приводит нас к следующему методу вычисления ап — бинарному возведению в степень слева направо (left-to-right binary exponentiation).

АЛГОРИТМ ***LeftRightBinaryExponentiation (a, b*** (n))

// Вычисление an при помощи алгоритма бинарного возведения // в степень слева направо

// Входные данные: Число а и список Ь(п) битов 6/,..., 6о

// в бинарном представлении натурального п

// Выходные данные: Значение ап product <— а

for i <— / — 1 downto 0 do

***product*** <— ***product*** \* ***product*** if ***к =*** **1 *product*** <— ***product*** \* ***a*** return ***product***

Пример 2. Вычислим а13 при помощи алгоритма бинарного возведения в степень слева направо. Здесь п = 13 = IIOI2. Таким образом, мы имеем

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Биты n | 1 | 1 | 0 | 1 |
| Аккумулятор | a | a2 • a = a3 | (a3)2 = a6 | (a6)2 • a = a13 |

Поскольку алгоритм выполняет при каждом повторении своего единственного цикла одно или два умножения, общее количество умножений М (п) при вычис­лении ап составляет

(b --1) < М(п) < 2(6- 1),

где Ь — длина битовой строки, представляющей показатель степени п. Учитывая, что b — 1 = |\_log2 п\, можно заключить, что эффективность бинарного возведения в степень слева направо — логарифмическая, так что этот алгоритм принадлежит лучшему классу эффективности, чем возведение в степень, основанное на грубой силе (которое всегда требует п — 1 умножений).

При бинарном возведении в степень справа налево (right-to-left binary expo­nentiation) используется тот же полином р (2) (см. (6.4)), дающий значение п, но вместо применения схемы Горнера полином используется иначе:

ап = аЬ,2'+-+Ь2‘+-+6о = аМ' . . аЬ\* . .. . . а&о.

Таким образом, ап можно вычислить как произведение членов

b.2i J а2' если bi = 1,

1 1 если bi = О,

т.е. произведение последовательных членов а2', с опусканием тех из них, для кото­рых бит bi равен 0. Кроме того, о2‘ можно вычислять путем возведения в квадрат

члена, вычисленного для предыдущего значения г, поскольку о2" = ^а2\* 1 ^ . Мы вычисляем все такого вида степени а, от наименьшей к наибольшей (справа нале­во), но в результат включаются только те из них, для которых соответствующий бит равен 1. Вот как выглядит псевдокод данного алгоритма.

**АЛГОРИТМ *RightLeftBinaryExponentiation (а, b*** **(*п*))**

// Вычисление ап при помощи алгоритма бинарного возведения // в степень справа налево

// Входные данные: Число а и список Ь(п) битов 6/,..., &о

// в бинарном представлении натурального п

// Выходные данные: Значение ап term, <— а // Инициализация а2' if 6о = 1 product •\*— а else product \*— 1 for г «— 1 to / do

***term*** <— ***term, \* term*** if ***Ы =*** **1**

***product <— product*** \* ***term*** return ***product***

Пример 3. Вычислим а13 при помощи алгоритма бинарного возведения в степень справа налево. Здесь п = 13 = 1101.2. Таким образом, имеем

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 1 | Биты п |
| а8 | а4 | а2 | а | Члены а2\* |
| а5 - а8 = а13 | а • а4 = а5 |  | а | Аккумулятор произведения |

Очевидно, что эффективность данного алгоритма также логарифмическая — по тем же причинам, что и для алгоритма бинарного возведения в степень сле­ва направо. Полезность обоих алгоритмов несколько снижается из-за того, что требуется точное бинарное разложение показателя степени п. В упражнении 8 к данному разделу требуется разработать алгоритм, который лишен этого недо­статка.

**Упражнения 6.5**

1. Рассмотрим следующий алгоритм вычисления полинома, основанный на применении грубой силы.

АЛГОРИТМ ***BruteForcePolynomialEvaluation*** ***(Р*** [0..п], ***х)***

II Алгоритм вычисляет значение полинома Р в точке х II с использованием алгоритма на основе грубой силы,

// вычисляющего члены от большей степени к меньшей // Входные данные: Массив Р[0..п] коэффициентов полинома

// степени п, хранящиеся начиная от

// меньшего к большему, и число х

II Выходные данные: Значение полинома в точке х

**р\*—** **0.0**

for i <— n downto 0 do

power <— 1 for j <— 1 to i do

power •\*— power \* x p <— p + P[i] \* power return p

Найдите общее количество умножений и сложений, выполняемых этим алгоритмом.

1. Напишите псевдокод алгоритма вычисления полинома, основанного на применении грубой силы, который подставляет значение переменной в формулу и вычисляет ее от меньшего члена к большему. Найдите общее количество умножений и сложений, выполняемых таким алго­ритмом.
2. а) Оцените, насколько схема Горнера быстрее алгоритма грубой си­

лы из упражнения 2, если 1) время одного умножения значительно превышает время одного сложения; 2) время одного умножения при­мерно равно времени одного сложения,

б) Является ли схема Горнера быстрее алгоритма грубой силы ценой использования повышенного количества памяти?

1. а) Примените схему Горнера для вычисления полинома

р (х) = Зх4 — х3 + 2х + 5 в точке х = —2.

б) Используйте результаты работы схемы Горнера для поиска частного и остатка от деления р (х) на х + 2.

1. Сравните количество умножений и сложений/вычитаний, необходимых для “длинного деления” полинома р(х) — апхп + an\_ixn\_1 + ■ • • + + ао на х - с, где с — константа, с количеством этих операций при “синтетическом делении”.
2. а) Примените бинарное возведение в степень слева направо для вы­

числения а17.

б) Можно ли расширить бинарное возведение в степень слева направо так, чтобы оно работало для всех неотрицательных целых показате­лей степени?

1. Примените бинарное возведение в степень справа налево для вычис­ления а17.
2. Разработайте нерекурсивный алгоритм для вычисления ап, который имитирует бинарное возведение в степень справа налево, но не ис­пользует явное бинарное представление п.
3. Стоит ли использовать такой алгоритм общего назначения, как схема Горнера, для вычисления полинома р (х) — хп + xn\_1 -I 1- х + 1?
4. В соответствии со следствием из Основной теоремы алгебры, каждый полином

р (х) = апхп + an\_]Xn\_1 -I 1- ао

может быть представлен в виде

р (х) = ап (х - xi) (х - хг) ■.. (х - хп),

где XI, Х2,... ,хп — корни полинома (в общем случае комплексные и не обязательно различные). Подумайте, какое из двух представлений более удобно для каждой из следующих операций:

а) вычисления полинома в данной точке;

б) сложения двух полиномов;

в) умножения двух полиномов.

б.б Приведение задачи

Знаете ли вы историю о мальчике, которого мама учила, как сварить яйцо на завтрак в ее отсутствие? Он должен был снять с полки кастрюльку, налить в нее воду, поставить на плиту, включить плиту, положить яйцо в воду и через 5 минут после закипания достать вареное яйцо, выключить плиту, помыть и поставить на место кастрюльку... Но однажды мама не очень торопилась на работу, так что пришедший на кухню мальчик увидел, что кастрюлька с водой уже стоит на плите... Не растерявшись, он вылил воду, поставил кастрюльку на полку и вышел из кухни. А потом зашел и выполнил все действия, которым его научила мама.

Способ, которым мальчик приготовил яйцо на завтрак, — пример важной стра­тегии решения задач, именуемой приведением, или сведением задачи (problem reduction). Если вам надо решить задачу, ее можно привести к другой задаче, решение которой вам известно (рис. 6.15).

*Приведение Алгоритм А*

Задача 1 w Задача 2 \_

\_ Решение задачи 2

(которую надо решить) ^ (решаемая при помощи алгоритма А) ^

Рис. 6.15. Стратегия приведения задачи

Идея приведения задачи играет центральную роль в теоретической кибернети­ке, где она используется для классификации задач в соответствии с их сложностью (об этом мы поговорим в главе 10). Однако эта же стратегия может использоваться и для решения практических задач. Сложность заключается в том, чтобы найти задачу, к которой можно привести исходную. Кроме того, если мы хотим полу­чить практический результат, необходимо, чтобы алгоритм приведения был более эффективным, чем алгоритм непосредственного решения исходной задачи.

Заметим, что раньше вы уже встречались с этой методикой. Например, в раз­деле 6.5 упоминалось о так называемом синтетическом делении, выполняемом при использовании схемы Горнера для вычисления значения полинома. В раз­деле 4.6 мы использовали следующий факт из аналитической геометрии: если Pi = (#ъ 2/i)> Р2 = (^2, У2) и рз = (хз, уз) — три произвольные точки на плоско­сти, то определитель

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | XX | У\ | 1 |  |
| det | Х2 | У2 | 1 | = Х1У2 + XZУг + Х2УЗ ~ Х3у2 - Х2у\ - Х\уг |
|  | хз | Уз | 1\_ |  |

положителен тогда и только тогда, когда точка рз находится слева от ориенти­рованной прямой Р1Р2, проведенной через точки р\ и p<i- Другими словами, мы свели геометрическую задачу об относительном расположении трех точек к задаче о знаке определителя матрицы. Более того, вся аналитическая геометрия основана на идее приведения геометрических задач к алгебраическим, и основная заслуга в этом принадлежит Рене Декарту (Rene Descartes) (1596-1650). В этом разде­ле мы приведем еще несколько примеров алгоритмов, основанных на стратегии приведения задач.

Вычисление наименьшего общего кратного

Вспомним, что наименьшее общее кратное (least common multiple) двух на­туральных чисел тип (обозначаемое как 1cm (m, га)) определяется как наимень­шее натуральное число, делящееся и на т, и на п. Например, 1cm (24,60) = 120 и 1cm (11,5) = 55. Наименьшее общее кратное — одно из наиболее важных поня­тий в арифметике и алгебре; возможно, вы вспомните метод его поиска, которому вас учили в школе. Если имеется разложение чисел т и п на простые множители, то 1cm (m, п) можно вычислить как произведение всех общих простых множите­лей тип, умноженное на произведение простых множителей т, не являющихся множителями га, и на произведение простых множителей га, не являющихся мно­жителями т. Например,

24 = 2 • 2 • 2 • 3 60 = 2 • 2 • 3 • 5 lcm (24,60) = (2 • 2 • 3) • 2 • 5 = 120

В качестве вычислительной процедуры данный алгоритм имеет те же недостат­ки, что и упоминавшийся в разделе 1.1 алгоритм поиска наибольшего общего делителя путем разложения чисел на простые множители — неэффективность и требование наличия списка последовательных простых чисел.

Существенно более эффективный алгоритм вычисления наименьшего общего кратного можно разработать при помощи приведения задачи. У нас есть очень эффективный алгоритм Евклида для поиска наибольшего общего делителя, кото­рый представляет собой произведение всех общих простых множителей тип. Можем ли мы написать формулу, связывающую lcm (m, п) и gcd (m, га)? Нетруд­но увидеть, что произведение lcm (гаг, га) и gcd (гаг, га) включает по одному разукаждый из простых множителей тип, следовательно, это произведение просто равно произведению тип. Это наблюдение приводит к формуле

т • п

lcm (m, тг) =

gcd (га, тг) ’

где gcd (га, тг) можно вычислить при помощи эффективного алгоритма Евклида.

Подсчет путей в графе

В качестве следующего примера рассмотрим задачу подсчета путей между двумя вершинами графа. Методом математической индукции нетрудно доказать, что количество различных путей длиной к > 0 от г-ой к j-ой вершине графа (ори­ентированного или неориентированного) равно (г, j)-OMy элементу Ак, где А — матрица смежности графа (кстати, рассмотренные в предыдущем разделе алгорит­мы возведения чисел в степень применимы и для возведения в степень матриц). Таким образом, задача подсчета путей в графе может быть решена при помощи алгоритма для вычисления соответствующей степени его матрицы смежности.

В качестве конкретного примера рассмотрим граф на рис. 6.16. Его матрица смежности А и ее квадрат А2 указывают количество путей (длиной, соответ­ственно, 1 и 2) между вершинами графа. В частности, имеется три пути длиной 2, начинающиеся и заканчивающиеся в вершине а: а —Ь — а, а — с — а и a — d — а, и только один путь длиной 2 от а до с: а — d — с.

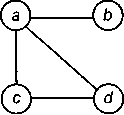
*b*

1

О

О

О



*А =*

А2:

Рис. 6.16. Граф, его матрица смежности и ее квадрат А2. Элементы матриц А и А2 указывают количество путей длиной 1 и 2, соответ­ственно

Приведение задач оптимизации

Очередной пример — решение задач оптимизации. Если в задаче требуется найти максимум некоторой функции, говорят, что это задача максимизации (max­imization problem); если же задача состоит в поиске минимума — то это задача минимизации (minimization problem). Предположим, требуется найти минимум некоторой функции / (х), и у нас есть алгоритм, позволяющий найти макси­мум функции. Как можно им воспользоваться? Ответ находится в простейшей формуле:

min / (х) = — max [—/ (х)].

Другими словами, чтобы минимизировать функцию, можно максимизировать функцию с обратным знаком, а чтобы получить корректное значение минимума, надо изменить знак у найденного максимума. Это свойство проиллюстрировано на рис. 6.17.

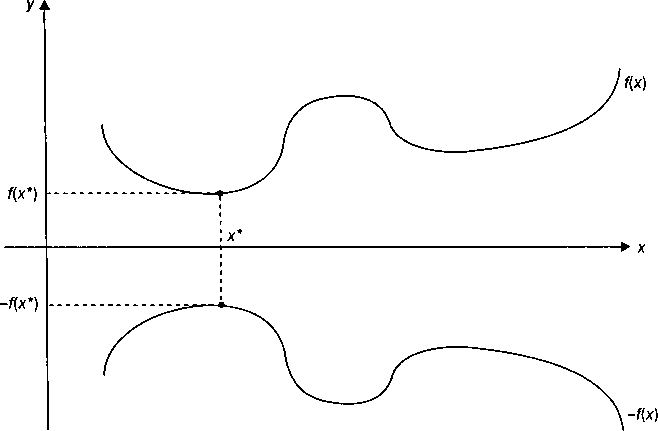


Рис. 6.17. Связь между задачами минимизации и максимизации: min / (х) = - шах [-/ (х)]

Само собой, справедлива и формула, позволяющая свести задачу максимиза­ции к задаче минимизации:

max / (х) = — min [—/ (х)}.

Это соотношение между задачами минимизации и максимизации носит самый общий характер — оно выполняется для функций с любой областью определения D. В частности, его можно применить к функциям нескольких переменных при дополнительных ограничениях. Очень важный класс таких задач будет рассмотрен в следующем подразделе.

Раз уж мы рассматриваем задачи оптимизации функции, стоит отметить, что стандартная процедура поиска точек экстремума функции фактически основана на приведении задачи. Она предполагает поиск производной функции /' (х) и ре­шение уравнения /' (х) = 0 для поиска критических точек функции. Другими словами, задача оптимизации приводится к задаче решения уравнения как основ­ной части поиска точек экстремума. Заметим, что мы не называем эту процедуру алгоритмом, поскольку она не является точно определенной. В действительности не существует общих методов решения уравнений. Маленький секрет учебников

заключается в том, что рассматриваемые в них примеры задач тщательно ото­браны так, что критические точки всегда можно найти без особых затруднений. Это облегчает жизнь студентам и преподавателям, но может создать у студентов ложное впечатление простоты задач.

Линейное программирование

Многие задачи, в которых требуется принять оптимальное решение, могут быть приведены к экземпляру задачи линейного программирования (linear pro­gramming), которая представляет собой задачу оптимизации линейной функции нескольких переменных при накладываемых ограничениях в виде линейных урав­нений и неравенств.

Пример 1. Рассмотрим задачу вклада денег. Имеется сумма в 100000$, которую можно разместить в акциях, на депозитном счету или на текущем счету. Акции дают годовую прибыль 10%, депозит — 7%, а текущий счет — 3%. Поскольку вкладывание денег в акции — занятие рискованное, в них можно вложить не более трети от суммы, положенной на депозитный счет. Кроме того, не менее 25% от общей суммы, вложенной в акции и на депозит, должно быть положено на текущий счет. Как разместить деньги, чтобы получить максимальную прибыль?

Создадим математическую модель. Пусть х, у и z — суммы в тысячах долла­ров, вкладываемые, соответственно, в акции, на депозит и на текущий счет. При

использовании таких обозначений мы можем сформулировать задачу следующим

образом:

Максимизировать 0.10х + 0.07у + 0.03z при условиях х + у + z = 100

х < h

z ^ 0.25 (х + у)

х ^ 0, у ^ 0, г ^ 0. ш

Хотя эта конкретная задача невелика и проста, она показывает, как задача принятия оптимального решения может быть приведена к экземпляру задачи ли­нейного программирования

Максимизировать с\х\ + • ■ • + CnXn

(или минимизировать)

при условиях ацх\ Н + ainxn ^ (или или =) bi

**При i = 1, ... ,771** Х\ **^ 0,...,** хп **^ 0.**

(Последняя группа ограничений — так называемые ограничения неотрицательно­сти — являются, строго говоря, необязательными, поскольку они представляют

собой частные случаи более общих ограничений ацх\ Н Ь сцпхп ^ Ы; просто

их удобнее рассматривать отдельно.)

Доказано, что линейное программирование достаточно гибко для моделиро­вания широкого диапазона важных приложений, таких как расписание полетов экипажей авиакомпаний, нефтеразведка и нефтедобыча или оптимизация про­мышленного производства. Многие ученые рассматривают линейное программи­рование как одно из наиболее важных достижений в истории прикладной матема­тики. Классический алгоритм для решения задачи линейного программирования называется симплекс-методом (simplex method); он был открыт американским математиком Дж. Данцигом (G. Dantzig) в 1940-х годах [33]. Хотя в наихудшем случае этот алгоритм экспоненциален, для обычных входных данных он работает очень хорошо. Усилия множества кибернетиков за последние 50 лет довели ал­горитм и его реализацию до того состояния, когда задача с десятками, если не сотнями тыся^ переменных и ограничений решается за вполне разумное время. Кроме того, относительно недавно были открыты и другие алгоритмы для реше­ния задач линейного программирования общего вида\* наиболее известен среди них алгоритм Наренды Кармаркара (Narenda Karmarkar) [57]. Теоретическая важ­ность этих новейших алгоритмов заключается в их доказанной полиномиальности в наихудшем случае; алгоритм же Кармаркара, как показали эмпирические тесты, способен, помимо прочего, составить конкуренцию по эффективности симплекс- методу.

Важно отметить, однако, что симплекс-метод и алгоритм Кармаркара в со­стоянии успешно решать только задачи линейного программирования, в которых отсутствует ограничение, что все переменные могут принимать только целочис­ленные значения. Если такое ограничение присутствует в задаче, то она называет­ся целочисленной задачей линейного программирования (integer linear program­ming). Известно, что целочисленные задачи линейного программирования намно­го сложнее, и для них неизвестен алгоритм решения с полиномиальным временем работы (как вы узнаете из главы 10, вполне вероятно, что такой алгоритм вообще не существует). Для решения таких задач используются иные подходы, с одним из которых вы познакомитесь в разделе 11.2.

Пример 2. Рассмотрим приведение задачи о рюкзаке к задаче линейного програм­мирования. Вспомним формулировку этой задачи из раздела 3.4: дано п предметов весом ..., wn и ценой v\,..., vn, а также рюкзак, выдерживающий вес W. На­ша задача — найти подмножество предметов, которые можно поместить в рюкзак и которые имеют при этом максимальную стоимость. Сначала рассмотрим так называемую непрерывную (continuous), или дробную (fractional), версию задачи, в которой в рюкзак можно помещать произвольную часть любого предмета. Пусть Xj, j = 1,..., п — переменная, представляющая часть предмета j, помещаемую в рюкзак. Очевидно, что Xj должно удовлетворять неравенству 0 ^ Xj ^ 1. Тогда общий вес выбранных предметов можно выразить как сумму Yl7j=iwjxj'> а °^“ щую их стоимость — как vjxj• Таким образом, непрерывную версию задачи о рюкзаке можно представить в виде следующей задачи линейного программиро­вания:

*п*

Максимизировать ^ vjxj з=1

***П***

при условиях wjxj ^

3 **=1**

О ^ Xj ^ 1, j = 1, . . . ,71.

Не имеет смысла применять универсальные методы решения задач линейного программирования для данной конкретной задачи — она решается гораздо проще при помощи специального алгоритма, с которым мы встретимся в разделе 11.3 (но зачем ждать? Попробуйте разработать его самостоятельно). Приведение задачи о рюкзаке к экземпляру задачи линейного программирования имеет смысл только для доказательства корректности рассматриваемого алгоритма.

В так называемой дискретной (discrete), или 0-1, задаче о рюкзаке предмет можно класть в рюкзак только целиком, или не брать его совсем. Следовательно, эта задача приводится к следующей задаче линейного программирования:

*п*

Максимизировать VjXj з=1

***П***

при **УСЛОВИЯХ** ^T^WjXj **^** W

3 = **1**

Х3 6 {Mb J = 1, - - . , ГС.

Такое кажущееся совсем незначительным отличие приводит к кардинальному отличию в сложности этой и подобных задач линейного программирования, в ко­торых переменные могут принимать только дискретные значения. Несмотря на то что 0-1 версия задачи выглядит более простой, так как в ней заведомо отбрасы­ваются все подмножества непрерывной версии, в которых используются дробные части предметов, на самом деле такая задача гораздо сложнее своего непрерывно­го аналога. Читатель, интересующийся алгоритмами решения этой задачи, найдет массу литературы по данному вопросу, включая монографию [77]. ■

Приведение к задачам о графах

Как мы упоминали в разделе 1.3, многие задачи можно решить путем при­ведения к одной из стандартных задач о графах. Это верно, в частности, для множества головоломок и игр. В приложениях такого рода вершины графа обыч­но представляют возможные состояния рассматриваемой задачи, в то время как ребра представляют разрешенные переходы между состояниями. Одна из вершин графа представляет начальное состояние, а некоторая другая — конечное, целевое состояние задачи (таких конечных вершин может быть несколько). Такой граф называется графом пространства состояний (state-space graph). Таким обра­зом, только что описанное преобразование приводит задачу к вопросу о пути из начальной вершины в конечную.

Пример 3. В качестве примера давайте обратимся к классической головоломке о переправе через реку, с которой мы уже встречались в упражнении 1.2.1: на берегу реки находятся крестьянин, волк, коза и кочан капусты. Крестьянин должен перевезти их на другой берег. Однако в лодке только два места — для крестьянина и еще одного объекта (т.е. либо волка, либо козы, либо капусты). В отсутствие крестьянина волк может съесть козу, а коза — капусту. Помогите крестьянину решить эту задачу или докажите, что она не имеет решения.

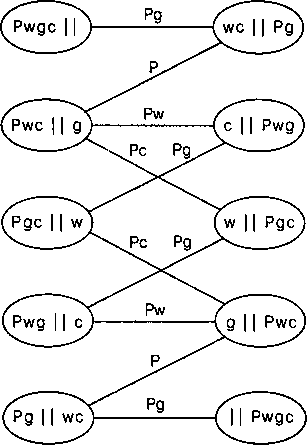


Рис. 6.18. Граф пространства со­стояний для головоломки о пере­праве через реку

Граф пространства состояний для головоломки о переправе через реку пока­зан на рис. 6.18. В вершинах имеются метки, показывающие представляемые ими состояния: Р, w, g и с означают крестьянина, волка, козу и капусту, соответственно (мы оставили английские обозначения, так как на русском языке три персонажа головоломки начинаются с одной буквы... — Прим. перев.)\ две вертикальные

черты (||) символизируют реку. Для удобства мы также пометили ребра, указывая, кто именно находится в лодке при пересечении реки. В терминах данного графа нас интересует поиск пути из начальной вершины, помеченной Pwgc||, в конеч­ную — ||Pwgc.

Легко видеть, что имеется два различных простых пути из начальной вершины в конечную (какие?). Если мы найдем их при помощи поиска в ширину, это будет служить формальным доказательством того, что данные пути — кратчайшие. Следовательно, головоломка имеет два решения, каждое из которых состоит из семи пересечений реки. ■

Успех в решении этой простой головоломки не должен привести нас к убежде­нию, что генерация и исследование графа пространства состояний — всегда про­стая задача. Для лучшего понимания графов пространств состояний обратитесь к книгам по проблеме искусственного интеллекта, области кибернетики, в которой такие задачи являются основным объектом исследования. В книге мы встретимся с важными частными случаями графов пространств состояний в разделах 11.1 и 11.2.

Упражнения 6.6

1. а) Докажите равенство, на котором основан алгоритм поиска lcm (тп, п):

***тп-п***

lcm ***(тп, п) =***

gcd ***(тп, п)'***

б) Как известно, алгоритм Евклида имеет время работы О (log п). Если этот алгоритм используется для поиска gcd (тп,п), то какова эффек­тивность алгоритма поиска lcm (тп, п)?

1. Дано множество чисел, для которого вы должны построить неубываю­щую пирамиду (в неубывающей пирамиде каждый ключ не превышает дочерних ключей). Как использовать алгоритм построения невозраста­ющей пирамиды (пирамиды, определенной в разделе 6.4) для постро­ения неубывающей пирамиды?
2. Докажите, что количество различных путей длиной к > 0 из г-ой вер­шины в j-ую вершину графа (неориентированного или ориентирован­ного) равно (г, j)-My элементу Ак, где А — матрица смежности графа.
3. а) Разработайте алгоритм со временной эффективностью, лучшей ку­

бической, для проверки того, не содержит ли граф с п вершинами цикла длиной 3 [76].

б) Рассмотрим следующий алгоритм для решения этой задачи. Начи­ная с произвольной вершины, обходим граф при помощи поиска

в глубину и проверяем, нет ли в лесу поиска в глубину вершины с обратным ребром, ведущим к “деду”. Если такая вершина есть, граф содержит треугольник; если нет — в графе нет подграфа в виде треугольника. Корректен ли данный алгоритм?

1. На координатной плоскости дано п > 3 точек Pi = (xi, у\),..., Рп — = {хп,Уп)- Разработайте алгоритм для проверки того факта, что все точки лежат внутри треугольника, вершинами которого являются три точки из данного множества. (Вы можете как разработать собственный алгоритм с нуля, так и привести задачу к другой задаче с известным алгоритмом решения.)
2. Рассмотрим задачу поиска для заданного натурального п пары целых чисел, сумма которых равна п и произведение которых максимально. Разработайте эффективный алгоритм решения этой задачи и определи­те его класс эффективности.
3. Задача о назначениях, рассматривавшаяся в разделе 3.4, может быть сформулирована следующим образом. Имеется п работников, которые должны выполнить п заданий, по одному заданию каждый (т.е. каждо­му работнику назначается в точности одно задание, и каждое задание назначается одному человеку). Стоимость выполнения г-ым работни­ком j-ro задания известна и равна С [г, j] для всех пар г, j = 1,... п.

Задача заключается в том, чтобы распределить задания между работ­

никами с наименьшей общей стоимостью. Выразите эту задачу в виде 0-1 задачи линейного программирования.

1. Решите экземпляр задачи линейного программирования, приведенный в разделе 6.6:

Максимизировать О.Юх + 0.07у + О.ОЗг

при условиях х + у + z = 100

z ^ 0.25 (х + у) х ^ 0, у ^ 0, z ^ 0.

1. Задача раскрашивания графа обычно формулируется как задача рас­краски вершин: распределить минимальное количество цветов по вер­шинам данного графа так, чтобы никакие две смежные вершины не были окрашены в одинаковый цвет. Рассмотрим задачу раскрашива­ния ребер: распределить минимальное количество цветов по ребрам данного графа так, чтобы никакие два ребра, имеющие общую точку,

не были окрашены в одинаковый цвет. Поясните, как задача раскраши­вания ребер может быть приведена к задаче раскрашивания вершин.

image160

1. Головоломка о ревнивых мужьях. Имеется п ^ 2 семейных пар, кото­рым надо переправиться через реку. У них есть лодка, в которой может поместиться не более двух человек. Все мужья ревнивы, и не хотят, чтобы их жены оставались без них на берегу реки, где есть хоть один мужчина, жена которого находится на другом берегу реки, — даже ес­ли при этом присутствуют другие люди. Могут ли они переправиться через реку при таких ограничениях?

а) Решите задачу для п = 2.

б) Решите задачу при п = 3 (это и есть классическая версия данной головоломки).

в) Имеет ли головоломка решение для всех п ^ 4? Если да, укажи­те, сколько пересечений реки потребуется для переправы; если нет, поясните, почему.

Резюме

* Метод “преобразуй и властвуй ” — четвертая стратегия разработки ал­горитмов (и решения задач), рассмотренная в книге. Фактически она представляет собой группу методов, основанных на идее преобразова­ния в более простую для решения задачу.
* Имеется три основных варианта стратегии преобразования: упрощение экземпляра, изменение представления и приведение задачи.
* Упрощение экземпляра представляет собой метод преобразования эк­земпляра задачи в экземпляр той же задачи с некоторыми специаль­ными свойствами, которые делают более простым ее решение. Пред­варительная сортировка, исключение Гаусса и AVL-деревья являются хорошими примерами применения этого метода.
* Изменение представления — это преобразование одного представления экземпляра задачи в другое представление того же экземпляра задачи. Примеры использования этого метода, рассмотренные в данной главе, включают представление множества 2-3-деревом, пирамиды и пирами­дальную сортировку, схему Горнера для вычисления полиномов и два алгоритма бинарного возведения в степень.
* Приведением задачи называется преобразование данной задачи в дру­гую задачу, которая может быть решена при помощи известного ал­горитма. Среди примеров применения этой идеи к решению алгорит-

мических задач особенно важное место занимают приведение к задаче линейного программирования и задаче о графе.

* Некоторые примеры, использованные для иллюстрации метода преоб­разования, представляют собой очень важные структуры данных и ал­горитмы. Эт9 пирамиды и пирамидальная сортировка, AVL-деревья и 2-3-деревья, метод исключения Гаусса и схема Горнера.
* Пирамида представляет собой практически полное бинарное дерево с ключами (по одному на узел), удовлетворяющее требованию домини­рования родительских узлов. Будучи определенной как дерево, пирами­да обычно реализуется в виде массива. Пирамиды являются наиболее важной структурой данных при реализации очередей с приоритетами, а также для пирамидальной сортировки.
* Пирамидальная сортировка представляет собой теоретически важный алгоритм сортировки, основанный на расстановке элементов массива в виде пирамиды с последующим последовательным удалением наи­больших элементов пирамид, образующихся в результате удаления. Время работы такого алгоритма равно 0 (n log п) как в среднем, так и в наихудшем случае; кроме того, такая сортировка выполняется без привлечения дополнительной памяти.
* AVL-деревья представляют собой бинарные деревья поиска, которые всегда сбалансированы в той степени, в которой это возможно для бинарных деревьев. Сбалансированность поддерживается путем четы­рех видов преобразований, называющихся поворотами. Все базовые операции над AVL-деревьями выполняются за время 0 (log п); таким образом, в этих деревьях устранена неэффективность классических би­нарных деревьев поиска в наихудшем случае.
* 2-3-деревья достигают идеальной сбалансированности дерева поиска, позволяя узлу содержать до двух упорядоченных ключей и до трех дочерних узлов. Обобщение этой идеи дает очень важные Б-деревья, которые рассматриваются позже в этой книге.
* Метод исключения Гаусса — алгоритм для решения систем линейных уравнений — представляет собой основной алгоритм линейной алгеб­ры. Он решает систему линейных уравнений путем преобразования ее в эквивалентную систему с верхнетреугольной матрицей коэффици­ентов, которая легко решается методом обратной подстановки. Метод исключения Гаусса требует около г? j3 умножений.
* Схема Горнера является оптимальным алгоритмом вычисления поли­нома без предварительной обработки коэффициентов и требует толь­ко п умножений и п сложений. Кроме того, этот алгоритм дает по­лезные побочные результаты, как, например, алгоритм синтетического деления.
* Два алгоритма бинарного возведения в степень для вычисления ап рас­сматриваются в разделе 6.5. В обоих используется бинарное представ­ление показателя степени п, но работают они в разных направлениях: слева направо и справа налево.
* Линейное программирование предназначено для оптимизации линей­ной функции нескольких переменных при ограничениях в виде ли­нейных уравнений и линейных неравенств. Имеются эффективные ал­горитмы, способные решать очень большие экземпляры этой задачи со многими тысячами переменных и ограничений, если только не на­ложено требование целочисленности переменных. В этом случае мы получаем целочисленную задачу линейного программирования, относя­щуюся к классу гораздо более сложных задач.

image161

**Глава**

Пространственно-временной компромисс

Значащее много никогда не должно находиться во власти значащего мало.

— Иоганн Вольфганг фон Гете (Johann Wolfganh von Goethe)

(1749-1832)

Д

остижение компромисса между пространством и временем, т.е. между исполь­зуемой памятью и скоростью работы при разработке алгоритмов — вопрос, хорошо известный как теоретикам, так и практикам алгоритмики. Рассмотрим в качестве примера задачу вычисления значений функции во многих точках ее об­ласти определения. Если более важным является время работы, значения функции могут быть вычислены заранее и храниться в таблице. Это именно то, что делали люди до эпохи компьютеров для повышения скорости вычислений (возможно, кое-кто из вас еще помнит толстые тома математических таблиц). Поиск в таких таблицах в наше время потерял привлекательность благодаря наличию вычисли­тельных машин, но сама их идея весьма ценна при разработке некоторых важных алгоритмов. Говоря более обобщенно, идея состоит в предварительной полной или частичной обработке входных данных с сохранением полученной дополни­тельной информации для ускорения позднейшего решения задачи. Мы назовем такой подход улучшением входных данных1 (input enhancement) и рассмотрим следующие алгоритмы, основанные на этом методе:

* метод сортировки подсчетом (раздел 7.1);
* алгоритм Бойера-Мура для поиска подстрок и его упрощенная версия, предложенная Хорспулом (раздел 7.2).

**Стандартный термин, использующийся для обозначения этого метода, — предварительная об­работка (preprocessing). К сожалению, этот же термин применим и к методам, которые используют идею предварительной обработки, но не используют дополнительную память (см. главу 6). Поэтому, чтобы избежать неоднозначности, мы будем использовать для рассматриваемых здесь методов по­вышения производительности за счет использования памяти термин “улучшение входных данных”.**

Другой метод из этой группы просто использует дополнительную память для того, чтобы обеспечить более быстрый и/или более гибкий доступ к данным. Этот подход мы будем называть предварительной структуризацией (prestructuring). Это название подчеркивает два аспекта этой версии пространственно-временного компромисса: выполнение некоторой обработки до того, как приступить к соб­ственно решению задачи; однако речь здесь идет не об улучшении входных дан­ных, а о структуризации доступа к ним. Этот подход будет проиллюстрирован следующими алгоритмами:

* хеширование (раздел 7.3);
* индексирование при помощи В-деревьев.

Имеется еще один метод разработки алгоритмов, относящийся к рассматрива­емому пространственно-временному компромиссу, — это динамическое програм­мирование (dynamic programming). Эта стратегия основана на записи решений перекрывающихся подзадач данной задачи в таблице, при помощи которой затем получается решение основной задачи. Мы рассмотрим эту хорошо отработанную методику отдельно, в следующей главе книги.

Следует сделать еще два замечания о взаимодействии между временем и про­странством в разработке алгоритмов. Во-первых, эти два ресурса — пространство и время — не обязательно конкурируют друг с другом во всех ситуациях. Могут быть ситуации, когда тщательное проектирование может приводить к алгорит­мическому решению, минимизирующему как время работы, так и используемую память. Такая ситуация возникает, в частности, когда алгоритм применяет для представления входных данных эффективную с точки зрения использования па­мяти структуру данных, что, в свою очередь, приводит к более быстрому алго­ритму. Рассмотрим, например, задачу обхода графов. Вспомним, что временная эффективность двух основных алгоритмов обхода — поиск в ширину и поиск в глу­бину — зависит от структуры данных, используемой для представления графов: она равна 0 (п2) для представления с помощью матрицы смежности, и © (п + т) для представления с применением связанных списков смежности, где п и т — ко­личество вершин и ребер, соответственно. Если входной граф разрежен, т.е. имеет малое количество ребер по отношению к количеству вершин (скажем, те О (п)), то представление с применением связанных списков смежности может оказаться более эффективным как с точки зрения используемой памяти, так и с точки зрения времени работы алгоритма. Та же ситуация возникает и при работе с разрежен­ными матрицами или полиномами: если в таких объектах много нулей, то можно сэкономить и память, и время, если игнорировать нули в представлении объектов и при их обработке.

Во-вторых, невозможно обсуждать пространственно-временной компромисс, не упомянув о такой чрезвычайно важной области, как сжатие данных. Однако следует подчеркнуть, что при сжатии данных снижение размера является целью, а не способом решения другой задачи. В следующей главе мы рассмотрим один алгоритм сжатия данных; читателю, заинтересовавшемуся этой темой, рекомен­дуем книгу [101], в которой имеется богатый выбор таких алгоритмов.

1. Сортировка подсчетом

В качестве первого примера использования метода улучшения входных дан­ных рассмотрим его применение к задаче сортировки. Очевидная идея состоит в том, чтобы подсчитать для каждого элемента сортируемого списка общее коли­чество элементов, меньших данного, и занести полученный результат в таблицу. Полученные числа указывают позиции элементов в отсортированном списке: на­пример, если для некоторого элемента это количество равно 10, то он должен быть одиннадцатым (его индекс будет равен 10, если индексы начинаются с 0) в отсортированном массиве. Таким образом, мы можем отсортировать список, просто копируя его элементы в соответствующие позиции в новом, отсортиро­ванном списке. Такой алгоритм называется сортировкой подсчетом сравнений (comparison counting) (рис. 7.1).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 62 | 31 | 84 | 96 | 19 | 47 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 1 | 0 | 0 |
|  | 1 | 2 | 2 | 0 | 1 |
|  |  | 4 | 3 | 0 | 1 |
|  |  |  | 5 | 0 | 1 |
|  |  |  |  | 0 | 2 |
| 0 | 1 | 4 | 5 | 0 | 2 |

Изначально Count[]

После прохода / = 0 Count[]

После прохода / = 1 Count[]

После прохода / = 2 Count[]

После прохода / = 3 Count[]

После прохода / = 4 Count[]

Конечное состояние Count[]

19 31 47 62 84 96

Массив S[0..5]

Рис. 7.1. Пример сортировки подсчетом сравнений

Алгоритм ***ComparisonCountingSort*** ***(А*** [0..n — 1])

// Сортировка массива подсчетом сравнений

// Входные данные: Массив A[0..n — 1] упорядочиваемых значений // Выходные данные: Массив 5[0..n — 1] элементов А,

// отсортированных в неубывающем порядке

for г 0 to п — 1 do

Count[i] 0 for i <— 0 to ?7 — 2 do for j «— г + 1 to n — 1 do if A [г] < A[j]

***Count[j] \*— Count[j]*** + 1 else

Count[i\ <— Count[i] + 1 for i <— 0 to ?2 — 1 do S[Count[i]] А [г]

return S

Какова же временная эффективность данного алгоритма? Она должна быть квадратична, поскольку алгоритм рассматривает все различные пары п-элемент- ного массива. Более строго, количество выполнений базовой операции, сравнения А [г] < А [j], равно сумме, с которой мы уже не раз встречались:

п—2 гг—1 тг—2 тг—2 , л ч

С<") = Е Е i = EKn-1)-(i+1) + 1i = E(n-1-<) = !iiy^-

г=0 *j=i*+1 *i=0 i=0*

Поскольку алгоритм выполняет то же количество сравнений ключей, что и алго­ритм сортировки выбором, и при этом требуется линейное количество дополни­тельной памяти, его трудно рекомендовать для практического применения.

Однако идея подсчета хорошо работает в ситуации, когда сортируемые эле­менты принадлежат небольшому множеству значений. Предположим, необходимо отсортировать список, в котором могут быть только значения 1 и 2. Вместо при­менения алгоритма сортировки общего назначения мы можем воспользоваться преимуществами, которые дает знание дополнительной информации о сортируе­мых элементах. Можно просто сканировать список, вычисляя количество единиц и двоек, а затем, на втором проходе, сделать соответствующее количество первых элементов равными 1, а остальные — равными 2. Говоря более обобщенно, если значения элементов представляют собой целые числа в границах от I (нижняя граница) до и (верхняя граница), то мы можем вычислить частоты появления каждого из этих значений и сохранить их в массиве F[0..u — I]. Затем первые F [0] позиций в отсортированном списке должны быть заполнены значением Z, следующие F [1] позиций — значением I + 1, и так далее. Все это, конечно, можно делать, только если мы можем перезаписывать данные элементы.

Рассмотрим более реалистическую ситуацию сортировки списка элементов с некоторой сопутствующей информацией, связанной с ключами, так что мы не в состоянии перезаписывать элементы списка. В этом случае мы можем копиро­вать их в новый массив S [0..п — 1]. Те элементы А, чьи значения ключей равны наименьшему возможному значению I, копируются в первые F [0] элементов S, т.е. в позиции от 0 до F [0] — 1, элементы со значением ключа I + 1 — в пози­ции от F [0] до (F [0] + F[1]) — 1, и т.д. Поскольку такие накапливаемые суммы частот в статистике называются распределением, описанный метод известен как сортировка подсчетом распределения (distribution counting).

Пример 1. Рассмотрим сортировку массива

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 13 | 11 | 12 | 13 | 12 | 12 |

значения которого, как нам известно, могут принадлежать множеству {11,12,13} и не должны перезаписываться в процессе сортировки. Массивы частот и распре­делений выглядят следующим образом:

|  |  |  |  |
| --- | --- | --- | --- |
| Значения массива | 11 | 12 | 13 |
| Частоты | 1 | 3 | 2 |
| Значения распределения | 1 | 4 | 6 |

Заметим, что значения распределения указывают корректные позиции послед­них элементов с данным значением в отсортированном массиве (если массив индексируется начиная с 0, то значения распределения для получения корректных позиций должны быть уменьшены на 1).

Более удобно обрабатывать входной массив справа налево. Например, послед­нее значение равно 12, а поскольку значение распределения для него равно 4, мы помещаем 12 в позицию 4—1 = 3 массива S', в котором хранится отсортированный список. Затем мы уменьшаем значение распределения 12 на 1 и переходим к сле­дующему (в направлении справа налево) элементу исходного массива. Полностью рассматриваемый пример показан на рис. 7.2. ■

D[0..2] S[ 0..5]

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Л[5] = 12 | 1 | 4 | 6 |  |  |  |  | 12 |  |  |
| Л[4] = 12 | 1 | 3 | 6 |  |  |  | 12 |  |  |  |
| Л[3]=13 | 1 | 2 | 6 |  |  |  |  |  |  | 13 |
| Л[2] = 12 | 1 | 2 | 5 |  |  | 12 |  |  |  |  |
| Л [ 1 ] = 11 | 1 | 1 | 5 |  | 11 |  |  |  |  |  |
| Л[0]=13 | 0 | 1 | 5 |  |  |  |  |  | 13 |  |

Рис. 7.2. Пример сортировки подсчетом распределе­ния. Уменьшаемые значения распределения выделены полужирным шрифтом

Вот псевдокод рассмотренного алгоритма.

**АЛГОРИТМ *DistributionCounting*** **(*А*** **[0..n** — **1], Z, *и)***

// Сортировка массива целых чисел из ограниченного диапазона // при помощи сортировки подсчетом распределения // Входные данные: Массив A[0..n — 1] целых чисел

// между *I* ии(1 ^ и)

// Выходные данные: Массив S[0..n — 1] элементов А,

отсортированных в неубывающем порядке

//

image162

for г <— 0 to п — 1 do

// Инициализация частот

// Вычисление частот

// Получение распределения

***D{A[i\ -I]\*- D[A[i] -l} +*** 1

**for j** <— 1 **to u — I** **do**

D\j]\*-D\j-1] + D\j]

**for** г n — **1 downto 0 do**

j «- A[i] - I S[D\j] - 1] - АЩ

m - m -1

**return S**

В предположении фиксированного диапазона значений очевидно, что этот ал­горитм линеен, поскольку выполняет два последовательных прохода по входному массиву А. Это лучший класс эффективности, чем у самых эффективных алго­ритмов сортировки сравнением (слиянием, быстрой и пирамидальной). Однако очень важно помнить, что эта эффективность получена благодаря конкретному виду входных данных, для которых работает сортировка подсчетом распределе­ния (помимо использования дополнительной памяти).

Упражнения 7.1

1. Можно ли обменять числовые значения двух переменных без исполь­зования дополнительной памяти?
2. Будет ли сортировка подсчетом сравнений корректно работать для мас­сивов с одинаковыми значениями?
3. В предположении, что возможные входные значения ограничены мно­жеством {а, Ь, с, d}, отсортируйте в алфавитном порядке при помощи алгоритма сортировки подсчетом распределения следующие входные данные:

Ь, с, d, с, Ь, а, а, Ъ.

1. Является ли алгоритм сортировки подсчетом распределения устойчи­вым?
2. Разработайте однострочный алгоритм для сортировки массива разме­ром п, содержащего различные целые значения от 1 до п.
3. Задача о Российском флаге заключается в перестановке элементов массива символов К, С, Б (означающие красный, синий и белый цвета

флага) так, чтобы первыми шли символы Б, затем — С, а последни­ми — К. Разработайте алгоритм, решающий поставленную задачу за линейное время без привлечения дополнительной памяти.

1. Задача о родословной (ancestry problem) заключается в определении, является ли вершина и предком вершины v в данном бинарном (или, в общем случае, корневом упорядоченном) дереве из п вершин. Раз­работайте алгоритм с улучшением входных данных и эффективностью О (п), который обеспечивает достаточное количество информации для решения этой задачи для любой пары вершин дерева за постоянное время.
2. Описанный далее метод под названием виртуальной инициализации (virtual initialization) обеспечивает эффективный способ инициализи­ровать только некоторые из элементов данного массива А [0..п] так, чтобы мы могли для каждого из его элементов за постоянное время сказать, был ли он инициализирован, и если да, то каким значением. Это делается при помощи переменной counter, хранящей количество инициализированных элементов А, и двух вспомогательных массивов того же размера, В [0..n — 1] и С [0..п — 1], определенных следующим образом. Элементы В [0],..., В [counter — 1] содержат индексы тех элементов А, которые были инициализированы: В [0] содержит индекс элемента А, инициализированного первым, В [1] — индекс элемента А, инициализированного вторым, и т.д. Кроме того, если А [г] — элемент, инициализированный fc-ым 0 ^ к ^ countei— 1), то С [г] равен к.

а) Набросайте состояние массивов А [0..7], В [0..7] и С [0..7] после трех присваиваний А [3] <— х; А [7] \*— z и А [1] \*— у.

б) Как при помощи этой схемы в общем случае можно определить, был ли инициализирован элемент АЩ, и если да, то каким значением?

1. а) Напишите программу для умножения двух разреженных матриц —

image163

матрицы А размером р х q, и матрицы В размером q х р. б) Напишите программу для умножения двух разреженных полиномов р (х) и q (х) степени тип, соответственно.

1. Напишите программу, которая играет в крестики-нолики с человеком, путем хранения всех возможных позиций игры на доске 3x3 вместе с наилучшим ходом в данной позиции.
2. Улучшение входных данных в поиске подстрок

В этом разделе мы увидим, как метод улучшения входных данных можно при­менить к задаче поиска подстрок. Вспомним, что задача поиска подстрок состоит в поиске данной строки из т символов (именуемой шаблоном, или образцом (pat­tern)) в более длинной строке из п символов (называемой текстом (text)). Мы рассматривали алгоритм решения данной задачи на основе грубой силы в разде­ле 3.2: просто проверять равенство в соответствующих парах символов из образца и текста слева направо и при обнаружении различий сдвинуть образец на один символ для выполнения новой проверки. Поскольку наибольшее количество та­ких проверок — п — 77г+1ив наихудшем случае при каждой проверке требуется выполнить т сравнений, общее количество сравнений в наихудшем случае равно т (п — т + 1), так что производительность алгоритма грубой силы в наихудшем случае равна © (пт). В среднем, однако, можно ожидать, что при каждой про­верке будет сделано только несколько сравнений; в самом деле, для случайного естественного текста эффективность в среднем случае превращается в 0 (п).

Высокая производительность алгоритма на основе грубой силы в среднем слу­чае — это и хорошо, и плохо одновременно. Это хорошо с практической точки зрения, поскольку делает алгоритм на основе грубой силы вполне достойным кан­дидатом для практического применения, в особенности при коротких образцах. Но это плохо для теоретика, который хочет найти более быстрый алгоритм. Тем не менее был найден ряд более быстрых алгоритмов, большинство из которых использует идею улучшения входных данных: предварительная обработка образ­ца для получения некоторой информации о нем, сохранение этой информации в таблице, а затем ее использование при реальном поиске этого образца в дан­ном тексте. Эта идея лежит в основе двух наиболее известных алгоритмов этого типа — алгоритме Кнута-Морриса-Пратта [68] и алгоритме Бойера-Мура [21].

Основное различие между этими двумя алгоритмами лежит в способе срав­нивания символов образца и текста: алгоритм Кнута-Морриса-Пратта делает это слева направо, а алгоритм Бойера-Мура — справа налево. Поскольку последний способ приводит к более простому алгоритму, мы рассмотрим здесь именно его. (Заметим, что алгоритм Бойера-Мура начинает с выравнивания образца по пер­вому символу текста; если первая проверка неудачна, образец сдвигается вправо. Проверка же выполняется путем сравнения символов образца и текста справа на­лево, начиная с последнего символа образца.) Несмотря на то что идея, лежащая в основе алгоритма Бойера-Мура, проста, этого не скажешь о его реализации. Поэтому мы начнем рассмотрение с упрощенной версии алгоритма Бойера-Мура, предложенной Р. Хорспулом (R. Horspool) [55]. Несмотря на простоту, алгоритм

Хорспула при работе со случайными строками столь же эффективен, как и алго­ритм Бойера-Мура.

Алгоритм Хорспула

Рассмотрим в качестве примера поиск подстроки BARBER в некотором тексте:

so с • • • 1

***BARBER***

Мы сравниваем соответствующие пары символов из образца и текста, начиная с последнего символа R в образце и перемещаясь справа налево. Если все сим­волы образца соответствуют символам текста, искомая подстрока найдена (после этого поиск может либо завершиться, либо продолжиться, если требуется най­ти другое вхождение подстроки в текст). Если же мы встретили несоответствие, то должны сдвинуть образец вправо. Понятно, что хотелось бы сдвинуть его как можно сильнее, но без риска пропустить возможное вхождение подстроки в текст. Алгоритм Хорспула определяет величину такого сдвига, рассматривая символ с текста, который при выравнивании находится напротив последнего символа об­разца. В общем случае могут возникнуть четыре разные ситуации.

Случай 1. Если символа с в образце нет (например, если с в данном примере представляет собой символ S'), то смело можно сдвигать образец на всю его длину (при сдвиге меньшей величины против символа с окажется некоторый символ образца, который заведомо не может быть таким же, как с, которого в образце нет):

so S • • • 1

***BARBER***

***BARBER***

Случай 2. Если символ с в образце есть, но он не последний (например, символ В в нашем примере), то сдвиг должен выровнять образец так, чтобы напротив с в тексте было первое справа вхождение этого символа в образец:

**So • • •** в **• ‘ • sn\_i**

***BARBER***

***BARBER***

Случай 3. Если с — последний символ образца и среди остальных тп — 1 сим­волов образца такого символа больше нет, то сдвиг должен быть подобен сдвигу

в случае 1 — образец следует сдвинуть на всю длину тп:

***sq • • • М Е R • • • s***n—1

it II II

***LEADER***

***LEADER***

|  |  |  |
| --- | --- | --- |
| О | R | $п— 1 |
| it | II |  |
| Е | R |  |
| О | R D Е R |  |

Случай 4. И, наконец, если с — последний символ образца и среди остальных т — 1 символов образца имеются другие вхождения этого символа, то сдвиг должен быть подобен случаю 2 — крайнее справа вхождение с среди остальных т-1 символов образца должно располагаться напротив символа с в тексте:

***So***

***R Е О R D R Е***

Эти примеры ясно демонстрируют, что сравнение символов справа налево может привести к большим сдвигам, чем сдвиги на одну позицию в алгоритме на основе грубой силы. Однако если такой алгоритм будет просматривать все символы образца при каждой проверке, то все его преимущество будет потеряно. К счастью, идея улучшения входных данных делает такой просмотр при каждой проверке излишним. Мы можем предварительно вычислить величины сдвигов для каждого возможного символа и хранить их в таблице. Такая таблица ин­дексируется всеми возможными символами, которые могут встретиться в тексте, включая, для естественных языков, пробелы, символы пунктуации и другие спе­циальные символы (заметим, что никакой иной информации о тексте, в котором будет выполняться поиск, не требуется). Элементы таблицы заполняются величи­нами сдвигов. В частности, для каждого символа с мы можем вычислить величину сдвига по формуле

***t (с) =*** <

Длина образца тп, если с нет среди первых 771 — 1 символов образца

(7.1)

В противном случае — расстояние от крайнего

справа символа с среди первых 771 — 1 символов образца до его последнего символа

Например, для образца BARBER все элементы таблицы равны 6, за исключе­нием элементов для символов Е, В, R и А, для которых они равны 1, 2, 3 и 4,

соответственно.

Вот простой алгоритм для вычисления элементов таблицы сдвигов. Все зна­чения в таблице инициализируются длиной образца га, а затем выполняется ска­нирование образца слева направо с выполнением га — 1 раз следующих действий: для j-ro символа образца 0 ^ j ^ га — 2) соответствующий ему элемент таблицы перезаписывается значением га — 1 — j, которое представляет собой расстояние от символа до правого конца образца. Заметим, что, поскольку алгоритм сканиру­ет образец слева направо, последняя перезапись выполняется, когда встречается самое правое вхождение символа в образец, т.е. именно так, как требуется.

Алгоритм ShiftTable (Р [0..га - 1])

// Заполняет таблицу сдвигов, использующуюся алгоритмами // Хорспула и Бойера-Мура

// Входные данные: Образец Р[0..тп — 1] и алфавит возможных

// символов

// **Выходные данные:** Таблица Table[0..size **—** 1],

// индексированная символами алфавита

// и заполненная величинами сдвигов,

// вычисленных по формуле (7.1)

Инициализация всех элементов Table значениями га for j <— 0 to 771 — 2 do Table[P[j]\ <— га — 1 — j return Table

А вот как выглядит алгоритм Хорспула целиком.

Алгоритм Хорспула

Шаг 1. Для данного образца длиной га и алфавита, используемого в тексте и об­разце, описанным выше способом строится таблица сдвигов.

Шаг 2. Выравниваем начало образца с началом текста.

Шаг 3. До тех пор, пока не будет найдена искомая подстрока или пока образец не достигнет последнего символа текста, повторяем следующие действия. Начиная с последнего символа образца, сравниваем соответствующие символы в шаблоне и тексте, пока не будет установлено равенство всех т символов (при этом поиск прекращается) либо пока не будет обна­ружена пара разных символов. В последнем случае находим элемент t (с) из таблицы сдвигов, где с — символ текста, находящийся напротив последнего символа образца, и сдвигаем образец вдоль текста на t (с) символов вправо.

Вот как выглядит псевдокод алгоритма Хорспула.

АЛГОРИТМ Ног spool Matching (Р [0..m — 1], Т [0..тг — 1])

// Реализация алгоритма Хорспула поиска подстрок // Входные данные: Образец P[0..m — 1] и текст Т[0..п — 1]

// Выходные данные: Индекс левого конца первой найденной

// подстроки или —1, если искомой подстроки

// в тексте нет

ShiftTable(P[0..m — 1]) // Генерация таблицы сдвигов Table

i <— m — 1 // Позиция правого конца образца

**while** г ^ **п** — 1 **do**

к <— 0 // Количество совпадающих символов

while к ^ m — 1 and P[m — 1 — к] = T[i — fc] do

к <— к + 1 if к — m

**return** г — m**-f 1 else**

г <— г + ***Table[T[i]]* return — 1**

Пример 1. Рассмотрим в качестве законченного приложения алгоритма Хорспула поиск образца BARBER в тексте, состоящем из английских букв и пробелов (которые указаны символами подчеркивания). Таблица сдвигов для этого случая, как уже говорилось, выглядит следующим образом:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Символ с | А | В | С | D | Е | F |  | R |  | Z |  |
| Сдвиг t (с) | 4 | 2 | 6 | 6 | 1 | 6 | 6 | 3 | 6 | 6 | 6 |

Реальный поиск в конкретном тексте выглядит так:

***J I М \_ S A W \_ М Е \_ I N\_A\_BARBERSHOP BARBER BARBER***

BARBER ***BARBER***

BARBER *BARBER* и

На простом примере можно продемонстрировать, что эффективность алго­ритма Хорспула в наихудшем случае составляет 0 (пти) (см. упражнение 7.2.4). Однако для случайных текстов эффективность алгоритма Хорспула равна 0 (п). Таким образом, хотя алгоритм Хорспула принадлежит к тому же классу эффек­тивности, что и алгоритм, основанный на грубой силе, очевидно, что в среднем он превосходит последний. В действительности, как уже упоминалось, зачастую он столь же эффективен, как и его более сложный предшественник, открытый Р. Бойером (R. Boyer) и Дж. Муром (J. Moore).

Алгоритм Бойера-Мура

В этом разделе мы рассмотрим работу алгоритма Бойера-Мура. Если первое сравнение крайнего справа символа в шаблоне с соответствующим символом с в тексте показывает, что они различны, алгоритм работает точно так же, как и ал­горитм Хорспула, т.е. выполняется сдвиг вправо на количество символов, которое определяется таблицей сдвигов (которая вычисляется в точности так же, как и для алгоритма Хорспула). Однако алгоритмы Бойера-Мура и Хорспула работают по- разному, если некоторое положительное количество к (0 < к < т) символов образца совпадает с символами текста, перед тем как встретится первое отличие:

s о ... с ... ... Sn—1 Текст

# II II

Ро ■■■ Рт-к-1 Рт-к • • • Pm-1 Образец

sn\_ i Текст

Образец

В этой ситуации алгоритм Бойера-Мура определяет величину сдвига, рассмат­ривая две величины. Величина первого сдвига определяется символом текста с, который первым не соответствует символу образца при сравнении справа налево. Назовем его сдвигом несовпадающего символа (bad-symbol shift). Рассмотрение данного сдвига выполняется так же, как и сдвига в алгоритме Хорспула. Если символ с не входит в образец, мы сдвигаем образец так, чтобы символ с вышел за пределы образца. Значение этого сдвига легко вычислить по формуле t\ (с) — к, где t\ (с) — элемент таблицы сдвигов, использующейся в алгоритме Хорспула, а к — количество совпадающих символов:

**SO С Si\_fc\_|\_i ... Si ...**

-It II II

РО • • • Pm—к—1 Pm—к • • • Pm— 1

РО • • • Pm—I

Например, если мы ищем образец BARBER в некотором тексте и находим совпадение двух последних символов перед обнаружением в тексте символа S, то можем сдвинуть образец на t\ (S) — 2 = 6 — 2 = 4 позиции:

so .. • S Е R ... sn-1

tt ii ii

***BARBER***

***BARBER***

Та же формула используется и в случае, когда несовпадающий символ с имеет­ся в образце и при этом t\ {с)—к > 0. Например, если мы ищем образец BARBER

в некотором тексте и находим совпадение двух последних символов перед обнару­жением в тексте символа А, то можем сдвинуть образец на ti(А) — 2 = 4 — 2 = 2 позиции:

***sq ... А Е R*** ... sn-i

# II II

***BARBER***

***BARBER***

Если t\ (с) — к ^ 0, очевидно, что нельзя сдвигать образец на нулевое или отрицательное количество позиций. Вместо этого мы просто применяем метод грубой силы и сдвигаем образец на одну позицию вправо. Итак, сдвиг несовпада­ющего символа di, вычисляемый в алгоритме Бойера-Мура, равен либо t\ (с) — fc, если эта величина положительна, либо 1, если она отрицательна или равна нулю. Сказанное можно выразить простой компактной формулой:

d\ = max {ti (с) — fc, 1} . (7.2)

Сдвиг второго типа определяется совпадением последних к > 0 символов образца. Мы будем называть конечную часть образца его суффиксом длиной к и обозначать как suff(k). Соответственно, этот тип сдвига мы будем называть сдвигом совпадающего суффикса (good-suffix shift). Сейчас мы применим рас­суждения, которые позволили нам заполнить таблицу сдвигов несовпадающих символов на основании одного символа с, к построению таблицы сдвигов совпа­дающих суффиксов на основании суффиксов с длинами 1,..., т — 1.

Начнем с рассмотрения случая, когда в образце встречается последователь­ность символов такая же, как и suff(k), но не являющаяся суффиксом; говоря бо­лее строго, имеется другая последовательность suff{k), которую предваряет сим­вол, отличный от символа, предваряющего последнюю такую последовательность символов (было бы бесполезно сдвигать образец так, чтобы на место предыдущей последовательности suff(k) попала новая последовательность с таким же пред­шествующим символом — в этой ситуации мы бы просто повторили неудачную попытку сравнения). В таком случае мы можем сдвинуть образец на расстояние с?2 между второй справа последовательностью suff(k) (не предваряемой тем же символом, что и последняя) и такой же последовательностью, крайней справа в образце. Например, для образца ABC В АВ эти расстояния для k = 1 и 2 будут равны, соответственно, 4 и 6:

|  |  |  |
| --- | --- | --- |
| к | Образец | d2 |
| 1 | АВСВАВ | 2 |
| 2 | АВСВАВ | 4 |

Что следует предпринять, если в образце нет другой последовательности suff(k), которой предшествует символ, отличный от символа, предшествующе­го суффиксу suff(k)l В большинстве случаев мы можем сдвинуть образец на всю его длину т. Например, для образца DBCBAB и к = 3 мы можем выполнить сдвиг образца на всю его длину — 6 символов:

so ... с В А В ... sn-1

# I! II II

***DBCBAB***

***DBCBAB***

К сожалению, такой сдвиг образца на всю его длину т, если в образце нет другой последовательности suff(k), которой предшествует символ, отличный от символа, предшествующего суффиксу suff(k), не всегда корректен. Например, для образца АВСВАВ и к = 3 сдвиг на 6 может привести к пропуску подстроки, которая начинается с символов АВ, расположенных в тексте напротив двух по­следних символов образца:

***so ... сВАВСВАВ*** ... ***sn~*** **1**

it **ii ii ii**

***АВСВАВ***

***АВСВАВ***

Заметим, что сдвиг на 6 символов корректен для образца DBCBAB, но не для образца АВСВАВ, поскольку у него имеется префикс АВ, совпадающий с суф­фиксом. Чтобы избежать такого некорректного сдвига на основании суффикса длиной к, следует найти наибольший префикс длиной I < к, совпадающий с суф­фиксом той же длины I. Если такой префикс имеется, величина сдвига d,2 вы­

числяется как расстояние между префиксом и суффиксом; в противном случае устанавливается равным длине образца га. В качестве примера рассмотрим пол­ный список значений d<i — таблицу сдвигов совпадающих суффиксов алгоритма Бойера-Мура — для образца АВСВАВ:

|  |  |  |
| --- | --- | --- |
| к | Образец | d2 |
| 1 | АВСВАВ | 2 |
| 2 | АВСВАВ | 4 |
| 3 | АВСВАВ | 4 |
| 4 | АВСВАВ | 4 |
| 5 | АВСВАВ | 4 |

Теперь мы готовы к тому, чтобы полностью описать алгоритм Бойера-Мура.

**Алгоритм Бойера-Мура**

Шаг 1. Для данного образца и используемого алфавита описанным выше спосо­бом строится таблица сдвигов несовпадающих символов.

Шаг 2. Для данного образца описанным выше способом строится таблица сдви­гов совпадающих суффиксов.

Шаг 3. Выравниваем начало образца с началом текста.

Шаг 3. До тех пор, пока не будет найдена искомая подстрока или пока образец не достигнет последнего символа текста, повторяем следующие действия. Начиная с последнего символа образца, сравниваем соответствующие символы в шаблоне и тексте, пока не будет установлено равенство всех т символов (при этом поиск прекращается) либо пока не будет обна­ружена пара разных символов после к ^ 0 совпадающих символов. В последнем случае находим элемент t\ (с) из таблицы сдвигов несов­падающих символов, где с — не совпавший символ текста. Если к > О, выбираем, кроме того, соответствующее значение cfc из таблицы сов­павших суффиксов. Сдвигаем образец вправо на количество позиций, которое вычисляется по формуле

d=U если к = 0, (73)

[ max {di, g?2 } если к > 0,

где d\ = max {t\ (с) — fc, 1}.

Сдвиг на максимальное из двух допустимых значений логичен. Оба значения получены из наблюдений (одно за не совпавшим символом, второе — за группой совпавших символов), что меньшие значения сдвигов не могут привести к иско­мой подстроке в тексте. Поскольку нас интересуют как можно большие сдвиги без возможной потери искомой подстроки, выбираем большее из найденных зна­чений.

Пример 2. В качестве завершенного примера давайте рассмотрим поиск под­строки BAOBAB в тексте, состоящем из английских букв и пробелов. Таблица сдвигов несовпадающих символов выглядит следующим образом:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| с | А | В | С | D |  | О |  | Z |  |
| ti (с) | 1 | 2 | 6 | 6 | 6 | 3 | 6 | 6 | 6 |

Таблица сдвигов совпадающих суффиксов имеет такой вид:

к Образец d2

~1 BAOBAB 2~~

1. ***BAOBAB*** 5
2. ***BAOBAB*** 5
3. ***BAOBAB*** 5
4. ***BAOBAB*** 5

Поиск при помощи алгоритма Бойера-Мура с использованием приведенных таблиц показан на рис. 7.3. После того как последний символ образца В при сравнении не совпадает с символом К в тексте, алгоритм получает значение t\ (К) = 6 из таблицы сдвигов несовпадающих символов и перемещает обра­зец на расстояние d\ = max{£i (J4T) — 0,1} = 6 позиций вправо. Новая попытка находит два совпадающих символа, и после неудачного третьего сравнения сим­волов алгоритм получает t\ (\_) = 6 из таблицы сдвигов несовпадающих символов и d2 = 5 — из таблицы сдвигов совпадающих суффиксов и сдвигает образец впра­во на max{di,<i2} = шах{6 — 2,5} = 5 символов. Обратите внимание, что на этой итерации правило совпадающих суффиксов дает большее значение сдвига, чем правило последнего несовпадающего символа.

***BESS \_ К N Е W \_ ABOUT \_BAOBABS***

***BAOBAB***

***d\ = ti (К)*** — 0 = 6 ***В А О В А В***

***di = ti*** (J - 2 = 4 ***В А О В А В***

***d2 =*** 5 ***dl=ti(\_)~*** 1 = 5

d = max (4,5} = 5 d2 = 2

d = max (5,2} = 5

BAOBAB Рис. 7.3. Пример поиска подстроки при помощи алгоритма Бойера-Мура

Очередная проверка успешно находит одну пару совпадающих символов В. После неудачности следующего сравнения с пробелом в тексте алгоритм полу­чает из таблицы сдвигов несовпадающих символов значение t\ (\_) = 6, а из таблицы сдвигов совпадающих суффиксов — d2 = 2 и затем сдвигает образец на max {d\, d2} = max (6 — 2,5} = 5 символов. На этой итерации, как видите, правило совпадающих суффиксов дает меньшее значение сдвига, чем правило по­следнего несовпадающего символа. После выполнения сдвига очередная проверка дает совпадение всех шести символов образца с соответствующими символами текста. ■

При поиске первого вхождения образца в текст эффективность алгоритма Бойера-Мура в наихудшем случае оказывается линейной. Несмотря на скорость данного алгоритма, в особенности в случае больших алфавитов (по отношению к длине образца), многие, работая с текстами на естественных языках, предпочи­тают упрощенные версии данного алгоритма, такие как алгоритм Хорспула.

Упражнения 7.2

1. Примените алгоритм Хорспула к поиску подстроки BAOBAB в тексте BESS\_KNEW\_ABO UT\_BA OBABS.
2. Рассмотрите задачу поиска генов в последовательности ДНК с исполь­зованием алгоритма Хорспула. Последовательность ДНК представляет собой текст на алфавите {A, C,G,T}, а ген или отрезок гена — образец поиска.

а) Постройте таблицу сдвигов для отрезка гена ТССТАТТСТТ.

б) Примените алгоритм Хорспула для поиска этого образца в последо­вательности ДНК

***ТТА ТА G A TCTCG ТА ТТСТТТТА ТА GA ТС ТСС ТА ТТСТТ.***

1. Сколько сравнений символов будет выполнено алгоритмом Хорспула при поиске каждого из следующих образцов в тексте, состоящем из 1000 нулей?

а) 00001 б) 10000 в) 01010

1. Для поиска образца длиной m в тексте длиной п (п > т) при помощи алгоритма Хорспула приведите пример

а) наихудшего случая б) наилучшего случая

1. Может ли алгоритм Хорспула выполнить большее количество сравне­ний символов, чем алгоритм на основе грубой силы, при поиске того же образца в том же тексте?
2. Если алгоритм Хорспула находит искомую подстроку, какой сдвиг он должен сделать для продолжения поиска следующего вхождения под­строки в текст?
3. Сколько сравнений символов будет выполнено алгоритмом Бойера- Мура при поиске каждого из следующих образцов в тексте, состоящем из 1000 нулей?

а) 00001 б) 10000 в) 01010

1. а) Будет ли корректно работать алгоритм Бойера-Мура при использо­

вании только лишь таблицы сдвигов несовпадающих символов для определения величины очередного сдвига образца? б) Будет ли корректно работать алгоритм Бойера-Мура при использо­вании только лишь таблицы сдвигов совпадающих суффиксов для определения величины очередного сдвига образца?

1. а) Если последний символ образца и соответствующий ему символ тек­

ста совпадают, должен ли алгоритм Хорспула проверять остальные символы справа налево или он может проверять их слева направо? б) Ответьте на тот же вопрос для алгоритма Бойера-Мура.

1. Реализуйте алгоритмы Хорспула, Бойера-Мура и алгоритм на основе грубой силы из раздела 3.2 на языке программирования по вашему выбору и проведите эксперимент по сравнению их эффективности для поиска

а) случайного бинарного образца в случайном бинарном тексте;

б) случайный образец на естественном языке в случайном тексте на том же естественном языке.

1. Хеширование

В этом разделе мы рассмотрим очень эффективный способ реализации слова­рей. Вспомним, что словарем называется абстрактный тип данных, представляю­щий собой множество с операциями поиска, вставки и удаления, определенных над его элементами. Элементы такого множества могут иметь любую природу — быть числами, символами некоторого алфавита, строками и т.д. На практике наи­более важным случаем являются записи (записи с информацией о студентах в ин­ституте, о гражданах — в муниципальных органах, о книгах — в библиотеке и т.п.).

Обычно записи включают в себя несколько полей, каждое из которых отве­чает за хранение определенного рода информации об объекте, представленном записью. Например, запись о студенте может содержать поля с идентификатором студента, его фамилией, датой рождения, полом, домашним адресом и т.п. Среди полей записи имеется по крайней мере одно, именуемое ключом (key) и ис­пользуемое для идентификации объектов, представленных записями (например, идентификатор студента). В данном разделе мы полагаем, что должны разработать словарь из п записей с ключами К\, К2,..., Кп.

Хеширование (hashing) основано на идее распределения ключей в одномер­ном массива Н [0..m — 1], называющемся хеш-таблицей (hash table). Распреде­ление осуществляется путем вычисления для каждого ключа значения некоторой предопределенной функции h, которая называется хеш-функцией (hash function).

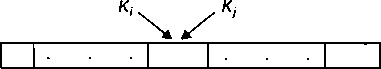
Эта функция назначает каждому из ключей хеш-адрес (hash address), который представляет собой целое число от 0 до тп — 1. Например, если ключи пред­ставляют собой неотрицательные целые числа, то хеш-функция может иметь вид h{K) — К mod тп (очевидно, что остаток от деления на тп всегда находится в диа­пазоне от 0 до тп — 1). Если ключи — символы некоторого алфавита, то сначала можно назначить символам их позиции в алфавите (обозначаемые как ord (К)), а затем применить к этим значениям ту же функцию, что и для целых чисел. Нако­нец, если К — символьная строка coci... cs\_i, мы можем использовать (в качестве простейшей и не очень разумной возможности) значение (Х^=о ord(ci)) mod га; существенно лучшим вариантом является вычисление h (К) по формуле[[51]](#footnote-51)

h <— 0; for г <— Otos — ldo/i\*— (h • С + ord (c\*)) mod ra,

где С — константа, большая, чем любое из значений ord (q).

В общем случае хеш-функция должна удовлетворять двум несколько противо­речивым требованиям:

* хеш-функция должна распределять ключи по ячейкам хеш-таблицы как можно более равномерно. (Исходя из этого требования га обычно выбирается простым. Это же требование делает желательной для боль­шинства приложений зависимость хеш-функции от всех битов ключа, а не только от некоторых из них.)
* Хеш-функция должна легко вычисляться.



0 Ь т-1

Рис. 7.4. Коллизия двух ключей при хеши­ровании: h(Ki) = h(Kj)

Очевидно, что, выбрав размер хеш-таблицы тп меньшим, чем количество клю­чей п, мы обречены на коллизии (collision) — ситуации, когда два (или более) ключей хешируются в одну и ту же ячейку хеш-таблицы (рис. 7.4). Но колли­зий следует ожидать, даже если га значительно больше п (см. упражнение 7.3.5). В действительности в наихудшем случае все ключи могут быть хешированы в од­ну ячейку хеш-таблицы. К счастью, при соответствующем выборе размера хеш- таблицы и хорошей хеш-функции такая ситуация встречается очень редко. Тем не менее любая схема хеширования должна иметь механизм разрешения коллизий.

Этот механизм различен в двух основных версиях хеширования — открытом хешировании (open hashing) (его называют также хешированием с раздельными цепочками (separate chaining)) и закрытом хешировании (closed hashing) (назы­ваемом также хешированием с открытой адресацией (open addressing)).

Открытое хеширование (раздельные цепочки)

При открытом хешировании ключи хранятся в связанных списках, присоеди­ненных к ячейкам хеш-таблицы. Каждый список содержит все ключи, хеширо­ванные в данную ячейку. Рассмотрим, например, следующий список слов:

***A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED.***

В качестве хеш-функции мы будем использовать простую функцию для строк, упоминавшуюся выше, т.е. суммировать позиции в алфавите букв, составляющих слова, и вычислять остаток от деления этой суммы на 13.

Начнем с пустой таблицы. Первый ключ — слово А; его хеш-значение h (А) = = 1 mod 13 = 1. Второй ключ — слово FOOL — попадает в девятую ячей­ку (поскольку (6 + 15 + 15 + 12) mod 13 = 9), и т.д. Окончательный резуль­тат этого процесса показан на рис. 7.5. Обратите внимание на коллизию клю­чей ARE и SOON (вызванную тем, что h(ARE) = (1 + 18 + 5) mod 13 = И и h {SOON) = (19 + 15 + 15 + 14) mod 13 = 11).

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ключи | А | FOOL | AND | HIS | MONEY | ARE | SOON | PARTED |
| Хеш-адреса | 1 | 9 | 6 | 10 | 7 | 11 | 11 | 12 |
| 0 12 3 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

A AND MONEY FOOL HIS ARE PARTED

***I***

SOON

Рис. 7.5. Пример хеш-таблицы с раздельными цепочками

Каким образом осуществляется поиск в такой таблице связанных списков? Для этого к ключу поиска применяется та же процедура, что и при создании таблицы. Для иллюстрации: если мы хотим найти в приведенной на рис. 7.5 хеш-таблице слово KID, то должны начать с вычисления для него хеш-функции: h {KID) = 11. Поскольку список, присоединенный к ячейке И, не пуст, он может содержать соответствующий ключ. Однако из-за возможных коллизий мы не можем сказать, так ли это, пока не обойдем весь список. После сравнения слова KID сначала со словом ARE, а затем со словом SOON мы убеждаемся, что искомого слова в таблице нет.

В общем случае эффективность поиска зависит от длины связанных списков, которая, в свою очередь, зависит от размеров словаря и таблицы и качества хеш- функции. Если хеш-функция распределяет п ключей по т ячейкам равномерно (или практически равномерно), то в каждом списке будет содержаться пример­но около п/т ключей. Отношение а = п/т называется коэффициентом за­полнения (load factor) хеш-таблицы и играет ключевую роль в эффективности хеширования. В частности, среднее количество проверяемых узлов цепочек при успешном поиске S и при неудачном U равны, соответственно

5~1 + ^и[/ = а (7.4)

(при стандартных предположениях о поиске случайно выбранного элемента и хеш- функции, равномерно распределяющей ключи по ячейкам таблицы). Полученные результаты вполне естественны и, само собой, практически идентичны поиску в связанном списке — все, чего мы достигаем при помощи хеширования, — это снижение среднего размера связанного списка в т раз.

Обычно желательно, чтобы коэффициент заполнения был близок к 1. Слишком малый коэффициент заполнения означает множество пустых списков и неэффек­тивное использование памяти; слишком большой — более длинные списки и более продолжительный поиск. Но если коэффициент заполнения близок к 1, мы полу­чаем наиболее эффективную схему, которая позволяет находить заданный ключ путем одного или двух сравнений в среднем. Конечно, в дополнение к сравнени­ям мы должны затратить некоторое время на вычисление хеш-функции, но это операция с постоянным временем выполнения, не зависящим от п и т. Заме­тим, что мы получаем такую замечательную эффективность не только благодаря хитроумности самого метода, но и ценой излишнего потребления памяти.

Две другие словарные операции — вставка и удаление — практически идентич­ны поиску. Вставка обычно делается в конец списка (однако в упражнении 7.3.6 вы найдете возможные модификации этого правила). Удаление выполняется путем поиска удаляемого ключа с последующим удалением его из связанного списка. Следовательно, эффективность этих операций равна эффективности поиска, и все они в среднем случае имеют эффективность 0(1), если количество ключей п примерно равно размеру хеш-таблицы т.

Закрытое хеширование (открытая адресация)

В случае закрытого хеширования все ключи хранятся в хеш-таблице, без ис­пользования связанных списков (само собой, это приводит к требованию, чтобы размер хеш-таблицы т был не меньше количества ключей п). Для разрешения коллизий могут применяться разные стратегии. Простейшая из них — линейное исследование (linear probing), когда в случае коллизии ячейки проверяются одна за другой. Если ячейка пуста, новый ключ вносится в нее; если заполнена — прове­ряется ячейка, следующая за ней. Если при проверке достигается конец таблицы, поиск переходит к первой ячейке таблицы, которая рассматривается как цикли­ческий массив. Этот метод проиллюстрирован на рис. 7.6 с применением тех же слов, которые мы использовали для иллюстрации хеширования с раздельными цепочками (здесь мы используем ту же хеш-функцию, что и ранее).

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ключи | А | FOOL | AND | HIS | MONEY | ARE | SOON | PARTED |
| Хеш-адреса | 1 | 9 | 6 | 10 | 7 | 11 | 11 | 12 |
| 0 1 2 | 3 | 4 5 | 6 | 7 | 8 | 9 | 10 11 12 | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A |  |  |  |  |  |  |  |  |  |  |  |
|  | A |  |  |  |  |  |  |  | FOOL |  |  |  |
|  | A |  |  |  |  | AND |  |  | FOOL |  |  |  |
|  | A |  |  |  |  | AND |  |  | FOOL | HIS |  |  |
|  | A |  |  |  |  | AND | MONEY |  | FOOL | HIS |  |  |
|  | A |  |  |  |  | AND | MONEY |  | FOOL | HIS | ARE |  |
|  | A |  |  |  |  | AND | MONEY |  | FOOL | HIS | ARE | SOON |
| PARTED | A |  |  |  |  | AND | MONEY |  | FOOL | HIS | ARE | SOON |

Рис. 7.6. Пример хеш-таблицы, построенной с использованием линейного иссле­дования

Для поиска заданного ключа К мы начинаем с вычисления h (К), где h — хеш- функция, использующаяся при построении таблицы.. Если ячейка h {К) пуста, поиск неудачен. Если ячейка не пуста, мы сравниваем К с ключом, хранящимся в ячейке. Если они равны, то искомый ключ найден; если нет — то мы перехо­дим к следующей ячейке и повторяем описанные действия до тех пор, пока не встретим искомый ключ (успешный поиск) или пустую ячейку (неудачный поиск). Например, если хмы ищем слово LIT в таблице на рис. 7.6, то получим h {LIT) = = (12 + 9 + 20) mod 13 = 2 и, поскольку ячейка 2 пуста, тут же прекратим поиск. При поиске слова KID мы находим h {KID) = (И + 9 + 4) mod 13 = 11, так что сравниваем KID с ключами ARE, SOON, PARTED, прежде чем сможем объявить поиск неудачным.

В то время как операции поиска и вставки в такой версии хеширования весьма просты, этого нельзя сказать об удалении. Например, удалив ключ ARE из по­следнего состояния таблицы на рис. 7.6, мы больше не сможем обнаружить в ней слово SOON. Судите сами: h{SOON) = 11, и алгоритм поиска, обнаружив пу­стую ячейку с этим индексом, сообщит об отсутствии такого слова в таблице. Простейшим решением проблемы является “отложенное удаление”, т.е. ранее за­пятая ячейка помечается специальным образом, чтобы можно было отличить ее от ячеек, которые никогда не были заняты.

Математический анализ линейного исследования — существенно более слож­ная задача, чем анализ хеширования с раздельными цепочками.[[52]](#footnote-52) Упрощенная версия полученных при таком анализе результатов гласит, что среднее количество обращений к хеш-таблице с коэффициентом заполнения а в случае успешного и неудачного поиска равно соответственно

5кК1+г^)ис,4(1+^) (7-[[53]](#footnote-53)»

(точность этого приближения растет с ростом размера хеш-таблицы). Получаю­щиеся числа на удивление малы даже для плотно заполненных таблиц: так, для достаточно больших значений а получаются следующие величины:

|  |  |  |
| --- | --- | --- |
| а | К1 + ^) | 2 (Х 1 (1-а)0 |
| 50% | 1.5 | 2.5 |
| 75% | 2.5 | 8.5 |
| 90% | 5.5 | 50.5 |

По мере заполнения хеш-таблицы производительность линейного исследо­вания ухудшается из-за эффекта кластеризации. Кластером (cluster) в линейном исследовании называется последовательность соседних заполненных ячеек (с воз­можным переходом из последней ячейки таблицы в первую). Например, в окон­чательном состоянии таблицы на рис. 7.6 имеются два кластера. В хешировании кластеры представляют собой отрицательное явление, поскольку снижают эф­фективность словарных операций. Заметим также, что с ростом кластеров растет вероятность того, что новый элемент будет добавлен к кластеру. Кроме того, рас­тет и вероятность слияния кластеров при вставке нового ключа, что еще больше увеличивает кластеризацию.

Для снижения эффекта кластеризации был предложен ряд других стратегий разрешения коллизий. Одна из наиболее важных среди них — двойное хеширо­вание (double hashing). В этой схеме для определения фиксированного значения шага последовательности исследований при коллизии в ячейке I = h (К) исполь­зуется другая хеш-функция s (К):

(I + s (К)) mod 7п, (Z + 2s (К)) mod m, (7.6)

Для каждой ячейки таблицы, исследуемой при помощи такой последователь­ности (7.6), величина шага s (К) и размер таблицы га должны быть взаимно про­стыми величинами, т.е. не должны иметь общих делителей, отличных от 1 (это условие выполняется автоматически, если га — простое число (и s (К) ф га)). Вот некоторые функции, рекомендуемые в литературе: s (к) = га — 2 — к mod (га — 2), 5 {к) = 8 — (к mod 8) — для небольших таблиц и s (к) = к mod 97 + 1 — для больших (см. [102, 103]). Математический анализ двойного хеширования очень сложен. Некоторые частичные результаты и значительный практический опыт свидетельствуют о том, что при выборе хороших хеш-функций — как первичной, так и вторичной — двойное хеширование превосходит метод линейных исследова­ний. Однако производительность этого метода также падает по мере приближения таблицы к полностью заполненному состоянию. Единственным решением в такой ситуации является новое хеширование: текущая таблица сканируется, и все ключи из нее хешируются в новую таблицу большего размера.

Со времени открытия хеширования в 1950-х годах исследователями из IBM оно нашло множество важных применений. В частности, хеширование стало стан­дартным методом для хранения таблиц символов — таблицы символов в компью­терной программе, генерируемой в процессе компиляции последней. С неболь­шими модификациями метод хеширования, как было доказано, пригоден для хра­нения очень больших словарей на диске; эта версия хеширования называется расширяемым хешированием (extensible hashing). Поскольку обращение к диску существенно дороже исследований в оперативной памяти, предпочтительно вы­полнение существенно большего количества исследований, чем обращений к дис­ку. Соответственно, положения, вычисляемые при помощи хеш-функции в рас­ширяемом хешировании, указывают адрес на диске, где находится блок (bucket), в котором может храниться до Ь ключей. После того как определен блок для искомого ключа, все ключи блока считываются в оперативную память и среди них выполняется поиск искомого ключа. В следующем разделе мы рассмотрим В-деревья, главный альтернативный способ хранения больших словарей.

Упражнения 7.3

1. Для входных данных 30, 20, 56, 75, 31, 19 и хеш-функции h(K) = = К mod 11

а) постройте закрытую хеш-таблицу;

б) найдите наибольшее количество сравнений ключей при успешном поиске в полученной таблице; найдите среднее количество сравне­ний ключей при успешном поиске в полученной таблице.

1. Чем плоха идея хеш-функции, зависящей только от одной (например, первой) буквы слова естественного языка?

image165

1. Найдите вероятность того, что все п ключей будут хешированы в од­ну ячейку хеш-таблицы размером т, если хеш-функция распределяет ключи равномерно по всем ячейкам таблицы.
2. Парадокс дней рождений. Парадокс дней рождений заключается в определении количества людей, которых надо собрать в одной ком­нате, чтобы шансы на то, что среди них найдется хотя бы одна пара родившихся в один день года (имеется в виду день и месяц), превысили шансы того, что все присутствующие родились в разные дни. Найдите (достаточно неожиданный) ответ на этот вопрос. Какое важное след­ствие из полученного ответа вытекает для хеширования?
3. Ответьте на следующие вопросы для версии хеширования с раздель­ными цепочками.

а) Куда вы предпочтете вставлять ключи, если известно, что все ключи в словаре различны? Какие словарные операции могут выиграть от такой модификации?

б) Мы можем поддерживать ключи в каждом связанном списке в от­сортированном порядке. Какие словарные операции могут выиграть от такой модификации? Как можно воспользоваться этим для сор­тировки всех ключей, хранящихся в таблице?

1. Поясните, как можно применить хеширование для проверки того, что все ключи в списке различны? Какова временная эффективность такого применения?
2. Заполните следующую таблицу классами эффективности в среднем случае для пяти реализаций абстрактного типа данных словаря:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Неупорядочен­ный массив | Упорядочен­ный массив | Бинарное дерево поиска | Раздельные  цепочки | Линейное  исследование |
| Поиск |  |  |  |  |  |
| Вставка |  |  |  |  |  |
| Удаление |  |  |  |  |  |

1. Мы рассматривали хеширование в контексте методов, основанных на пространственно-временном компромиссе. Однако оно использует пре­имущества еще одной общей стратегии. Какой именно?
2. Напишите компьютерную программу, которая использует хеширование для следующей задачи. Имеется текст на естественном языке. Требует­ся сгенерировать список различных слов в тексте (с количеством вхож­дений их в этот текст). Внесите в программу необходимые счетчики для сравнения эмпирической эффективности хеширования с соответ­ствующими теоретическими результатами.
3. В-деревья

Идея использования дополнительной памяти для ускорения доступа к данным особенно важна, если рассматриваемый набор данных содержит очень большое количество записей, которые требуется хранить на диске. Основной метод органи­зации таких наборов данных — индексы, которые предоставляют определенную информацию о размещении записей с указанными значениями ключей. Для на­боров данных, состоящих из структурированных записей (в противоположность таким “неструктурированным” данным, как текст, изображения, звук и видео), наиболее важной организацией индексов являются В-деревья (B-tree), разработан­ные Р. Бойером (R. Bayer) и Э. Мак-Грейтом (Е. McGreight) [12]. Они расширяют идею 2-3-деревьев (см. раздел 6.3), разрешая иметь в одном узле дерева поиска много ключей.

В В-дереве все записи данных (или ключи) хранятся в листьях, в возраста­ющем порядке ключей, а родительские узлы используются для индексирования. В частности, каждый родительский узел содержит п — 1 упорядоченных ключей К\ < • • • < Кп-1 (которые для простоты полагаем различными). Ключи разделе­ны п указателями на дочерние узлы, так что все ключи в поддереве То меньше К\9 все ключи в поддереве Т\ — не меньше К\ и меньше , причем К\ равен наи­меньшему ключу в Ti, и так далее до последнего поддерева Tn\_i, ключи которого не меньше Кп-ь причем Кп-\ равен наименьшему ключу в Тп\_\ (рис. 7.7)[[54]](#footnote-54).

Кроме того, В-дерево порядка тп ^ 2 должно удовлетворять следующим струк­турным свойствам.

* Корень либо является листом, либо имеет от 2 до га потомков.
* Каждый узел, за исключением корня и листьев, имеет от [га/2] до га потомков (и, следовательно, от [га/2] — 1 до га — 1 ключей).

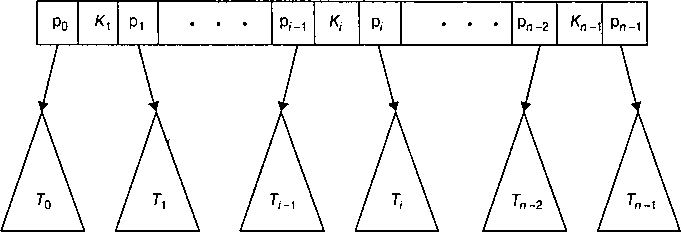


Рис. 7.7. Родительский узел В-дерева

■ Дерево (идеально) сбалансировано, т.е. все листья находятся на одном и том же уровне.

Пример В-дерева порядка 4 приведен на рис. 7.8.

Поиск в В-дереве похож на поиск в бинарном дереве поиска, и еще больше — на поиск в 2-3-дереве. Начиная с корня, мы следуем по цепочке указателей к листу, который может содержать искомый ключ. Затем мы ищем ключ среди ключей этого листа. Заметим, что, поскольку ключи хранятся в отсортированном порядке как в родительских узлах, так и в листьях, мы можем воспользоваться бинарным поиском, если количество ключей в узле достаточно велико, чтобы сделать такой поиск оправданным.

Однако при рассмотрении типичного приложения этой структуры данных мы должны в первую очередь рассматривать не количество сравнений ключей. При использовании для хранения больших объемов данных дискового файла узлы В-дерева обычно соответствуют дисковым страницам. Поскольку обычно время, необходимое для обращения к дисковой странице, на несколько порядков пре­вышает время сравнения двух ключей в быстрой основной памяти, основным показателем эффективности для этой и подобных структур данных является ко­личество обращений к диску.

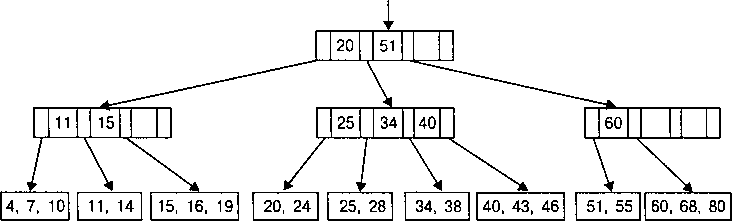


Рис. 7.8. Пример В-дерева порядка 4

К скольким узлам В-дерева требуется доступ при поиске записи с данным значением ключа? Очевидно, что эта величина равна высоте дерева плюс еди­ница. Для оценки высоты найдем, чему равно наименьшее количество ключей в В-дереве порядка т высотой h. Корень дерева содержит как минимум один ключ. На уровне 1 должно быть как минимум два узла с как минимум [m/2] — 1 ключами в каждом, т.е. общее количество ключей на первом уровне не менее 2([m/2] — 1). На втором уровне имеется не менее 2 [т/2] узлов (узлы, дочер­ние по отношению к узлам первого уровня) с как минимум [m/2] — 1 ключами в каждом, так что общее минимальное количество ключей на втором уровне со­ставляет 2 [m/2] ([т/2] — 1). В общем случае узлы на г-ом уровне (1 ^ г ^ h— 1) содержат как минимум 2 [т/2]г\_1 ([т/2] — 1) ключей. И, наконец, уровень h — уровень листьев — содержит как минимум 2 \m/2]h~l узлов с как минимум одним ключом в каждом. Таким образом, для любого В-дерева порядка теп узлами и высотой h > 0 справедливо следующее неравенство:

/i-i

п ^ 1 + 2 [т/2]1-1 ([т/2] — 1) + 2 \m/2]h~l.

**i—1**

После ряда стандартных упрощений (см. упражнение 7.4.2) это неравенство сво­дится к

п ^ 4 ***\m/2\h~1 —*** 1,

которое, в свою очередь, дает следующую верхнюю границу высоты h В-дерева порядка men узлами:

h ^ 1о§Гт/21 —7—

+ 1. (7.7)

Из этого неравенства непосредственно следует вывод, что поиск в В-дереве яв­ляется операцией, принадлежащей классу эффективности O(logn). Однако нас интересует не только класс эффективности, но и конкретные числа обращений к диску, вычисленные по данной формуле. В приведенной далее таблице содер­жатся значения правой части формулы для файла со 100 миллионами записей и типичных значений порядка В-дерева т:

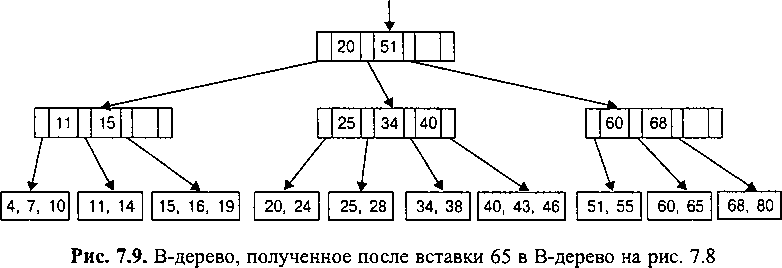
Порядок т 50 100 250

Верхняя граница h 6 5 4

Учтите, что в таблице приведены верхние оценки для количества обращений к диску. В реальных приложениях это число редко превосходит 3, к тому же корень В-дерева, а иногда и узлы первого уровня зачастую хранятся в оперативной памяти для минимизации количества обращений к диску.

Операции вставки и удаления оказываются не столь простыми, как операция вставки, но тем не менее они также выполняются за время О (logn). Здесь мы набросаем только алгоритм вставки; описание алгоритма удаления можно найти в соответствующей литературе (например, в [5] или [32]).

Наиболее простой алгоритм вставки новой записи в В-дерево очень похож на алгоритм вставки в 2-3-дерево, описанный в разделе 6.3. Сначала мы используем процедуру поиска hobqto ключа К, для того чтобы найти лист для его вставки. Если в нем есть место для нового ключа, мы вставляем его (в соответствующую позицию, так, чтобы ключи в листе оставались отсортированными), и на этом ра­бота завершается. Если же в листе нет места для новой записи, лист разбивается на два, при этом вторая половина записей переходит в новый узел. После этого наименьший узел К' нового узла и указатель на него вставляются в родительский узел старого листа (непосредственно после ключа и указателя на старый лист). Эта рекурсивная процедура может распространяться вплоть до корня дерева. Ес­ли корень также заполнен, старый корень разбивается на две части, и создается новый корень, у которого две части старого корня становятся дочерними узлами. В качестве примера на рис. 7.9 показан результат вставки ключа 65 в В-дерево, изображенное на рис. 7.8 (при ограничении, что листья В-дерева не могут содер­жать более трех элементов).



Вы должны знать, что существуют и другие алгоритмы для реализации вставки в В-дерево. Например, мы можем избежать возможного рекурсивного разделения заполненных узлов, если будем разделять их при поиске листа для новой запи­си. Еще одна возможность избежать рекурсивного разделения заполненных узлов заключается в перемещении ключа в “братский” узел. Например, вставка 65 в В- дерево на рис. 7.8 может быть выполнена путем перемещения 60, наименьшего ключа в листе, в “братский” лист с 51 и 55, и замещением значения ключа в ро­дительском узле на 65 (новое наименьшее значение ключа у правого потомка). Такое видоизменение позволяет сэкономить память ценой усложнения алгоритма.

В-дерево не обязательно должно быть связано с индексированием большого файла и может рассматриваться как один из вариантов дерева поиска. Как и в слу­чае других типов деревьев поиска — таких как бинарные деревья, AVL-деревья

или 2-3-деревья, — В-деревья могут строиться путем последовательной вставки записей в изначально пустое дерево (пустое дерево также рассматривается как В- дерево). Когда все ключи расположены в листьях, а более высокие уровни органи­зованы как В-дерево, содержащее индекс, вся такая структура обычно называется В+-деревом.

Упражнения 7.4

1. а) Напишите программу, реализующую алгоритм вставки ключа в В- дерево.

б) Напишите программу для визуализации алгоритма вставки ключа в В-дерево.

Резюме

* Пространственно-временной компромисс в проектировании алгорит­мов хорошо известен как теоретикам, так и практикам в этой области. В качестве метода проектирования алгоритма использование лишней памяти для экономии времени встречается гораздо чаще, чем лишнее время для экономии памяти.
* Улучшение входных данных является одним из двух основных вариан­тов пространственно-временного компромисса в проектировании алго­ритмов. Его идея состоит в предварительной обработке входных дан­ных — целиком или частично — и сохранении полученной дополни­тельной информации для ускорения решения поставленной задачи. На этом методе основаны, в частности, сортировка подсчетом распределе­ния и несколько важных алгоритмов поиска подстрок.
* Сортировка подсчетом распределения представляет собой специаль­ный метод сортировки списков элементов из небольшого множества возможных значений.
* Алгоритм Хорспула для поиска подстрок можно рассматривать как упрощенную версию алгоритма Бойера-Мура. Оба эти алгоритма ос­нованы на идее улучшения входных данных и сравнении символов об­разца справа налево. Оба алгоритма используют одну и ту же таблицу сдвигов несовпадающих символов, а алгоритм Бойера-Мура использует еще одну таблицу — таблицу сдвигов совпадающих суффиксов.
* Предварительная структуризация — второй тип методики с примене­нием пространственно-временного компромисса — использует допол­нительную память для более быстрого и/или более гибкого доступа к данным. Важными примерами предварительной структуризации яв­ляются хеширование и В-деревья.
* Хеширование — очень эффективный способ реализации словарей. Он основан на идее отображения ключей в одномерную таблицу. Ограни­чения, налагаемые на размер такой таблицы, вынуждают применять ме­ханизм разрешения коллизий. Двумя основными вариантами хеширова­ния являются открытое хеширование, или хеширование с раздельными

цепочками (когда ключи хранятся в связанных списках вне хеш-табли­цы), и закрытое хеширование, или открытая адресация (когда ключи хранятся внутри хеш-таблицы). Оба варианта в среднем случае обес­печивают эффективность поиска, вставки и удаления, равную 0 (1).

■ В-дерево представляет собой сбалансированное дерево поиска, кото­рое обобщает идею 2-3-дерева, разрешая наличие нескольких ключей в одном узле. Основное применение В-деревьев — хранение индексной информации о данных, сохраненных на диске. Путем соответствую­щего выбора порядка В-дерева можно реализовать операции поиска, вставки и удаления при помощи всего лишь нескольких обращений к диску даже для очень больших файлов.

image169

**Глава**

Динамическое программирование

Об идее, как и о привидении,... следует немного поговорить, прежде чем она явится.

— Чарльз Диккенс (Charles Dickens) (1812-1870)

Д

инамическое программирование представляет собой метод проектирования алгоритмов с весьма интересной историей. Оно было открыто известным американским математиком Ричардом Веллманом (Richard Bellman) в 1950-х го­дах как общий метод оптимизации многостадийных процессов принятия решения. Таким образом, слово “программирование” в названии метода означает “плани­рование” и не имеет никакого отношения к компьютерному программированию. После того как было доказано, что этот метод представляет собой важный ин­струмент прикладной математики, динамическое программирование стало рас­сматриваться, по крайней мере в среде кибернетиков, как метод проектирования алгоритмов, который не ограничивается только узким кругом задач оптимизации. Именно с такой точки зрения мы и будем рассматривать здесь указанный метод.

Динамическое программирование представляет собой метод решения задач с перекрывающимися подзадачами. Обычно такие подзадачи возникают из ре­куррентных соотношений, связывающих решение данной задачи с решениями меньших подзадач того же вида. Вместо того чтобы решать перекрывающиеся подзадачи снова и снова, динамическое программирование предлагает решить каждую из меньших подзадач только один раз, записав при этом результат в таб­лицу, из которой затем можно будет получить решение исходной задачи.

Этот метод можно проиллюстрировать на примере чисел Фибоначчи, рас­сматривавшихся в разделе 2.5 (даже если вы не читали этот раздел, вы все равно можете следить за ходом рассуждений; однако это интересный вопрос, так что, почувствовав желание прочесть указанный раздел, — не противьтесь ему). Числа Фибоначчи представляют собой элементы последовательности

0,1,1,2,3,5,8,13,21,34,... ,

которую можно определить с помощью простого рекуррентного соотношения

F (п) = F (п — 1) + F (п — 2) при п > 2 (8.1)

и двух начальных условий

F (0) = 0, F (1) = 1. (8.2)

Если мы попытаемся использовать для вычисления n-го числа Фибоначчи F (п) рекуррентное соотношение (8.1) непосредственно, то придется много раз вычислять одни и те же значения данной функции (см. пример на рис. 2.6). Заме­тим, что задача вычисления F (п) выражается через перекрывающиеся подзадачи меньшего размера — вычисления F (п — 1) и F (п — 2). Таким образом, мы мо­жем просто заполнить элементы одномерного массива п + 1 последовательными значениями F (п), начиная с начальных значений (8.2) и используя рекуррентное соотношение (8.1). Очевидно, что последний элемент данного массива будет со­держать значение F (п). Псевдокод такого очень простого алгоритма можно найти в разделе 2.5.

Заметим, что можно обойтись и без использования дополнительного масси­ва, ограничившись запоминанием только двух последних элементов последова­тельности Фибоначчи (см. упражнение 2.5.6). Такая ситуация не является чем- то необычным, мы встретимся с ней еще в нескольких примерах в данной гла­ве. Хотя непосредственное применение динамического программирования можно рассматривать как частный случай экономии времени за счет памяти, зачастую алгоритмы динамического программирования можно усовершенствовать, избежав использования излишней памяти.

Некоторые алгоритмы вычисляют n-е число Фибоначчи без вычисления всех предшествующих элементов последовательности (см. раздел 2.5). Однако для ал­горитмов, основанных на классическом подходе восходящего динамического про­граммирования, типично решение всех меньших подзадач данной задачи. Один из вариантов динамического программирования состоит в том, чтобы избежать решения ненужных подзадач. Этот метод, проиллюстрированный в разделе 8.4, использует так называемые функции с запоминанием и может рассматриваться как нисходящая разновидность динамического программирования. Однако использу­ем ли мы классическую восходящую разновидность динамического программиро­вания или нисходящее динамическое программирование с применением функций с запоминанием, основной этап в разработке таких алгоритмов остается один и тот же, а именно: вывод рекурсивного соотношения, связывающего решение экземпляра задачи с решениями меньших (перекрывающихся) подэкземпляров. Непосредственная применимость уравнения (8.1) для вычисления n-го числа Фи­боначчи является одним из немногих исключений из этого правила.

В разделах и упражнениях этой главы имеется несколько стандартных приме­ров алгоритмов динамического программирования (некоторые из них были раз­работаны до открытия динамического программирования или независимо от него и только позже стали рассматриваться как примеры применения этого метода). Множество других применений динамического программирования лежат в диапа­зоне от оптимального разбиения текста на строки (например, [8]) до оптимальной триангуляции многоугольника (например, [111]) и решения различных сложных инженерных задач (например, [14, 18]).

1. Вычисление биномиальных коэффициентов

Вычисление биномиальных коэффициентов — стандартный пример примене­ния динамического программирования к задаче, не являющейся задачей опти­мизации. Вспомним из курса элементарной комбинаторики, что биномиальным коэффициентом (binomial coefficient), обозначаемым C(n,fc), С„ или (£), назы­вается количество комбинаций (подмножеств) из к элементов у п-элементного множества 0 ^ к ^ п). Название “биномиальные коэффициенты” происходит от участия этих чисел в формуле бинома:

(а + Ь)п = С°ап + • • • + Скап~кЪк + • • • + С%Ъп.

Из множества свойств биномиальных коэффициентов мы остановимся только на двух:

***СП = Ckz\ +ск\_х*** при ***п > к >*** 0 (8.3)

**И**

С° = СЦ = 1. (8.4)

Природа рекуррентного соотношения (8.3), выражающего задачу вычисления

посредством решения меньших перекрывающихся задач вычисления C^Z\ и C^-i подводит нас к решению задачи с использованием метода динамического программирования. Для этого запишем значения биномиальных коэффициентов в таблицу с n+1 строками и fc+1 столбцами, пронумерованными, соответственно, от 0 до п и от 0 до к (рис. 8.1).

Для вычисления С% мы заполняем таблицу на рис. 8.1 строку за строкой, начиная со строки 0 и заканчивая строкой п. Каждая строка г (0 < г ^ п) запол­няется слева направо, начиная с 1, так как С° = 1. На главной диагонали таблицы в строках от 0 по fc-ую также находятся 1, поскольку С\ = 1 (0 ^ г ^ к). Осталь­ные элементы таблицы вычисляются по формуле (8.3) путем сложения элементов предыдущей строки — из того же и предшествующего столбца (если в этой табли­це вы узнали треугольник Паскаля — очаровательную математическую структуру,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 2 | • к-1 | к |
| 0 | 1 |  |  |  |
| 1 | 1 | 1 |  |  |
| 2 | 1 | 2 1 |  |  |
| к | 1 |  |  | 1 |
| п — 1 | 1 |  | s~ik—1 °п-1 | cLi |
| п | 1 |  |  | Скп |

Рис. 8.1. Таблица для вычисления бино­миальных коэффициентов Ск при помо­щи алгоритма динамического программи­рования

обычно рассматриваемую при изучении сочетаний, — то вы правы: перед ва­ми действительно треугольник Паскаля). Вот псевдокод, реализующий данный алгоритм.

**Алгоритм Binomial** **(п, к)**

II Вычисление С(п, к) при помощи алгоритма динамического // программирования

// Входные данные: Пара неотрицательных целых чисел к ^ О

// Выходные данные: Значение С(п, к) for г <— 0 to п do

for j <— 0 to min (г, к) do if 7 = 0 or j = г

***C[i,j] \*-*** **1**

else

C[i,j] «- C[i - 1, j - 1] + C[i - 1, j] return C[n,k]

Какова временная эффективность данного алгоритма? Очевидно, что базовой операцией алгоритма является сложение, так что обозначим через А (п, к) общее количество сложений, выполняемых при вычислении С^. Заметим, что вычисле­ние каждого элемента по формуле (8.3) требует только одного сложения. Кроме того, поскольку первые к + 1 строк таблицы образуют треугольник , а осталь­ные п —к строк — прямоугольник, можно разбить сумму, выражающую значение

А (п, к), на две части:

*к* г—1 *п к к п*

A(n>fc) = EE1+ 2 Хл = Х^\_1) + 2 =

**г=1 j=l г=/с+1 j=l г=1 г=/с+1**

= (k ~ ^ fc + к (n - fc) е 0 (пк).

В упражнениях от вас потребуется сравнить полученную эффективность с эф­фективностью некоторых других алгоритмов для решения этой задачи. Еще одно из упражнений состоит в анализе того, нельзя ли сэкономить дополнительную память, используемую данным алгоритмом динамического программирования.

Упражнения 8.1

1. а) Что общего имеет динамическое программирование с методом де­

композиции?

б) В чем главное отличие между этими двумя методами?

1. а) Вычислите С| при помощи алгоритма динамического программи­

рования.

б) Можно ли вычислить С\*;, заполняя таблицу из алгоритма динами­ческого программирования по столбцам, а не по строкам?

1. Докажите следующее утверждение, сделанное в тексте раздела при определении временной эффективности алгоритма динамического про­граммирования для вычисления С\*;:

------- + fc(n-fc)G0 (пк).

Z

1. а) Сколько дополнительной памяти требуется алгоритму динамическо­

го программирования Binomial для вычисления С^?

б) Как можно повысить эффективность использования дополнитель­ной памяти этим алгоритмом (попытайтесь снизить количество до­полнительной памяти настолько, насколько сможете).

1. а) Найдите порядок роста следующих функций:

1) С\ 2) С% 3) Сп^2 для четных п

б) Какое основное следствие для вычисления вытекает из ответа на часть а) данного упражнения?

1. Найдите точное количество сложений, выполняемое следующим ре­курсивным алгоритмом, основанным на непосредственном примене­нии формул (8.3) и (8.4):

Алгоритм BinomCoeff(n, к) if к = 0 or к = п return 1 else

return ***BinomCoeff{n*** — 1, ***к*** — 1) + ***BinomCoeff[n*** — 1, ***к)***

1. Какой из следующих алгоритмов для вычисления биномиальных коэф­фициентов наиболее эффективен?

а) Использование формулы

***к п\***

***п к\(п — к)\'***

б) Использование формулы

к \_ **га (га - l)---(n-fc + l)** **к\**

в) Рекурсивное применение формулы

при га > fc > О, С% = С% = 1.

г) Применение алгоритма динамического программирования.

1. Докажите, что

С\* = С£~к при п^к^О, и поясните, как можно воспользоваться этой формулой при вычислении

image170

1. Задача о чемпионате. Рассмотрим две команды, А и В, которые про­водят серию игр до тех пор, пока одна из них не одержит п побед. Предположим, что вероятность победы команды А одна и та же в каж­дой игре и равна р, а вероятность проигрыша — q = 1 — р (следова­тельно, ничьих в игре не бывает). Пусть Р (i,j) — вероятность победы А в серии игр, если А для победы требуется выиграть еще г игр, а В для победы требуется выиграть еще j игр.

а) Напишите рекуррентное соотношение для которое можно

использовать в алгоритме динамического программирования.

б) Найдите вероятность того, что А победит в серии из 7 игр, если вероятность победы в одной игре равна 0.4.

в) Напишите псевдокод алгоритма динамического программирования для решения этой задачи и определите его временную и простран­ственную эффективность.

1. Алгоритмы Воршалла и Флойда

В этом разделе мы рассмотрим два хорошо известных алгоритма: алгоритм Воршалла для вычисления транзитивного замыкания ориентированного графа и алгоритм Флойда для решения задачи поиска кратчайших путей между все­ми парами вершин. Эти алгоритмы основаны, по сути, на одной и той же идее, которую можно рассматривать как применение метода динамического програм­мирования.

Алгоритм Воршалла

Вспомним, что матрица смежности А = {а^ } ориентированного графа пред- ставляет собой булеву матрицу, которая содержит на пересечении г-ой строки и j-ro столбца 1 тогда и только тогда, когда имеется ориентированное ребро от г-ой вершины к j-ой. Нас может также интересовать матрица, содержащая ин­формацию о существовании ориентированного пути произвольной длины между вершинами графа.

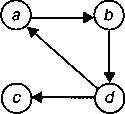
Определение 1. Транзитивное замыкание (transitive closure) ориентированного графа с п вершинами можно определить как булеву матрицу Т = {tij} размером пхп, в которой элемент на пересечении г-ой строки (1 ^ г ^ п) и j-ro столбца (1 ^ j ^ п) равен 1, если существует нетривиальный ориентированный путь (т.е. ориентированный путь положительной длины) из вершины г в вершину j; в противном случае значение равно 0. ■

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | a | Ь | с | d |  | a |
| a | ’о | 1 | 0 | о" | а | ’l |
| Ь | 0 | 0 | 0 | 1 | b | 1 |
|  |  |  |  |  | Т = |  |
| с | 0 | 0 | 0 | 0 | с | 0 |
| d | 1 | 0 | 1 | 0 | d | 1 |

а) б) в)

Рис. 8.2. а) Ориентированный граф, его б) матрица смежности и в) транзитивное замыкание

А =



На рис. 8.2 приведен пример ориентированного графа, его матрицы смежности и транзитивного замыкания.

bed 11 1 "

1 1 1 ООО 11 1 \_

Транзитивное замыкание ориентированного графа можно построить с помо­щью поиска в глубину или в ширину. Выполнение одного из поисков, начиная от г-ой вершины, дает информацию о вершинах, достижимых из нее, а следователь­но, — столбцы, на пересечении которых с г-ой строкой матрицы транзитивного замыкания содержатся единицы. Таким образом, полную матрицу транзитивно­го замыкания можно получить, выполняя обход графа для каждой его вершины в качестве начальной точки.

Поскольку такой метод неоднократно обходит один и тот же ориентированный граф, можно надеяться на существование более эффективного алгоритма. Такой алгоритм существует и называется алгоритмом Воршалла (Warshall’s algorithm) по имени его автора, С. Воршалла (S. Warshall) [119]. Алгоритм Воршалла строит транзитивное замыкание ориентированного графа с п вершинами как последова­тельность булевых матриц размером пхп:

,..., #(fc\_1), R(k),..., Д(п). (8.5)

Каждая из этих матриц предоставляет определенную информацию о направлен-

*(к)*

ных путях в ориентированном графе. В частности, элемент r\-J на пересечении г-ой строки и j-го столбца матрицы RW (А: = 0,1,..., гг) равен 1 тогда и только тогда, когда существует ориентированный путь (положительной длины) из г-ой в j-ую вершину такой, что все промежуточные вершины, если таковые имеются, имеют номера не выше к. Таким образом, пути в Rне могут иметь промежуточ­ных вершин, а значит, представляет собой не что иное, как матрицу смеж­ности ориентированного графа. Матрица R1'1'1 содержит информацию о путях, в которых в качестве промежуточной вершины может выступать вершина с номе­ром 1, а значит, если можно так выразиться, должна содержать больше единиц, чем В общем случае каждая следующая матрица последовательности (8.5) допускает в качестве промежуточных вершин на одну больше, чем предыдущая, а следовательно, может (но не обязана) содержать большее количество единиц. Последняя матрица последовательности R^ в качестве промежуточных вершин может использовать все п вершин ориентированного графа, а значит, представляет собой не что иное, как транзитивное замыкание ориентированного графа.

Основная особенность алгоритма заключается в том, что можно вычислить все элементы каждой матрицы R№ на основании ее непосредственного предше­ственника R^k~^ в ряду (8.5). Пусть rjk\ элемент на пересечении г-ой строки и j-го столбца матрицы R(k\ равен 1. Это означает, что существует путь из г-ой вершины Vi в j-ую вершину Vj, в котором все промежуточные вершины имеют номера не выше к:

Vi Список вершин с номерами, не превышающими к Vj. (8.6)

Что касается такого пути, то возможны две ситуации. Первая: список промежу­точных вершин не содержит к-ую. Тогда этот путь из vt в Vj состоит из вершин, имеющих номера не выше к — 1, следовательно, элемент также равен 1.

Вторая: путь (8.6) содержит среди прочих промежуточных вершин к-ую вершину vi\*. Без потери общности можно считать, что Vk встречается в списке только один

раз (если это не так, можно создать новый путь от V{ до Vj, в котором будут удалены все вершины между первым и последним вхождениями г^). С учетом сказанного путь (8.6) можно переписать следующим образом:

^ вершины с номерами ^ к — 1, и\*., вершины с номерами ^ к — 1 ^ Vj.

Первая часть этого представления означает, что существует путь от vi до Vk такой, что все промежуточные вершины в нем имеют номера не больше, чем к — 1 (следовательно, — 1), а вторая часть означает, что существует путь от Vk до

Vj такой, что все промежуточные вершины в нем имеют номера не больше, чем к — 1 (следовательно, = 1).

В результате мы только что доказали, что если = 1, то либо = 1,

***(к—1) (к—1)***

либо и г^к — 1, и rjy = 1. Легко увидеть, что истинно утверждение, обратное этому. Таким образом, мы получаем формулу для генерации элементов матрицы

***R(k)***

из элементов матрицы

rij] = rij~l) ог г2~1} and rS\_1)- <8-7)

Формула (8.7) представляет собой “сердце” алгоритма Воршалла. Из нее вытекает следующее правило генерации элементов матрицы из элементов матрицы которое, в частности, удобно при использовании алгоритма Воршалла вручную.

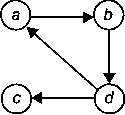
* Если элемент гц равен 1 в R^k~l\ то он остается равен 1 и в
* Если элемент T\j равен 0 в R^k~x\ то он становится равным 1 в д(\*) тогда и только тогда, когда в R(k~^ и элемент в г-ой строке и к-ои столбце, и элемент в к-ой строке и j-ом столбце равны 1 (это правило проиллюстрировано на рис. 8.3).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| У | к | ■ |  | У | к |
| 1 |  |  | R(k)= к | 1 |  |
| Т  0 -> | 1 |  | / | 1 | 1 |

Рис. 8.3. Правило для замены нолей в алгоритме Воршалла

Применение алгоритма Воршалла к ориентированному графу, изображенному на рис. 8.2, показано на рис. 8.4.

Вот как выглядит псевдокод алгоритма Воршалла.



д( 0)=

1 О О

о о о о

О 1

Единицы отражают наличие путей без промежуточных вершин (Я(0) — матрица смежности); строка и столбец в прямоугольниках используются для вычисления Я(1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| а | 0 | 1 | 0 | 0 |
| b | 1 0 | 0 | 0 | 11 |
| с | 0 | 0 | 0 | 0 |
| d | 1 | 1 | 1 | 0 |
|  | a | b | с | d |
| а | 0 | 1 | 0 | 1 |
| b | 0 | 0 | 0 | 1 |
| с | I o | 0 | 0 | 0 I |
| d | 1 | 1 | 1 | 1 |
|  | a | b | с | d |
| а | 0 | 1 | 0 | 1 |
| b | 0 | 0 | 0 | 1 |
| с | 0 | 0 | 0 | 0 |
| d | 11 | 1 | 1 | 1 |

**я(1)=**

Д(2) =

**д(3)=**

Единицы отражают наличие путей с промежуточными вершинами, номера которых не выше 1, т.е. только с вершиной а (новый путь от d к Ь)\ выделенные прямоугольниками строка и столбец используются для получения Я(2).

Единицы отражают наличие путей с промежуточными вершинами, номера которых не выше 2, т.е. с вершинами а и b (два новых пути); выделенные прямоугольниками строка и столбец используются для получения Я(3).

Единицы отражают наличие путей с промежуточными вершинами, номера которых не выше 3, т.е. с вершинами а, b и с (новых путей нет); выделенные прямоугольниками строка и столбец используются для получения Я(4).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | с | d |  |
| a | 1 | 1 | 1 | 1 | Единицы отражают наличие путей с промежуточными |
| b | 1 | 1 | 1 | 1 | вершинами, номера которых не выше 4 |
| с | 0 | 0 | 0 | 0 | т.е. с вершинами а, Ь, с и d (пять новых путей). |
| d | 1 | 1 | 1 | 1 |  |

Я<4>=

Рис. 8.4. Применение алгоритма Воршалла к ориентированному графу. Новые единицы в матрицах выделены полужирным шрифтом

Алгоритм Warshall (A [l..n, 1..п])

// Реализует алгоритм Воршалла для вычисления транзитивного // замыкания

// Входные данные: Матрица смежности А ориентированного

// графа с п вершинами

// Выходные данные: Транзитивное замыкание ориентированного // графа

д(°) <- А for к <— 1 to п do for i <— 1 to n do for j <— 1 to n do

***R(k)[i,j] \*— R^liJ]*** or ***k]*** and ***R^k~^[k, j]***

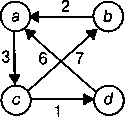
return

По поводу алгоритма Воршалла требуется сделать несколько замечаний. Во- первых, он очень краток. Во-вторых, его временная эффективность составляет 0(п3), так что для разреженных графов, представленных связными списками смежности, алгоритм с использованием обхода графа, упомянутый в начале дан­ного раздела, оказывается асимптотически эффективнее алгоритма Воршалла (по­чему?). Можно ускорить приведенную выше реализацию алгоритма Воршалла, видоизменив его внутренний цикл (см. упражнение 8.2.4). Еще один путь уско­рения алгоритма состоит в рассмотрении строк матрицы как битовых строк и ис­пользовании побитовой операции or, имеющейся на большинстве современных компьютеров.

Что же касается пространственной эффективности алгоритма Воршалла, то ситуация похожа на два примера, рассматривавшиеся ранее в этой главе: вычис­ление чисел Фибоначчи и биномиальных коэффициентов. Хотя мы используем отдельные матрицы для промежуточных результатов алгоритма, на самом деле это не является необходимым (в упражнении 8.2.3 спрашивается, как можно из­бежать такого напрасного расхода памяти). Наконец, позже мы увидим, как идея, лежащая в основе алгоритма Воршалла, может быть применена к более общей задаче поиска длин кратчайших путей во взвешенных графах.

**Алгоритм Флойда поиска кратчайших путей между всеми парами вершин**

Задача поиска кратчайших путей между всеми парами вершин (all-pairs shortest-paths problem) состоит в поиске для данного взвешенного связного графа (ориентированного или неориентированного) расстояний от каждой вершины до всех других вершин. Для записи длин кратчайших путей удобно воспользоваться матрицей D размером пхп, которая называется матрицей расстояний (distance matrix): элемент dij на пересечении г-ой строки и j-ro столбца такой матрицы ука­зывает длину кратчайшего пути от г-ой вершины до j-ой вершины (1 ^ ^ п).



*w =*

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | а | Ь | с | d |  | а | Ь | с | d |
| а | "о | 00 | 3 | 00 | а | "о | 10 | 3 | 4~ |
| b | 2 | 0 | 00 | 00 | ■О  II  О | 2 | 0 | 5 | 6 |
| с | 00 | 7 | 0 | 1 | С | 7 | 7 | 0 | 1 |
| d | 6 | 00 | 00 | 0 | d | 6 | 16 | 9 | 0 |

а) б) в)

Рис. 8.5. а) Ориентированный граф, б) его весовая матрица и в) матрица расстояний

Пример такой матрицы приведен на рис. 8.5.

Мы можем построить матрицу расстояний при помощи алгоритма, очень по­хожего на алгоритм Воршалла. Этот алгоритм называется алгоритмом Флойда (Floyd’s algorithm) по имени его изобретателя Р. Флойда (R. Floyd) [36]. Он при­меним как к ориентированным, так и к неориентированным взвешенным графам, лишь бы они не содержали циклов с отрицательной длиной (естественно, в случае ориентированного графа под путем или циклом мы подразумеваем ориентирован­ный путь или цикл).

Алгоритм Флойда вычисляет матрицу расстояний взвешенного графа с п вер­шинами посредством построения последовательности

£>(0),..., £»(fc\_1), £>(fc),..., D^. (8.8)

Каждая из этих матриц содержит длины кратчайших путей с определенными ограничениями. В частности, элемент на пересечении г-ой строки и j-ro столбца матрицы к = 0,1,..., гг) равен длине кратчайшего пути среди всех путей от г-ой вершины к j-ой, в которых промежуточные вершины (если таковые есть) не могут иметь номера, превышающие к. В частности, последовательность начинается с матрицы D^°\ в которой в путях не может быть промежуточных вершин (т.е. Dпредставляет собой просто весовую матрицу графа). Последняя матрица последовательности, D^n\ содержит длины кратчайших путей среди всех путей, в которых в качестве промежуточных вершин могут быть любые из п вершин графа, так что матрица D^ и есть искомая матрица расстояний графа.

Как и в алгоритме Воршалла, мы можем вычислить все элементы каждой мат­рицы на основании информации об элементах предшествующей ей матрицы в последовательности (8.8). Пусть — элемент на пересечении г-ой

строки и j-ro столбца матрицы D^k\ Это означает, что dfff равен длине кратчай­шего пути среди всех путей от г-ой вершины Vi к j-ой вершине Vj, промежуточные вершины которых имеют номера не выше к:

Vi ^ Список вершин с номерами, не превышающими к ^ Vj (8.9)

Все такие пути можно разбить на два непересекающихся подмножества: те пути, в которых в качестве промежуточной не участвует к-ая вершина , и те, в которых она является одной из промежуточных. Поскольку пути в первом подмножестве содержат промежуточные вершины с номерами не выше fc — 1, то кратчайший путь между этими вершинами, по определению наших матриц, имеет длину

Чему равна длина кратчайшего пути во втором подмножестве? Если граф не содержит циклов отрицательной длины, мы можем ограничить рассмотрение путей во втором подмножестве только теми, в которые вершина v^ входит только один раз (поскольку посещение вершины vк больше одного раза может только увеличить длину пути). Все такие пути имеют следующий вид:

Vi ^ вершины с номерами ^ к — 1, и\*., вершины с номерами ^ к — 1 ^ Vy

Другими словами, каждый из этих путей состоит из пути от V{ до Vk, причем все промежуточные вершины имеют номера не выше к — 1, и пути от v^ до Vj, у которого все промежуточные вершины также имеют номера не выше к — 1. Схематично данная ситуация изображена на рис. 8.6.

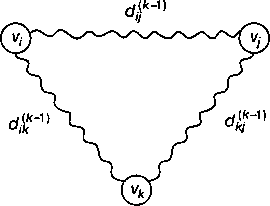


Рис. 8.6. Идея, лежащая в ос­нове алгоритма Флойда

Поскольку длина кратчайшего пути от до v^ среди всех путей, не исполь­зующих промежуточных вершин с номерами больше к — 1, равна d^Tl\ а длина кратчайшего пути от до Vj среди всех путей, не использующих промежуточ­ных вершин с номерами больше к — 1, равна d^~l\ длина кратчайшего пути от Vi

к Vj через вершину Vf~ равна + d^~l\ С учетом длины кратчайших путей

в обоих подмножествах получаем следующее рекуррентное соотношение:

d\f = min |d[j~l\d[l~1^ + ^\_1)| для к ^ 1, d\f = Wij. (8.10)

Т.е. элемент на пересечении г-ой строки и j-го столбца текущей матрицы рас­стояний D^k~1^ заменяется суммой элементов на пересечении той же г-ой строки и fc-ro столбца, и к-ой строки и того же j-го столбца тогда и только тогда, когда эта сумма оказывается меньше текущего значения.

Применение алгоритма Флойда к графу, изображенному на рис. 8.5, показано на рис. 8.7.

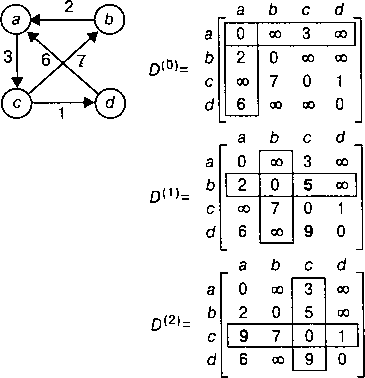
Вот как выглядит псевдокод алгоритма Флойда. В нем используется тот факт, что каждая последующая матрица в последовательности (8.8) может быть запи­сана поверх предшественника.

Алгоритм Floyd (W [l..n, l..n])

// Реализует алгоритм Флойда для решения задачи поиска // кратчайших путей между всеми парами вершин // Входные данные: Весовая матрица W графа

// Выходные данные: Матрица длин кратчайших путей D \*—W //Не требуется, если W может быть перезаписана

Длины кратчайших путей без промежуточных вершин (D(0) представляет собой просто весовую матрицу)



а

О 10 3

2 0 5

9 7 0

Длины кратчайших путей с промежуточными вершинами, номера которых не больше 1, т.е. только а (два новых кратчайших пути от b к с и от d к с)

***bed***

т

Длины кратчайших путей с промежуточными вершинами, номера которых не больше 2, т.е. а и b (новый кратчайший путь от с к а)

**D<3) =**

**6 16 9**

Длины кратчайших путей с промежуточными вершинами, номера которых не больше 3, т.е. а, b и с (четыре новых кратчайший пути от а к Ь, от а к с(, от b к d и от d к Ь)

b с 10 3 0 5 7 О 16 9

d<4>=

Длины кратчайших путей с промежуточными вершинами, номера которых не больше 4, т.е. а, Ь, с и d (новый кратчайший путь от с к а)

Рис. 8.7. Применение алгоритма Флойда к ориентированному графу. Обновленные элементы выделены полужирным шрифтом

**for** к <— 1 **to** п **do for** г **<— 1 to** п **do for** j **<— 1 to** n **do**

*D[i,j]* <- min{£»[i, *j], D[i*, *k]* + *D[k,j}}* return *D*

Очевидно, что временная эффективность алгоритма Флойда — кубическая, как и временная эффективность алгоритма Воршалла. В следующей главе мы рассмотрим еще один метод поиска кратчайших путей — алгоритм Дейкстры.

Мы завершаем этот раздел важным глобальным комментарием, посвященным общему принципу, лежащему в основе применения алгоритмов динамического программирования для оптимизационных задач. Ричард Беллман (Richard Bell­man) назвал его принципом оптимальности (principle of optimality). В несколько измененной по сравнению с оригинальной формулировке он гласит, что оптималь­ное решение любого экземпляра задачи оптимизации составляется из оптималь­ных решений его подэкземпляров. Принцип оптимальности выполняется гораздо чаще, чем не выполняется (в качестве редкого примера его невыполнения можно привести задачу поиска длиннейших простых путей). Хотя, конечно же, примени­мость этого принципа к каждой конкретной задаче должна быть доказана, обычно это не является главной трудностью при разработке алгоритма динамического программирования. Обычно самое сложное — найти, какие наименьшие подэк- земпляры должны быть рассмотрены, и вывести уравнение, связывающее реше­ние любого экземпляра с решениями его меньших подэкземпляров. В оставшейся части главы мы рассмотрим еще несколько примеров алгоритмов динамического программирования.

Упражнения 8.2

1. Примените алгоритм Воршалла для поиска транзитивного замыкания ориентированного графа, определенного следующей матрицей смеж­ности:

**О 1 О о"**

0 0 10 0 0 0 1 0 0 0 0

1. а) Докажите, что временная эффективность алгоритма Воршалла —

кубическая.

б) Поясните, почему временная эффективность алгоритма Воршалла ниже эффективности алгоритма на основе обхода для разрежен­ных графов, представленных при помощи связанных списков смеж­ности.

1. Поясните, как реализовать алгоритм Воршалла без использования до­полнительной памяти для хранения элементов промежуточных матриц.
2. Поясните, как реструктуризировать внутренний цикл алгоритма Warshall так, чтобы он работал быстрее по крайней мере для неко­торых входных данных.
3. Перепишите псевдокод алгоритма Воршалла в предположении, что строки матрицы представлены битовыми строками, к которым может быть применена операция побитового “или” (or).
4. а) Поясните, как алгоритм Воршалла может использоваться для опре­

деления того факта, что данный граф является ориентированным

ациклическим графом. Является ли этот алгоритм подходящим для решения данной задачи?

б) Стоит ли использовать алгоритм Воршалла для поиска транзитив­ного замыкания неориентированного графа?

1. Решите задачу поиска кратчайших путей между всеми парами вершин в ориентированном графе с весовой матрицей

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 2 | оо | 1 | 8 |
| 6 | 0 | 3 | 2 | оо |
| оо | оо | 0 | 4 | оо |
| оо | оо | 2 | 0 | 3 |
| 3 | оо | оо | оо | 0 |

1. Докажите, что очередная матрица в последовательности (8.8) может быть записана поверх предшествующей.
2. Приведите пример ориентированного или неориентированного графа с отрицательными весами, для которого алгоритм Флойда дает невер­ный результат.

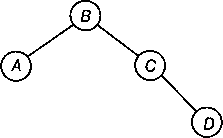
10. Улучшите алгоритм Флойда так, чтобы можно было найти не только длины кратчайших путей, но и сами эти пути.

1. Оптимальные бинарные деревья поиска

Бинарное дерево поиска представляет собой одну из наиболее важных струк­тур данных в кибернетике. Одно из главных ее применений — реализация словаря, множества элементов с операциями поиска, вставки и удаления. Если вероятности поиска элементов множества известны (например, из накопленных данных о по­следних поисках), естественным образом встает вопрос об оптимальном бинарном дереве поиска, для которого среднее количество сравнений является наименьшим возможным (для простоты мы ограничимся минимизацией среднего количества сравнений при успешном поиске. Рассматриваемый метод может быть расширен для включения неудачных поисков).

В качестве примера рассмотрим четыре ключа А, В, С и D, поиск которых осуществляется с вероятностями 0.1, 0.2, 0.4 и 0.3, соответственно. На рис. 8.8 изображены два из четырнадцати возможных бинарных деревьев поиска, содер­жащих эти ключи. Среднее количество сравнений при успешном поиске в первом из этих деревьев равно 0.1 • 1 + 0.2 • 2 + 0.4 • 3 + 0.3 • 4 = 2.9, а во втором —

1. 1 • 2 + 0.2 • 1 + 0.4 • 2 + 0.3 -3 = 2.1. Ни одно из этих деревьев не является оптимальным. (Можете ли вы сказать, какое дерево будет оптимальным?)



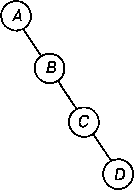


Рис. 8.8. Два из четырнадцати возможных бинар­ных деревьев поиска с ключами А, В, С и D

В этом небольшом примере мы можем найти оптимальное дерево, постро­ив все 14 бинарных деревьев поиска с данными ключами. Однако в качестве универсального алгоритма метод исчерпывающего перебора нереален: общее ко­личество бинарных деревьев поиска с п ключами равно n-му числу Каталина (Catalan number):

*Сп*

с (п) = —при п > 0, с (0) = 1, п + 1

которое растет с ростом п как 4п/п1Ъ (см. упражнение 8.3.7).

Итак, пусть ai,... ,an — различные ключи, упорядоченные от наименьшего к наибольшему, и пусть pi,.. .рп — вероятности их поиска. Пусть С [i,j] — наи­меньшее среднее количество сравнений, выполняемых при успешном поиске в би­нарном дереве Г/, составленном из ключей а\*,..., а^, где г и j — некоторые целые индексы, такие, что 1 ^ г ^ j < п. Таким образом, следуя классическому подхо­ду динамического программирования, мы будем искать значения С [i,j] для всех меньших экземпляров задачи, хотя нас интересует только значение С [1, п]. Чтобы вывести рекуррентное соотношение, лежащее в основе алгоритма динамического программирования, надо рассмотреть все возможные способы выбора корня а& среди узлов а^,... , a^. Для такого бинарного дерева (рис. 8.9) корень содержит ключ а^, левое поддерево if-1 — оптимально упорядоченные ключи ..., ak-i, а правое поддерево Т^+1 — оптимально упорядоченные ключи ь ..., aj (обра­тите внимание на использование здесь принципа оптимальности).

Если мы будем считать уровни дерева, начиная с 1 (для того, чтобы количе­ство сравнений равнялось уровню ключа), то получим следующее рекуррентное соотношение:

fc-i

**С** [г, **j]** = min i **рк** • 1 + **ps** • (Уровень **as** в 1 + l) +

i^k^J —' \ /

**k *s—i***

+ ^2 Ps • ^Уровень as в T3k+l + 1^ > = s=k-\-1 J

{

fc-1 fc-1

Рк + • Уровень as в Tf\_1 + У"]ps+

5=г s=l

***3 3 Л***

+ ^ ps • Уровень os в \*k.+ E Ps

[S=fc + **1** S—**fc+1 J**](#bookmark259)

fc-1

= min < У" ps • Уровень as в Tf 1+

i^k^J 1 f

+ y^ ps • Уровень as в ■^fc+i + | —

***s=k+l s=i )***

3

***=*** min ***{C[i,k-l] + C[k + l,j]} + ^2ps.***

S=l

Таким образом, мы получаем рекуррентное соотношение

***j***

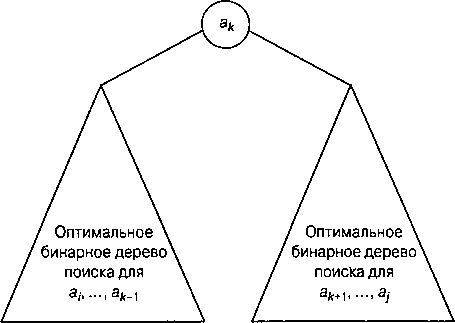


Рис. 8.9. Бинарное дерево поиска с корнем и дву­мя оптимальными бинарными поддеревьями поис-

ка Т\*-1 и Т1

fc+1

C[iJ] = min {С [г, к - 1] + С [к + IJ]} + У^р3 для 1 ^ г ^ j < п. (8.11)

В формуле (8.11) мы полагаем, что C[i,i — 1] = 0 при 1 < г < n + 1, что можно рассматривать как количество сравнений в пустом дереве. Заметим: из этой формулы следует, что

С [г, г] = pi при 1 ^ г ^ тг, как и должно быть для бинарного дерева с одним узлом щ.

0 1 jn

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | Pi |  |  | |  | |  | |  | Целевое  значение |
|  | 0 | Рг |  | |  | |  | |  |  |
|  |  | t | А | i. | А |  | А | к. | сил |  |
|  | | L |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  | | |  | V |  |  |  |  |  |  |
|  | | | |  |  |  |  |  |  |
|  | | | | |  | Ч |  |  |  |  |
|  | | | | | |  |  |  |  |
|  | | | | | | |  | ч |  | Рп |
|  | | | | | | | |  | W |
|  | | | | | | | | |  | 0 |

Рис. 8.10. Таблица алгоритма динамического про­граммирования для построения оптимального би­нарного дерева поиска

Двумерная таблица на рис. 8.10 показывает значения, необходимые для вы­числения С [i,j] по формуле (8.11): они находятся в строке г в столбцах слева от столбца j, и в столбце j ниже строки г. Стрелками показаны пары элементов таб­лицы, суммы которых вычисляются для поиска наименьшего значения, которое будет записано в качестве значения C[i,j]. Сказанное предполагает заполнение таблицы вдоль диагоналей, начиная с нулевых значений на главной диагонали и заданных вероятностей pi, 1 ^ г ^ п, прямо над ней, и перемещаясь к верхнему правому углу.

Описанный алгоритм вычисляет С [1, п] — среднее количество сравнений при успешном поиске в оптимальном бинарном дереве. Если мы, кроме того, хо­тим получить само оптимальное бинарное дерево поиска, то надо поддерживать еще одну двумерную таблицу для записи значений к, при которых достигается минимум в (8.11). Эта таблица имеет ту же форму, что и таблица на рис. 8.10, и заполняется точно так же, — начиная с элементов R [г, г] = г для 1 ^ г ^ п. Когда

таблица оказывается заполненной, ее элементы указывают индексы корней опти­мальных поддеревьев, которые позволяют реконструировать оптимальное дерево для всего множества.

Пример 1. Проиллюстрируем данный алгоритм, применяя его ко множеству из четырех ключей, которое было упомянуто в начале раздела:

Ключ ***А В С D***

Вероятность 0.1 0.2 0.4 0.3

Начальные таблицы выглядят следующим образом:

Главная таблица Таблица корней

01234 01234

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 0 0.1 | 1 | 1 |
| 2 | 0 0.2 | 2 | 2 |
| 3 | 0 0.4 | 3 | 3 |
| 4 | 0 0.3 | 4 | 4 |
| 5 | 0 | 5 |  |

Давайте вычислим С [ 1,2]:

**С [1,21** = **min** = 1 : **с I1' °1 +** с **I2’ 21 +** §=■ Л - **0** + **М** + **0** л = **0.5** \_

1 J к = 2 : C[1,1] + C[3,2] + ELiPS = 0-1 + 0 + °-3 = 0-4

Таким образом, из двух возможных бинарных деревьев, содержащих первые два ключа, А и В, корень оптимального дерева имеет индекс 2 (т.е. содержит В), а среднее количество сравнений при успешном поиске в этом дереве равно 0.4.

Завершите пример самостоятельно. Вы должны получить следующие оконча­тельные таблицы:

Главная таблица Таблица корней

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |  | 0 | 1 2 | 3 | 4 |
| 1 | 0 | 0.1 | 0.4 | 1.1 | 1.7 | 1 |  | 1 2 | 3 | 3 |
| 2 |  | 0 | 0.2 | 0.8 | 1.4 | 2 |  | 2 | 3 | 3 |
| 3 |  |  | 0 | 0.4 | 1.0 | 3 |  |  | 3 | 3 |
| 4 |  |  |  | 0 | 0.3 | 4 |  |  |  | 4 |
| 5 |  |  |  |  | 0 | 5 |  |  |  |  |

Таким образом, среднее количество сравнений ключей в оптимальном дереве равно 1.7. Поскольку R [1,4] = 3, корень оптимального дерева содержит третий

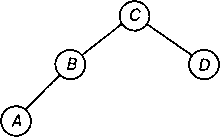


Рис. 8.11. Оптимальное бинарное дерево поиска из примера 1

ключ, т.е. С. Его левое поддерево состоит из ключей А и В, а правое содержит единственный ключ D (почему?). Чтобы определить структуру поддеревьев, мы должны сначала найти их корни с использованием таблицы корней следующим образом. Поскольку Д[1,2] = 2, корнем оптимального дерева, содержащего А и В, является В, причем А — его левый дочерний узел (и корень дерева с одним узлом: Л [1,1] = 1). Поскольку Л [4,4] = 4, корнем этого оптимального дерева с одним узлом является D. Полученное оптимальное бинарное дерево поиска приведено на рис. 8.11. ■

Вот как выглядит псевдокод рассмотренного алгоритма.

**Алгоритм Optimal В ST** **(Р** **[1..п])**

// Поиск оптимального бинарного дерева поиска // с использованием динамического программирования // Входные данные: Массив Р[1..га] вероятностей поиска для

// отсортированного списка из п ключей

// Выходные данные: Среднее количество сравнений при // успешном поиске в оптимальном бинарном

// дереве поиска и таблица корней

// поддеревьев оптимального бинарного

// дерева поиска R

for i «— 1 to ft do C[i, г — 1] <— 0 C[i, i] «- P[i] i?[z, i] \*— i C[n+ l,n] <— 0

for d \*— 1 to n — 1 do // Счетчик диагоналей for i \*— 1 to n — d do j \*-i + d minval \*— oo for к \*— i to j do if С[г, к — 1] + C[k + l,j] < minval

minval <— С[г, к — 1] + C[fc + kmin <— к

R[iij] <— ктгп sum <— Р[г]; for s «— г + 1 to j do sum <— sum + P[s]

***C[i:j] \*— minval*** + ***sum*** return ***C[l,n],R***

Очевидно, что алгоритму требуется дополнительная память, квадратично за­висящая от п; временная эффективность алгоритма — кубическая (почему?). Бо­лее точный анализ показывает, что элементы таблицы корней всегда находятся в неубывающем порядке как вдоль строк, так и вдоль столбцов. Это ограничивает значения R [г, j] диапазоном R [г, j — 1],..., R [г + 1, j] и позволяет снизить время работы алгоритма до © (п2).

Упражнения 8.3

1. Завершите построение оптимального бинарного дерева поиска, начатое в примере в тексте раздела.
2. а) Почему временная эффективность алгоритма OptimalBST кубиче­

ская?

б) Почему пространственная эффективность алгоритма OptimalBST квадратичная?

1. Напишите псевдокод алгоритма с линейным временем работы, который генерирует оптимальное бинарное дерево поиска по таблице корней.
2. Разработайте способ вычисления за постоянное время сумм используемых в алгоритме динамического программирования для по­строения оптимального бинарного дерева поиска.
3. Истинно или ложно следующее утверждение: корень оптимального би­нарного дерева поиска всегда содержит ключ с наивысшей вероятно­стью поиска.
4. Как бы вы построили оптимальное бинарное дерево поиска для мно­жества из п ключей, если вероятности поиска у всех ключей равны? Чему будет равно среднее количество сравнений в дереве, если п — 2к1
5. а) Покажите, что количество различных бинарных деревьев поиска

Ь (п), которые могут быть построены для множества из п упорядо­чиваемых ключей, удовлетворяет рекуррентному соотношению

П— 1

Ъ (тг) = ^ b (к) b (тг — 1 — к) для тг > О, b (0) = 1.

к=0

б) Известно, что решение этого рекуррентного соотношения дает числа Каталана. Убедитесь в этом для п = 1,2,..., 5.

в) Найдите порядок роста Ъ{п). Как ответ на этот вопрос влияет на применение алгоритма исчерпывающего перебора для поиска опти­мального бинарного дерева поиска?

1. Разработайте алгоритм поиска оптимальных бинарных деревьев поиска за время 0 (п2).
2. Обобщите алгоритм построения оптимальных бинарных деревьев по­иска с учетом неудачных поисков.
3. Перемножение цепочек матриц. Рассмотрим задачу минимизации об­щего количества умножений, выполняемых при вычислении произве­дения п матриц

А\ • А2 •... • Ап,

размеры которых — do х d\, d\ х d2> • • • 1 х dn, соответственно.

Считаем, что все промежуточные произведения двух матриц вычисля­ются при помощи алгоритма грубой силы (основанном на определении умножения матриц).

а) Приведите пример трех матриц, для которых количество умножений при вычислении (А\ • А2) • As и А\ • (А2 • As) отличается не меньше чем в 1000 раз.

б) Сколько всего имеется различных способов вычислить произведе­ние п матриц?

в) Разработайте алгоритм динамического программирования для поис­ка оптимального порядка перемножения п матриц.

1. Задача о рюкзаке и функции с запоминанием

Этот раздел мы начнем с разработки алгоритма динамического программи­рования для решения задачи о рюкзаке: даны п предметов с известными весами wi,... ,wnvi стоимостями ui,..., vn и рюкзак вместимостью W. Требуется найти наиболее ценное подмножество предметов, помещающееся в рюкзаке. (Эта задача упоминалась в разделе 3.4, где мы рассматривали ее решение методом исчерпыва­ющего перебора.) Здесь мы считаем, что все веса и емкость рюкзака представляют

собой положительные целые числа; стоимости предметов — не обязательно целые числа.

Для разработки алгоритма динамического программирования мы должны вы­вести рекуррентное соотношение, которое выражает решение экземпляра задачи о рюкзаке через решения его меньших подэкземпляров. Рассмотрим экземпляр, определяемый первыми г предметами, 1 ^ г ^ п, весами w\,..., wu стоимостями vi,..., Vi и емкостью рюкзака 1 ^ j ^ W. Пусть V [г, j] — значение оптимального решения этого экземпляра, т.е. стоимость наиболее ценного подмножества из пер­вых г предметов, которое помещается в рюкзак емкостью j. Мы можем разделить все подмножества первых г предметов, которые помещаются в рюкзак емкостью j, на две категории: те, которые не включают г-ый предмет, и те, которые его включают. Заметим следующее.

1. Среди подмножеств, которые не включают г-ый предмет, стоимость оптимального подмножества по определению равна V[i—l,j].
2. Среди подмножеств, которые включают г-ый предмет (следовательно, j — Щ ^ 0), оптимальное подмножество составляется из этого предмета и оптимального подмножества первых г — 1 предметов, которое разме­щается в рюкзаке емкостью j — Стоимость такого оптимального подмножества равна Vi + V [г — l,j — Wi\.

Таким образом, стоимость оптимального решения среди всех допустимых под­множеств из первых г предметов представляет собой большее из этих двух значе­ний. Конечно, если г-ый предмет не помещается в рюкзак, стоимость оптималь­ного подмножества, выбранного из первых г предметов, оказывается той же, что и стоимость оптимального подмножества, выбранного из первых г — 1 предметов. Это наблюдение приводит нас к следующему рекуррентному соотношению:

т/г- -1 \_ Г max ■[V' [i — 1, j], Wi + V [\* — 1, j — tu\*]> если j - wt ^ 0,

\* I т/ Г- i 1 ^ n

I У [г - \,j] если j — Wi < 0.

Начальные условия удобно определить следующим образом:

V [0, j] = 0 при j ^ 0, и V [г,0] = 0 приг ^ 0. (8.13)

Наша цель состоит в том, чтобы найти V [n, W] — максимальную стоимость под­множества из п предметов, которое помещается в рюкзаке емкостью W, и само это подмножество.

На рис. 8.12 показаны значения, входящие в (8.12) и (8.13). При г, j > 0 для вычисления элемента таблицы на пересечении г-ой строки и j-ro столбца V [г, j] мы берем значение элемента в предыдущей строке и том же столбце и сумму значений уг и элемента в предыдущей строке и столбце, отстоящем на u>i столбцов слева, и находим максимальное из них. Таким образом мы заполняем таблицу либо строка за строкой, либо столбец за столбцом.

О j -Wi j W

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 0 | 0 |
| Wi,Vi i — 1 | 0 | V[i-l,j -Wi\ V[i-l,j] |  |
| i | 0 | V[i,j] |  |
| п | 0 |  | Целевое значение |

Рис. 8.12. Таблица для решения задачи о рюкзаке методом динамиче­ского программирования

Пример 1. Рассмотрим экземпляр задачи, определяемый следующими данными. Емкость рюкзака W = 5.

|  |  |  |
| --- | --- | --- |
| Предмет | Вес | Стоимость |
| 1 | 2 | 12 |
| 2 | 1 | 10 |
| 3 | 3 | 20 |
| 4 | 2 | 15 |

Таблица динамического программирования после заполнения в соответствии с формулами (8.12) и (8.13) показана на рис. 8.13.

Емкость j

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | i | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| wi = 2, | vi = 12 | 1 | 0 | 0 | 12 | 12 | 12 | 12 |
| w2 = 1, | V2 = 10 | 2 | 0 | 10 | 12 | 22 | 22 | 22 |
| 00  II  со  5 | о  <N  II  CO | 3 | 0 | 10 | 12 | 22 | 30 | 32 |
| W4 - 2, | V4 = 15 | 4 | 0 | 10 | 15 | 25 | 30 | 37 |

Рис. 8.13. Пример решения экземпляра задачи о рюк­заке при помощи алгоритма динамического программи­рования

Итак, максимальная стоимость V [4,5] = 37. Мы можем найти состав опти­мального подмножества, отслеживая вычисления этого элемента таблицы. По­скольку V [4,5] Ф V [3,5], предмет 4 был включен в оптимальное решение вме­сте с оптимальным подмножеством, заполняющим оставшиеся 5 — 2 = 3 еди­ницы емкости рюкзака. Последние представлены элементом V [3,3]. Поскольку V [3,3] = V [2,3], элемент 3 не является частью оптимального подмножества. Да-

лее, так как V" [2,3] ф V [1,3], предмет 2 также является частью оптимального выбора, после чего элемент V [1,3 — 1] остается в качестве определения остав­шейся части подмножества. Аналогично, так как V [1,2] ф V [0,2], делаем вывод, что предмет 1 является последней частью оптимального решения, которое пред­ставляет собой множество {Предмет 1, Предмет 2, Предмет 4}. ■

Как временная, так и пространственная эффективность данного алгоритма равна 0 (nW). Время, требующееся для поиска состава оптимального подмноже­ства, равно 0 (n + W). Эти утверждения читателю предлагается доказать само­стоятельно в качестве упражнений.

**Функции с запоминанием**

Как говорилось в начале главы и было показано в последующих разделах, динамическое программирование работает с задачами, решения которых удовле­творяют рекуррентным соотношениям с перекрывающимися подзадачами. Непо­средственный нисходящий подход к поиску решения такого рекуррентного соот­ношения приводит к алгоритму, который решает общие подзадачи несколько раз, а следовательно, крайне неэффективен (обычно такой алгоритм экспоненциален или имеет еще более низкую эффективность). С другой стороны, классический подход динамического программирования работает в восходящем направлении: он заполняет таблицу решениями всех подзадач меньшего размера, но зато каж­дую он решает только один раз. Неприятной стороной такого подхода является то, что решения ряда задач меньшего размера могут быть не нужны для получе­ния решения исходной задачи. Поскольку нисходящий подход лишен этого недо­статка, представляется естественным попытаться объединить сильные стороны нисходящего и восходящего подходов. Цель заключается в том, чтобы получить метод, который позволяет решать только необходимые подзадачи и только один раз. Такой метод существует; он основан на функциях с запоминанием (memory functions) [22].

Этот метод решает поставленную задачу в нисходящем направлении, но, кро­ме того, поддерживает таблицу такого же вида, как и используемые в восходящих алгоритмах динамического программирования. Изначально все записи таблицы инициализируются специальным значением null, которое указывает, что дан­ное значение еще не было вычислено (здесь может пригодиться метод вирту­альной инициализации (virtual initialization), который рассматривался в упражне­нии 7.1.8). Затем, когда требуется вычислить новое значение, данный метод снача­ла проверяет, не было ли оно уже вычислено ранее. Если соответствующая запись в таблице не равна NULL, то из таблицы просто извлекается уже вычисленное ранее значение; в противном случае оно вычисляется при помощи рекурсивного вызова, и полученный результат записывается в таблице.

Приведенный далее алгоритм реализует эту идею для задачи о рюкзаке. После инициализации таблицы рекурсивная функция вызывается с г = п (количество предметов) и j = W (емкость рюкзака).

**Алгоритм MFKnapsack** (г, **j**)

// Реализация метода функций с запоминанием для решения

// задачи о рюкзаке

// Входные данные: Натуральное г, указывающее количество

// первых рассматриваемых предметов,

// и натуральное j — емкость рюкзака

// Выходные данные: Стоимость оптимального допустимого // подмножества из первых i предметов

// Примечание: Использует входные массивы

// ***Weights\l..rii\,*** *Values[\..n\*и таблицу

// У[0..п, 0..W] (элементы которой

// инициализированы значениями —1, за

// исключением строки 0 и столбца О,

// элементы которых инициализированы

// значениями 0) в качестве глобальных

// переменных

***UV[i,j]<0*** if ***j < Weights[i]***

***value \*— MFKnapsack(i — I, j)*** else

***value*** <— ma***x(MFKnapsack(i*** — 1 ***,j),***

***Values[i] + MFKnapsack(i*** — 1 ***,j — Weights[i]))***

***V[i,j\*** <— ***value*** return ***V[i,j]***

Пример 2. Применим метод функций с запоминанием к экземпляру задачи, рас­смотренному в примере 1. На рис. 8.14 показаны результаты вычислений. Как видите, вычислены только 10 из 20 нетривиальных (т.е. не находящихся в нуле­вой строке или нулевом столбце) значений. Только одно нетривиальное значение, V [1,2], было повторно выбрано из таблицы вместо вычисления заново. Однако для больших экземпляров задачи эта пропорция может оказаться существенно большей. ■

В общем случае мы не можем ожидать повышения скорости работы алгоритма решения задачи о рюкзаке с использованием функций с запоминанием по срав­нению с рассмотренным ранее более чем на некоторый постоянный множитель. Это связано с тем, что класс эффективности алгоритма с применением функций с запоминанием тот же, что и у восходящего алгоритма (почему?). Более суще­ственного улучшения можно ожидать для алгоритмов динамического программи-

Емкость j

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | г | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Wi = 2, | v\ = 12 | 1 | 0 | 0 | 12 | - | 12 | 12 |
| W2 = 1, | V2 — Ю | 2 | 0 | - | 12 | 22 | - | 22 |
| 00  II  СО  5 | о  (N  II  со | 3 | 0 | - | - | 22 | - | 32 |
| W4 = 2, | V4 = 15 | 4 | 0 | - | - | - | - | 37 |

Рис. 8.14. Пример решения экземпляра задачи о рюкза­ке при помощи алгоритма с использованием функций с запоминанием

рования, в которых одно вычисление требует больше константного времени. Вы должны также не забывать, что метод функций с запоминанием может оказаться более расточительным в плане использования памяти, чем более эффективный в этом смысле восходящий алгоритм.

Упражнения 8.4

1. а) Воспользуйтесь восходящим алгоритмом динамического програм­мирования для решения следующего экземпляра задачи о рюкзаке. Емкость рюкзака W = 6.

|  |  |
| --- | --- |
| Предмет | Вес Стоимость |
| 1 | 3 25 |
| 2 | 2 20 |
| 3 | 1 15 |
| 4 | 4 40 |
| 5 | 5 50 |

б) Сколько различных оптимальных подмножеств имеет экземпляр за­дачи из части я)?

в) Как в общем случае мы можем воспользоваться таблицей, сгене­рированной алгоритмом динамического программирования, чтобы выяснить, имеется ли у данного экземпляра задачи о рюкзаке более одного оптимального подмножества.

1. а) Напишите псевдокод восходящего алгоритма динамического про­граммирования для задачи о рюкзаке.

б) Напишите псевдокод алгоритма, который находит состав оптималь­ного подмножества по таблице, сгенерированной восходящим алго­ритмом динамического программирования для задачи о рюкзаке.

1. Докажите для восходящего алгоритма динамического программирова­ния для задачи о рюкзаке:

а) что его временная эффективность равна 0 (nW),

б) что его пространственная эффективность равна 0 (nW),

в) что время, требующееся для поиска состава оптимального подмно­жества по заполненной алгоритмом динамического программирова­ния таблице, равно 0 (п + W).

1. а) Истинно или ложно следующее утверждение: последовательность

стоимостей в строке таблицы динамического программирования для экземпляра задачи о рюкзаке всегда неубывающая,

б) Истинно или ложно следующее утверждение: последовательность стоимостей в столбце таблицы динамического программирования для экземпляра задачи о рюкзаке всегда неубывающая.

1. Примените метод функций с запоминанием для решения экземпляра задачи о рюкзаке из упражнения 1. Укажите элементы таблицы дина­мического программирования, которые 1) не вычисляются при исполь­зовании метода функций с запоминанием; 2) выбираются из таблицы без вычисления.
2. Докажите, что класс эффективности алгоритма с функциями с запоми­нанием для задачи о рюкзаке тот же, что и у восходящего алгоритма (см. упражнение 3).
3. Напишите псевдокод функции с запоминанием для задачи об оптималь­ном бинарном дереве поиска (можете ограничиться поиском наимень­шего количества сравнений ключей при успешном поиске).
4. Приведите две причины, по которым подход с использованием функций с запоминанием не подходит для задачи вычисления биномиальных коэффициентов.
5. Разработайте алгоритм динамического программирования для задачи о размене (change-making problem): дана величина п и неограниченное количество монет достоинством di,d2,... ,dm. Требуется найти наи­меньшее количество монет, которые в сумме дают величину п, или указать, что задача не имеет решения.
6. Напишите реферат об одном из хорошо известных применений дина­мического программирования:

а) Поиск наибольшей общей подпоследовательности двух последова­тельностей.

б) Оптимальное редактирование строк.

в) Минимальная триангуляция многоугольника.

Резюме

* Динамическое программирование представляет собой метод решения задач с перекрывающимися подзадачами. Обычно такие подзадачи воз­никают из рекуррентных соотношений, связывающих решение постав­ленной задачи с решениями меньших подзадач того же вида. Динами­ческое программирование предполагает, что решение каждой из мень­ших подзадач находится только один раз, и записывает полученные результаты в таблицу, из которой затем получается решение исходной задачи.
* Применимость динамического программирования к задачам оптими­зации требует, чтобы задача удовлетворяла принципу оптимальности'. оптимальное решение любого из ее экземпляров должно слагаться из оптимальных решений ее подэкземпляров.
* Вычисление биномиального коэффициента путем построения треуголь­ника Паскаля может рассматриваться как применение метода динами­ческого программирования к задаче, не являющейся задачей оптими­зации.
* Алгоритм Воршалла для поиска транзитивного замыкания и алгоритм Флойда для решения задачи поиска кратчайших путей между всеми парами вершин основаны на идее, которую можно рассматривать как применение метода динамического программирования.
* Динамическое программирование можно использовать для построения оптимального бинарного дерева поиска для данного множества ключей и известных вероятностей их поиска.
* Решение задачи о рюкзаке при помощи алгоритма динамического про­граммирования является примером применения этого метода к слож­ным задачам комбинаторной оптимизации.
* Метод функций с запоминанием призван объединить сильные стороны нисходящего и восходящего подходов к решению задач с перекрываю­щимися подзадачами. Это достигается решением задачи в нисходящем направлении, с однократным решением только необходимых подзадач исходной задачи и записью полученных решений в таблице.

**Глава**

Жадные методы

Жадность, за отсутствием лучшего слова, — это здорово! Это правильно!

— Майкл Дуглас (Michael Douglas), американский актер, в роли Гордона Геко (Gordon Gecko) в фильме

“Уолл-стрит” (Wall Street), 1987

Н

ачнем с задачи о размене (change-making problem), которая постоянно вста­ет перед миллионами кассиров во всем мире: как выплатить сумму п при помощи наименьшего количества монет номиналом d\ > d<i > • • • > dm, исполь­зующихся в той или иной стране. Например, в США широко распространены монеты достоинством d\ = 25 центов (quarter), d<i = 10 центов (dime (гривен­ник)), ds = 5 центов (nickel (пятак)) и d± = 1 цент (penny). Как дать такими монетами сдачу, например, в 48 центов? Если вы ответите, что это одна монета в 25 центов, две — по 10 центов и три — по 1 центу, то, сознательно или нет, вы следуете стратегии, которая позволяет сделать оптимальный выбор среди возмож­ных альтернатив. На первом шаге вы можете выбрать монету любого номинала из четырех приведенных. “Жадное” мышление приводит вас к мысли, что это должна быть монета в 25 центов, что максимально снизит остающуюся к выдаче сумму — а именно до 23 центов. На втором шаге вы уже не можете выбрать моне­ту в 25 центов, поскольку это нарушило бы ограничения, поставленные в условии задачи. Поэтому наилучшим выбором в данной ситуации оказывается выбор мо­неты в 10 центов, что снижает сумму до 13 центов. Еще один выбор монеты в 10 центов приводит нас к остатку в 3 цента, которые можно дать только тремя монетами по 1 центу.

Является ли это решение экземпляра задачи о размене оптимальным? Да. Можно доказать, что жадный алгоритм для данных номиналов монет дает опти­мальное решение для любой суммы, выражающейся натуральным числом. В то же время легко привести пример необычных номиналов монет (например, d\ = 7, g?2 = 5, с?з = 1), которые для определенных сумм могут дать некорректные ре­шения (именно по этой причине при разработке алгоритма динамического про­граммирования для данной задачи в упражнении 8.4.9 оговаривалось, что алго­ритм либо должен вернуть оптимальное решение, либо сообщить об отсутствии решения).

Такой подход к решению задачи о размене называется жадным (greedy). Кибернетики рассматривают его как общий метод проектирования алгоритмов, несмотря на то, что он применим только к задачам оптимизации. Жадный под­ход предполагает построение решения посредством выбора последовательности шагов, на каждом из которых получается частичное решение поставленной зада­чи, пока не будет получено полное решение. При этом на каждом шаге — и это является главным в рассматриваемом методе — выбор должен быть

* допустимым, т.е. удовлетворять ограничениям задачи;
* локально оптимальным, т.е. наилучшим локальным выбором среди всех допустимых вариантов, доступных на каждом шаге;
* окончательным, т.е. будучи сделан, он не может быть изменен после­дующими шагами алгоритма.

Эти требования поясняют название метода: на каждом шаге он предполагает “жадный” выбор наилучшей доступной альтернативы в надежде, что последова­тельность локально оптимальных выборов приведет к (глобально) оптимальному решению всей задачи. Мы не будем вдаваться в философский спор о том, хоро­шо ли быть жадным (если вы не смотрели фильм, упомянутый в эпиграфе, то для справки: его герой плохо кончил). С точки зрения алгоритмов нас интересует лишь то, работает метод или нет. Как мы увидим, существуют задачи, для которых последовательность локально оптимальных выборов приводит к оптимальному решению для любого экземпляра рассматриваемой задачи. Однако есть и другие задачи, для которых это не так; для задач такого рода жадный алгоритм может представлять интерес только в том случае, если нас устраивает приближенное решение.

В двух первых разделах главы мы рассмотрим два классических алгоритма для решения задачи о минимальном остовном дереве: алгоритмы Прима и Крускала. В этих алгоритмах примечателен тот факт, что они решают одну и ту же задачу с применением жадного подхода различными способами, и оба дают оптимальное решение. В разделе 9.3 мы рассмотрим еще один классический алгоритм — ал­горитм Дейкстры для поиска кратчайшего пути во взвешенном графе. Раздел 9.4 посвящен деревьям Хаффмана и их основному приложению — кодам Хаффмана — важному методу сжатия данных, который может рассматриваться как применение жадной технологии. Наконец, в разделе 11.3 вы встретитесь с несколькими при­мерами приближенных алгоритмов, основанных на жадном методе.

Как правило, жадные алгоритмы интуитивно привлекательны и просты. Не­смотря на несомненную простоту, за ними стоит сложная теория, основанная на абстрактной комбинаторной структуре, которая называется “матроид” (matroid).

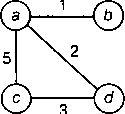
Мы не будем рассматривать матроиды; заинтересованным читателям рекомендуем обратиться к другим книгам, например к [32].

1. Алгоритм Прима

Во многих практических ситуациях естественным образом возникает следую­щая задача: требуется соединить п данных точек таким образом, чтобы из каждой точки существовал путь в любую другую, причем суммарная стоимость соеди­нений должна быть минимальна. Точки можно представить вершинами графа, а стоимости соединений — весами ребер. В такой модели задача превращается в задачу о минимальном остовном дереве, формально определяемом следующим образом.

Определение 1. Остовное дерево (spanning tree) связного графа представляет собой связный ациклический подграф (т.е. дерево), которое содержит все верши­ны графа. Минимальное остовное дерево (minimum spanning tree) взвешенного связного графа представляет собой остовное дерево с наименьшим весом, где вес дерева определяется как сумма весов всех его ребер. Задача о минимальном остов­ном дереве представляет собой задачу поиска минимального остовного дерева для данного взвешенного связного графа. ■

На рис. 9.1 показан простой пример, иллюстрирующий приведенные опреде­ления.



Q—L-® ©—L-© ©—L-©

\2 5 5

[**©-?-©** (°У~г**^© © ©**](#bookmark271)

Граф w(71) = 6 w(T2) = 9 w(7"3) = 8

Рис. 9.1. Граф и его остовные деревья. Т\ — минимальное остовное дерево

Если мы попытаемся применить к построению минимального остовного дере­ва метод исчерпывающего перебора, то встретим два серьезных препятствия. Во- первых, количество остовных деревьев экспоненциально растет с ростом размера графа (как минимум для плотных графов). Во-вторых, генерация всех остовных деревьев для графа — задача непростая; в действительности она гораздо сложнее поиска минимального остовного дерева взвешенного графа одним из эффективных алгоритмов для решения этой задачи. В этом разделе мы рассмотрим алгоритм Прима (Prim’s algorithm), который был разработан еще в 1957 году [92].

Алгоритм Прима строит минимальное остовное дерево как последователь­ность расширяющихся поддеревьев. Начальное поддерево такой последователь­ности состоит из единственной вершины, произвольно выбранной из множества вершин графа V. На каждой итерации мы расширяем текущее дерево жадным образом, добавляя к нему ближайшую вершину, не входящую в дерево (под бли­жайшей вершиной подразумевается вершина, не входящая в дерево и соединенная с вершиной дерева ребром с минимальным весом. Неоднозначности разрешаются произвольным образом). Алгоритм завершает работу после того, как все верши­ны оказываются включены в строящееся дерево. Поскольку алгоритм расширяет дерево по одной вершине за итерацию, общее количество таких итераций равно п — 1, где п — количество вершин графа. Дерево, сгенерированное этим алго­ритмом, представляет собой множество ребер, использованных при расширении дерева.

Вот как выглядит псевдокод данного алгоритма.

Алгоритм **Prim** (**G**)

// Алгоритм Прима построения минимального остовного дерева // Входные данные: Взвешенный связный граф G = (V, Е)

// Выходные данные: Ет, множество ребер, составляющих // минимальное остовное дерево G

Vt <— {г>о} // Множество вершин дерева инициализируется

// произвольной вершиной

Ет <— 0

for i <— 1 to |V| — 1 do

Поиск ребра с минимальным весом е\* = {у\*, и\*) среди всех ребер (v, и) таких, что veVrnueV — Vt Vt <— Vt U {t^\*}

Et <— Et U {e\* } return Et

Природа алгоритма Прима заставляет снабдить каждую вершину, не входящую в текущее дерево, информацией о кратчайшем ребре, соединяющем ее с верши­ной дерева. Мы можем сделать это, присоединяя к вершинам по две метки: имя ближайшей вершины дерева и длину (вес) соответствующего ребра. Вершины, не являющиеся смежными ни для одной из вершин дерева, могут быть помечены меткой оо, указывающей “бесконечное” расстояние до вершин дерева и нулевым значением в поле ближайшей вершины дерева. (Альтернативный вариант состоит в разделении всех вершин, не входящих в дерево, на два множества — “пригранич­ных” и “незамеченных”. Приграничное множество содержит только те вершины, которые, не принадлежа дереву, смежны по крайней мере с одной из его вершин. Незамеченные вершины — все остальные вершины графа, которые называются так потому, что еще не исследованы алгоритмом.) При использовании таких ме­ток поиск следующей добавляемой в текущее дерево Т = (Vt,Et) вершины становится простой задачей поиска вершины с наименьшей меткой расстояния среди вершин множества V — Vt- Разрешение неоднозначностей выполняется произвольным образом.

После того как мы найдем вершину и\*, которая должна быть добавлена в де­рево, надо выполнить следующие две операции.

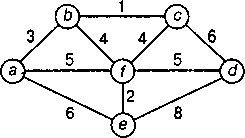
* Переместить вершину и\* из множества V — Vt во множество вершин дерева Vt-
* Для каждой вершины из остающихся во множестве V — Vt и связанных с и\* более коротким ребром, чем ее текущая метка расстояния, следу­ет обновить ее метку имени смежной вершины именем и\*, а метку расстояния — расстоянием до вершины и\*}

На рис. 9.2 продемонстрировано применение алгоритма Прима к показанному на рисунке графу.

Всегда ли алгоритм Прима дает минимальное остовное дерево? Ответ на этот вопрос — да. Докажем по индукции, что каждое поддерево X\*, г = 0,..., п — 1, ге­нерируемое алгоритмом Прима, является частью (т.е. подграфом) некоторого ми­нимального остовного дерева. (Отсюда, очевидно, непосредственно следует, что последнее дерево последовательности, Тп-ь представляет собой само минималь­ное остовное дерево, поскольку содержит все п вершин графа.) Базис индукции тривиален, поскольку дерево Го состоит из одной вершины, а значит, должно быть частью любого минимального остовного дерева. Для выполнения шага индукции предположим, что Ti-\ является частью некоторого минимального остовного де­рева Т. Надо доказать, что дерево Г\*, сгенерированное алгоритмом Прима из дерева Ti\_i, также является частью минимального остовного дерева. Мы дока­жем это от противного, предположив, что не существует минимального остовного дерева графа, содержащего Т\*. Пусть е\* = (г>, и) — ребро минимального веса от вершины в дереве Ti-1 к вершине, которая не входит в Ti-1 и используется ал­горитмом Прима для расширения дерева Ti-\ до Ti. В соответствии с нашими предположениями е\* не может принадлежать ни одному из минимальных остов­ных деревьев, включая Г. Таким образом, если мы добавим е\* к дереву Г, должен образоваться цикл (рис. 9.3).

Кроме ребра е\* = (v,u) цикл должен содержать другое ребро кото­

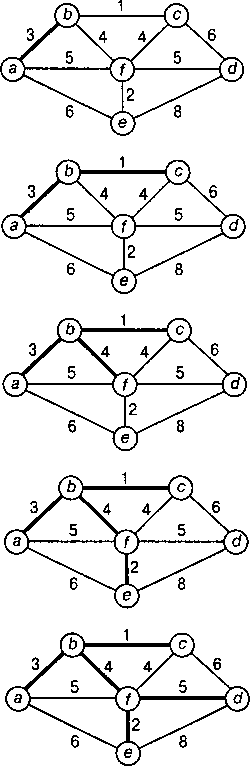
рое соединяет вершину vf е Ti-1 с вершиной и7 ^ Ti-1 (vf может совпадать с v или и7 может совпадать с и, но не оба одновременно). Если теперь мы удалим ребро (vf, и') из цикла, то получим другое остовное дерево всего графа, вес ко­торого не превышает веса дерева Г, поскольку вес е\* не превышает веса (г/, и[[55]](#footnote-55)).



Вершины дерева

Рисунок

Остальные вершины[[56]](#footnote-56)



Ь(а,3)

с(Ь,1)

f(b,4)

e(f,2)

b(a,3) **c(-,oo) d(-,oo) e(a,6) f(a,5)**

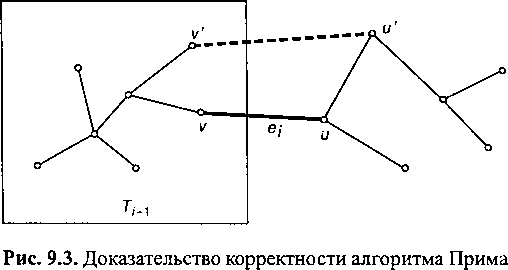
c(b,1) **d(-,oo)** e(a,6) f(b,4)

**d(c,6) e(a,6) f(b,4)**

d(f,5) e(f,2)

d(f,5)

d(f,5)



Следовательно, это остовное дерево является минимальным остовным деревом, что входит в противоречие с предположением, что не существует минимальное остовное дерево, содержащее Т\*. Тем самым доказывается корректность алгорит­ма Прима.

Какова эффективность алгоритма Прима? Ответ зависит от того, какие струк­туры данных выбраны для представления графа и для очереди с приоритетами для вершин из множества V — Vt (приоритетами вершин являются расстояния от них до ближайших вершин в текущем дереве; взгляните еще раз на рис. 9.2 и внима­тельное рассмотрите, как работает очередь с приоритетами в алгоритме Прима). Например, если граф представлен матрицей весов, а очередь с приоритетами реа­лизована при помощи неупорядоченного массива, время работы алгоритма будет равно 0(|У|2). В самом деле, на каждой из \ V\ — 1 итераций требуется полный об­ход массива для поиска и удаления элемента с минимальным расстоянием (и, при необходимости, для последующего обновления приоритетов оставшихся вершин).

Очередь с приоритетами можно реализовать и как неубывающую пирамиду, которая представляет собой зеркальное отражение пирамиды, рассматривавшей­ся в разделе 6.4 (на самом деле ее можно реализовать просто путем изменения знака всех значений ключей). Неубывающая пирамида является полным бинар­ным деревом, в котором каждый элемент не превышает дочерние элементы. Все основные свойства пирамид выполняются и для неубывающих пирамид, с неко­торыми очевидными модификациями. Например, корень неубывающей пирамиды содержит наименьший, а не наибольший элемент. Удаление наименьшего элемен­та и вставка нового элемента в неубывающую пирамиду размером п представляют собой операции, принадлежащие классу эффективности О (logn), как и операция изменения приоритета элемента (см. упражнение 9.1.10).

Если граф представлен с помощью связанного списка смежности, а очередь с приоритетами реализована с использованием неубывающей пирамиды, время работы алгоритма Прима равно О (|£7| log |V^|), поскольку алгоритм выполняет |V| — 1 удалений наименьшего элемента и делает \Е\ проверок (и, возможно,

изменений приоритета) элементов в неубывающей пирамиде размером не более |V|. Каждая из этих операций, как упоминалось ранее, принадлежит классу эф­фективности 0(log|V|). Следовательно, общее время работы алгоритма Прима равно

image184

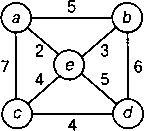
(| V| - 1 + |Я|) О (log IV|) = О (|Е| log IVI),

поскольку в связном графе \V\ — 1 < |я|.

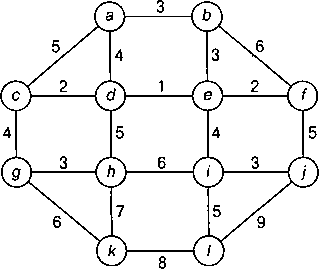
В следующем разделе мы познакомимся с еще одним жадным алгоритмом для решения задачи о минимальном остовном дереве, который работает “жадным” способом, отличным от способа алгоритма Прима.

Упражнения 9.1

1. а) Примените алгоритм Прима к приведенному графу. Включите в оче­редь с приоритетами все вершины, которые не входят в дерево.



б) Примените алгоритм Прима к приведенному графу. Включите в оче­редь с приоритетами только пограничные вершины (которые не вхо­дят в текущее дерево, но каждая из которых смежна по крайней мере одной из вершин дерева).



1. Понятие минимального остовного дерева применимо к связному взве­шенному графу. Требуется ли проверять связность графа перед при­менением к нему алгоритма Прима, или он способен выяснить это самостоятельно?
2. а) Как можно воспользоваться алгоритмом Прима для поиска остов­

ного дерева связного графа без весов его ребер?

б) Насколько этот алгоритм подходит для решения поставленной задачи?

1. Докажите, что любой взвешенный связный граф, у которого веса всех ребер различны, имеет только одно минимальное остовное дерево.
2. Набросайте эффективный алгоритм для изменения значений элементов в неубывающей пирамиде. Какова временная эффективность вашего алгоритма?
3. Алгоритм Крускала

В предыдущем разделе мы рассмотрели жадный алгоритм, который “выра­щивает” минимальное остовное дерево посредством жадного включения в него вершины, ближайшей к вершинам дерева. Интересно, что существует и другой жадный алгоритм, который решает задачу построения минимального остовного дерева и также дает оптимальное решение. Это — алгоритм Крускала (Kruskal’s algorithm) [71], названный так по имени Джозефа Крускала (Joseph Kruskal), ко­торый открыл его, будучи студентом-второкурсником. Алгоритм Крускала ищет минимальное остовное дерево взвешенного связного графа G = (У, Е) как ацик­лический подграф с \V\-l ребрами, сумма весов которых минимальна (нетрудно доказать, что такой подграф должен быть деревом). Следовательно, алгоритм стро­ит минимальное остовное дерево как расширяемую последовательность подгра­фов, которые всегда ацикличны, но на промежуточных стадиях не всегда связны.

Алгоритм начинает с сортировки ребер графа в неубывающем порядке их весов. Затем, начиная с пустого подграфа, просматривает отсортированный список и добавляет очередное ребро в список текущего подграфа, если при этом не создается цикл; в противном случае ребро просто пропускается.

Алгоритм **Kruskal** (**G**)

// Алгоритм Крускала построения минимального остовного // дерева

// Входные данные: Взвешенный связный граф G = (V, Е)

II Выходные данные: Множество ребер Ет, составляющее

// минимальное остовное дерево графа G

Сортировка множества Е в неубывающем порядке весов ребер w (eh) < • ^ tu(ei|E|)

Ет <— 0 Н Инициализация множества ребер дерева

ecounter \*—0 //и его размера

к <— 0 // Инициализация количества обработанных ребер

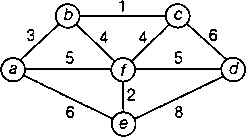
while ecounter < |V| — 1 do к <— к + 1

if Ет U {eik} — ациклический граф

***Ет Ет*** U ***{eik}\ ecounter*** <— ***ecounter*** + 1 return ***Ет***

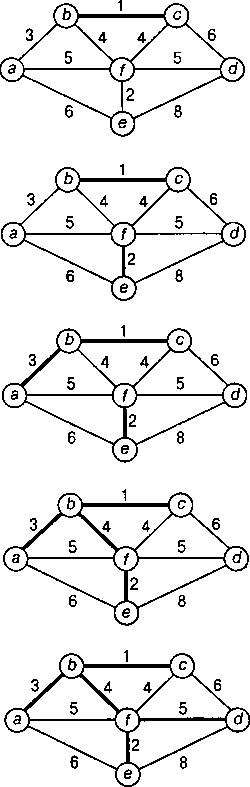
Корректность алгоритма Крускала можно доказать, повторяя основные шаги доказательства алгоритма Прима, приведенного в предыдущем разделе. Тот факт, что Ет в алгоритме Прима является деревом, а в алгоритме Крускала в общем случае — ациклическим подграфом, — трудность, которую несложно преодолеть.

На рис. 9.4 продемонстрировано применение алгоритма Крускала к рассмат­ривавшемуся в разделе 9.1 графу, где к нему был применен алгоритм Прима. При



Рисунок

Вершины дерева Отсортированный список ребер[[57]](#footnote-57)



be ef ad bf cf af df ae cd de 1 234455668

Ьс

1

ef

2

ab

3

bf

**4**

be ef ad bf cf af df ae cd de 1 234455668

be ef ad bf cf af df ae cd de 1 234455668

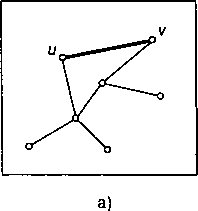
be ef ad bf cf af df ae cd de 1 234455668

be ef ad bf cf af df ae cd de 1 234455668

df

**5**

рассмотрении выполняемых алгоритмом операций обратите внимание на несвяз­ность некоторых промежуточных графов.



Применение алгоритмов Прима и Крускала к одному и тому же небольшому графу вручную может создать ложное ощущение большей простоты алгоритма Крускала. Беда в том, что на каждой итерации алгоритм Крускала должен прове­рить, не приведет ли добавление очередного ребра к появлению цикла. Нетрудно увидеть, что новый цикл образуется тогда и только тогда, когда новое ребро соеди­няет две вершины, уже соединенные некоторым путем, т.е. тогда и только тогда, когда эти две вершины принадлежат одному и тому же связному компоненту (рис. 9.5). Заметим также, что каждый связный компонент подграфа, генерируе­мого алгоритмом Крускала, является деревом, поскольку не содержит циклов.

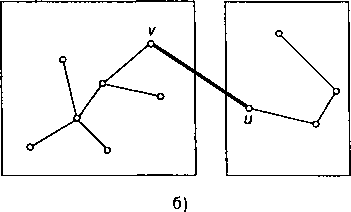


Рис. 9.5. Новое ребро, соединяющее две вершины, может как а) обра­зовать цикл, так и б) не образовывать его

С учетом этого удобнее использовать немного отличную интерпретацию алго­ритма Крускала. Мы можем рассматривать операции алгоритма как продвижение по последовательности лесов, содержащих все вершины исходного графа и только некоторые из его ребер. Изначально лес состоит из |У| тривиальных деревьев, каждое из которых представляет собой отдельную вершину графа. Конечный лес представляет собой единое дерево, являющееся минимальным остовным дере­вом графа. На каждой итерации алгоритм выбирает очередное ребро (и, у) из отсортированного списка ребер графа, находит деревья, содержащие вершины и и v, и, если это различные деревья, объединяет их в одно большее дерево путем добавления ребра (u,v).

К счастью, имеется эффективный алгоритм для выполнения этих действий, включая самое важное — выяснение того, принадлежат ли две вершины одному дереву. Он называется алгоритмом поиска объединений (union-find algorithm) и будет рассмотрен в следующем подразделе. При использовании эффективного алгоритма поиска объединений время работы алгоритма Крускала определяется временем, необходимым для сортировки ребер исходного графа. Следовательно, применяя эффективный алгоритм сортировки, мы найдем, что время работы ал­горитма Крускала равно 0 (\Е\ log \Е\).

**Непересекающиеся подмножества и алгоритмы поиска объединений**

Алгоритм Крускала — один из множества алгоритмов, которым требуется ди­намическое разделение некоторого n-элементного множества S на непересекаю­щиеся подмножества 5i, 5г,..., 5^. После инициализации набора из п одноэле­ментных подмножеств, каждое из которых содержит по одному элементу мно­жества S', этот набор подвергается последовательности операций объединений и поисков (заметим, что количество операций объединения в любой такой после­довательности операций не может превышать п — 1, так как каждое объединение уменьшает количество подмножеств как минимум на 1, а во всем множестве S все­го содержится п элементов). Таким образом, мы имеем дело с абстрактным типом данных, который представляет набор непересекающихся подмножеств некоторого конечного множества и поддерживает следующие операции:

makeset (х) создает одноэлементное множество {я}. Предполагается, что эта операция может быть применена к каждому элементу множе­ства S только один раз;

find (х) находит подмножество, содержащее х\

union (я, у) строит объединение непересекающихся подмножеств Sx и Sy, содержащих, соответственно, ж и у, и добавляет его в набор вместо Sx и Sy, которые из набора удаляются.

Пусть, например, S = {1,2,3,4,5,6}. Тогда таке{г) создает множество {г}, и применение этой операции шесть раз инициализирует набор шестью одноэле­ментными множествами

{1}, {2}, {3}, {4}, {5}, {6}.

Выполнение операций union (1,4) и union (5,2) дает набор

{1,4}, {5,2}, {3}, {6}.

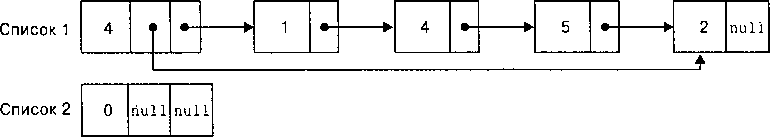
Дальнейшее выполнение union (4,5) и union (3,6) дает непересекающиеся под­множества

{1,4, 5,2} и {3,6}.

Большинство реализаций этого абстрактного типа данных используют по од­ному элементу из каждого из непересекающихся подмножеств набора в качестве представителя (representative). Некоторые реализации не накладывают на пред­ставителей каких-либо особых ограничений; другие требуют, например, чтобы представитель был наименьшим элементом подмножества. Кроме того, обычно считается, что элементы множества представляют собой (или могут быть отобра­жены на) целые числа. Имеется две основные альтернативные реализации этой структуры данных. Первая, называющаяся быстрым поиском (quick find), опти­мизирует временную эффективность операции поиска; вторая — быстрое объеди­нение (quick union), как следует из названия, оптимизирует временную эффектив­ность операции объединения.

Быстрый поиск использует массив, индексированный элементами множества 5; значения массива указывают представителей подмассивов, содержащих соот­ветствующие элементы. Каждое подмножество реализовано как связный список, заголовок которого содержит указатели на первый (first) и последний (last) элементы списка наряду с количеством элементов списка (size) (пример такой реализации показан на рис. 9.6).

size last first



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  |  |  | 3 |  |  | 6 | null |
|  |  |  |  |
|  | |  | |  |  | |  |  |  |

Представители подмножеств

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Список 4 | 0 | null | null | Индекс элемента | Представитель |
|  |  |  |  | 1 | 1 |
|  |  |  |  | 2 | 1 |
| Список 5 | 0 | null | null | 3 | 3 |
|  |  |  |  | 4 | 1 |
|  |  |  |  | 5 | 1 |
| Список 6 | 0 | null | nul Г | 6 | 3 |

Рис. 9.6. Представление подмножеств {1,4,5,2} и {3,6} с использованием связанных списков по методу быстрого поиска. Подмножества получены после выполнения опе­раций гтгоп (1,4), union (5,2), union (4,5) и union (3,6). Списки нулевого размера рассматриваются как удаленные из набора

При использовании такой схемы реализация makeset(x) требует присваи­вания соответствующего элемента в массиве представителей элементу, индекси­рованному значением х, и инициализации соответствующего связанного списка единственным узлом со значением х. Временная эффективность такой операции, очевидно, равна 0(1), следовательно, инициализация п одноэлементных под­множеств требует 0 (п) времени. Эффективность find(x) также равна 0(1): все, что нам надо, — это получить представителя х из массива представителей. Выполнение union (х, у) более продолжительное. Простейшее решение состоит в присоединении связанного списка у в конец связанного списка х, обновлении информации о представителе для всех элементов списка у и удалении списка у из набора. Однако, как легко убедиться, при использовании описанного алгоритма последовательность операций объединения

union (2,1), union (3,2),..., union (г + 1, г),..., union (n, п — 1)

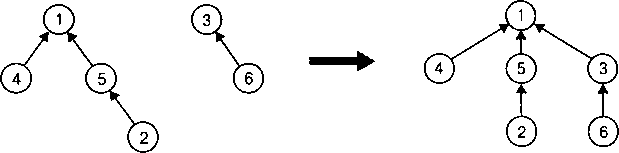
выполняется за время 0 (тг[[58]](#footnote-58)), что слишком медленно по сравнению с некоторыми известными альтернативами.

Простой путь повышения общей эффективности последовательности опера­ций union состоит в том, чтобы всегда добавлять более короткий из двух списков к более длинному, разрешая неоднозначности произвольным образом. Конечно, предполагается, что размер списка доступен (например, при помощи хранения количества элементов в заголовке списка). Такая модификация называется объ­единением по размеру (union by size). Хотя она и не улучшает эффективность единичного применения операции union в наихудшем случае (она остается рав­ной 0(п)), время работы в наихудшем случае произвольной корректной после­довательности операций union-by-size равно О (nlogn).2

Вот доказательство этого утверждения. Пусть а\* — элемент множества S, с подмножествами которого мы работаем, и пусть Ai — количество обновле­ний представителя а\* при выполнении последовательности операций union-by- size. Насколько большим может быть значение Аесли множество S состоит из п элементов? Каждый раз при обновлении представителя а\* элемент а\* дол­жен находиться в меньшем подмножестве, участвующем в объединении, так что получающееся в результате объединение должно содержать как минимум вдвое больше элементов, чем в подмножестве, содержавшем а\*. Следовательно, при первом обновлении представителя а\* получающееся объединение должно состо­ять как минимум из двух элементов; после второго — как минимум из четырех. В общем случае при обновлении А{ раз получающееся в результате множество должно содержать как минимум 2Ai элементов. Поскольку все множество S со­держит п элементов, 2Л\* ^пи, следовательно, Ai ^ log2 п. Таким образом, общее количество возможных обновлений для всех п элементов S не может превышать nlog2 п.

Итак, при использовании операции union-by-size временная эффективность последовательности не более n— 1 объединений и т поисков равна 0(n log n-frn).

Альтернативный метод быстрого объединения представляет каждое подмно­жество в виде корневого дерева. Узлы дерева содержат элементы подмножества, по одному в узле; корневой узел рассматривается как представитель подмноже­ства. Ребра дерева направлены от дочерних узлов к родительским (рис. 9.7). Кроме того, поддерживается не показанное на рисунке из соображений простоты отобра­жение множества элементов на соответствующие им узлы дерева, реализованное, например, при помощи массива указателей.



а)

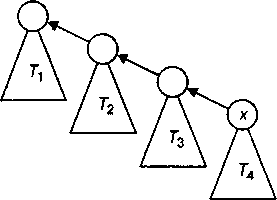
б)

Рис. 9.7. а) Представление методом быстрого объединения подмно­жеств {1,4,5,2} и {3,6} в виде леса.{1,4,5,2} {3,6} б) Результат выполнения операции union (5,6)

При такой реализации операция makeset (х) создает одноузловое дерево за время 0(1); следовательно, инициализация п одноэлементных подмножеств тре­бует 0 (п) времени. Операция union (х, у) реализуется присоединением корня дерева у к корню дерева х (и удалением дерева у из набора, делая указатель на его корень нулевым). Очевидно, что временная эффективность данной операции равна 0 (1). Операция find(x) выполняется путем следования по указателям на родительский узел до корня дерева (элемент которого возвращается в качестве представителя подмножества). Соответственно, временная эффективность одной операции find равна О(п), поскольку дерево, представляющее подмножество, может выродиться в связный список с п узлами.

Эту временную границу можно улучшить. Простейший путь для достижения этого заключается в том, чтобы всегда при выполнении операции union присо­единять меньшее дерево к корню большего дерева, с произвольным разрешением неоднозначностей. Размер дерева можно измерять либо количеством входящих в него узлов (эта версия называется объединением по размеру (union by size)), либо его высотой (эта версия называется объединением по рангу (union by rank)). Конечно, эти варианты требуют хранения для каждого узла дерева либо количе­ства его потомков, либо высоты поддерева, для которого данный узел является корнем. Можно легко доказать, что в любом случае высота дерева будет логариф­мической, что делает возможным выполнение каждого поиска за время О (logn). Таким образом, при быстром объединении временная эффективность последова­тельности не более п — 1 объединений и т поисков равна О (п + mlogn).

В действительности можно получить еще более высокую эффективность, если скомбинировать любой из описанных методов быстрого объединения со сжатием пути (path compression). Эта модификация заставляет каждый узел, встреченный в процессе выполнения операции поиска, указывать на корень дерева (рис. 9.8).



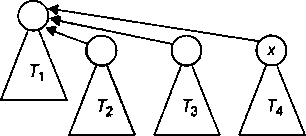
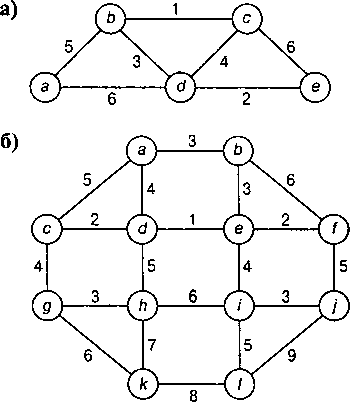


Рис. 9.8. Сжатие пути

Согласно достаточно сложному анализу, который превосходит уровень данной книги (см. [115]), этот и подобные методы повышают эффективность последова­тельности из не более п — 1 объединения и т, поисков до лишь слегка худшей, чем линейная.

Упражнения 9.2

1. Примените алгоритм Крускала для поиска минимальных остовных де­ревьев следующих графов.



1. Истинно или ложно каждое из следующих утверждений?

а) Если е — ребро с минимальным весом в связном взвешенном графе, оно должно быть среди ребер как минимум одного минимального остовного дерева графа.

б) Если е — ребро с минимальным весом в связном взвешенном гра­фе, оно должно быть среди ребер каждого минимального остовного дерева графа.

в) Если веса всех ребер связного взвешенного графа различны, он дол­жен иметь только одно минимальное остовное дерево.

г) Если не все веса ребер связного взвешенного графа различны, он должен иметь больше одного минимального остовного дерева.

1. Какие изменения следует внести в алгоритм Kruskal (если таковые требуются), чтобы он мог находить минимальный остовный лес про­извольного графа? (Минимальный остовный лес — это такой лес, де­ревья которого представляют собой минимальные остовные деревья связных компонентов графа.)
2. Будут ли корректно работать алгоритмы Крускала и Прима с графом, у которого имеются ребра с отрицательными весами?
3. Разработайте алгоритм для поиска максимального остовного дерева — т.е. остовного дерева с наибольшим возможным весом ребер — связного взвешенного графа.
4. Перепишите алгоритм Крускала с применением операций над абстракт­ным типом данных непересекающихся подмножеств.
5. Докажите корректность алгоритма Крускала.
6. Докажите, что временная эффективность операции find (х) в версии union-by-size быстрого объединения равна О (logn).
7. Найдите как минимум два Web-узла с анимацией алгоритмов Крускала и Прима. Обсудите достоинства и недостатки этих анимаций.
8. Разработайте и выполните эксперимент по эмпирическому сравнению эффективностей алгоритмов Прима и Крускала на случайных графах с различными размерами и плотностями.
9. Алгоритм Дейкстры

В этом разделе мы рассмотрим задачу поиска кратчайших путей из одной вершины (single-source shortest-paths problem): для данной вершины взвешенного связного графа, называющейся исходной (source), надо найти кратчайшие пути ко всем остальным вершинам. Важно подчеркнуть, что здесь нас не интересу­ет единый кратчайший путь, начинающийся в исходной вершине и проходящий через все остальные. Это существенно более сложная задача (представляющая собой версию задачи коммивояжера, которая упоминалась в разделе 3.4 и бу­дет рассмотрена позже в этой книге). Задача поиска кратчайших путей из одной вершины требует найти семейство путей, каждый из которых ведет от исходной вершины к другой вершине графа, хотя некоторые пути, конечно же, могут иметь общие ребра.

Множество практических применений задачи поиска кратчайших путей из одной вершины делает ее очень популярным объектом изучения. Имеется ряд широко известных алгоритмов для ее решения, включая алгоритм Флойда для решения более общей задачи поиска кратчайших путей между всеми вершинами, рассматривавшийся в главе 8. Здесь мы рассмотрим наиболее известный алгоритм решения задачи поиска кратчайших путей из одной вершины, который носит имя алгоритм Дейкстры (Dijkstra algorithm).[[59]](#footnote-59) Этот алгоритм применим только к графам с неотрицательными весами. Поскольку в большинстве приложений это условие выполняется, такое ограничение не снижает популярность алгоритма Дейкстры.

Алгоритм Дейкстры находит кратчайшие пути к вершинам графа в поряд­ке их удаления от данной исходной вершины. Сначала он находит кратчайший путь от исходной вершины до ближайшей, затем — до второй ближайшей и т.д. В общем случае перед началом г-ой итерации алгоритм определяет кратчайшие пути к г — 1 другим вершинам, ближайшим к исходной. Эти вершины, исходная вершина и ребра кратчайших путей, ведущих к ним из исходной вершины обра­зуют поддерево Ti данного графа (рис. 9.9). Поскольку веса всех ребер неотри­цательны, очередная ближайшая к исходной вершина может быть найдена среди вершин, смежных с Т\*. Множество вершин, смежных с вершинами в Ti, можно назвать “пограничными”; именно из них алгоритм Дейкстры выбирает очеред­ную вершину, ближайшую к исходной. (В действительности все прочие вершины можно рассматривать как пограничные, соединенные с вершинами дерева ребра­ми с бесконечными весами.) Для определения г-ой ближайшей вершины алгоритм вычисляет для каждой пограничной вершины и сумму расстояния до ближайшей вершины дерева v (определяемого весом ребра (u,v)) и длины dv кратчайшего пути из исходной вершины в вершину v (ранее определенную алгоритмом), а за­тем выбирает вершину с наименьшей суммой. Главным в алгоритме Дейкстры является то, что достаточно сравнить длины таких специальных путей.

Для упрощения работы алгоритма помечаем каждую вершину двумя метками. Числовая метка d указывает длину кратчайшего пути к данной вершине от исход­ной, определенную алгоритмом. Вторая метка указывает имя предыдущей верши­ны на таком пути, т.е. родительский узел строящегося дерева (эта метка остается пустой у исходной вершины и вершин, которые не являются смежными ни с одной

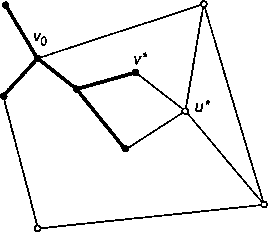


Рис. 9.9. Идея алгоритма Дейкстры. Выде­лено поддерево найденных кратчайших пу­тей. Очередная вершина, ближайшая к ис­ходной г>о, — вершина и\* выбрана путем сравнения длин путей поддерева, увели­ченных на расстояния до вершин, смежных с вершинами поддерева

вершиной текущего дерева). При использовании таких меток поиск следующей ближайшей вершины и\* становится простой задачей поиска пограничной вер­шины с наименьшим значением d. Неоднозначности разрешаются произвольным образом.

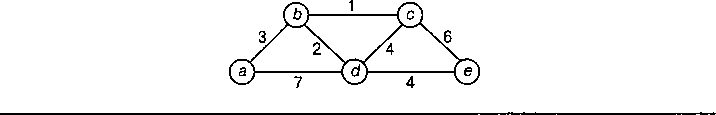
Определив добавляемую в дерево вершину и\*, мы должны выполнить две операции.

* Переместить вершину и\* из множества пограничных во множество вершин дерева.
* Для каждой остающейся вершины и, связанной с вершиной и\* ребром с весом w(u\*,u) такой, что du\* + w(u\*,u) < du, обновить метки и, заменив их на и\* и du\* + w (и\*, и), соответственно.

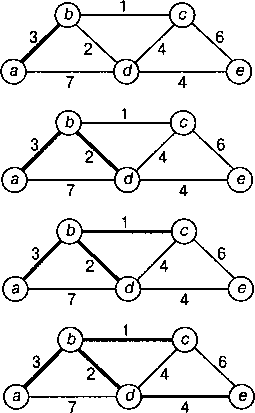
На рис. 9.10 продемонстрировано применение алгоритма Дейкстры к конкрет­ному графу.

Использование меток и механика алгоритма Дейкстры весьма схожи с ис­пользуемыми алгоритмом Прима (см. раздел 9.1). Оба они строят и расширяют поддерево вершин путем выбора очередной вершины из очереди с приоритетами, содержащей остающиеся вне дерева вершины. Однако очень важно не спутать их. Они решают разные задачи и, следовательно, работают с приоритетами, вычис­ляемыми различными способами: алгоритм Дейкстры сравнивает длины путей и должен суммировать веса ребер, в то время как алгоритм Прима сравнивает веса ребер как они есть.

Теперь мы можем привести псевдокод алгоритма Дейкстры. Он описывает работу алгоритма с использованием явного указания операций над двумя мно­жествами вершин с метками: множества Vt вершин, для которых уже найдены



Вершины дерева Остальные вершины Рисунок



а(-,0) Ь(а.З) **с(-,оо)** d(a,7) **е(-.оо)**

b(a,3) c(b,3+4) d(b,3+2) е(-.оо)

d(b,5) c(b,7) e(d,5+4)

c(b,7) e(d,9)

e(d,9)

Кратчайшие пути (определяемые буквенными метками от вершины назначения к исходной в левом столбце) и их длины (числовые метки у вершин дерева):

ОтакЬ: а-b длиной 3

Отакс(: a-b-d длиной 5

Отакс: a-b-с длиной 7

Отаке: a-b-d-e длинойЭ

Рис. 9.10. Применение алгоритма Дейкстры. Очередные ближайшие вершины выделены полужирным шрифтом

кратчайшие пути, и очереди с приоритетами Q, содержащей пограничные верши­ны. (Заметим, что в приведенном псевдокоде Vt содержит заданную исходную вершину, а очередь — смежные с ней вершины после выполнения итерации 0.)

Алгоритм ***Dijkstra*** (G, ***s)***

// Алгоритм Дейкстры для поиска кратчайших путей из одной // вершины

// Входные данные: Взвешенный связный граф G = (V, Е) и его

// вершина s

// Выходные данные: Длина dv кратчайшего пути от s к v

// и предпоследняя вершина pv на этом пути

// для всех вершин v G V

Initialize{Q) // Инициализация пустой очереди

for (Для) каждой вершины veV do

***dv <r- oo\pv <r-*** NULL

Insert(Q, г>, II Инициализация приоритета вершины

11 в очереди с приоритетами

ds 0;

Decrease{Q, s, ds) // Обновление приоритета 5 значением ds

VT <- 0

for i «— 0 to |V| — 1 do

u\* DeleteMin(Q) 11 Удаление элемента с минимальным

II приоритетом

***VT<- VTU{u\*}***

for (Для) каждой вершины и из У — Vt, смежной с u\* do

if du\* + w (и\*, и) < du

***du du\**** + ***w*** (u\*, ***u)\ pu*** <— ***u\****

***Decrease{Q***, ***u,*** du)

Временная эффективность алгоритма Дейкстры зависит от структур данных, используемых для реализации очереди с приоритетами и представления вход­ного графа. По причинам, пояснявшимся при анализе алгоритма Прима в раз­деле 9.1, она равна 0(|У|2) для графов, представленных их весовыми матрица­ми и очередью с приоритетами, реализованной в виде неупорядоченного мас­сива. Для графов, представленных связанными списками смежности и очередью с приоритетами, реализованной как неубывающая пирамида, эффективность рав­на О (\Е\ log |У|). Еще лучшую верхнюю границу можно получить как для ал­горитма Прима, так и для алгоритма Дейкстры, если реализовать очередь с при­оритетами с использованием такой сложной структуры данных, как пирамида Фибоначчи (Fibonacci heap) (см., например, [121]). Однако ее сложность и значи­тельные накладные расходы делают такое усовершенствование представляющим в первую очередь теоретический интерес.

Упражнения 9.3

1. Поясните, какие изменения (если таковые нужны) требуется внести в алгоритм Дейкстры и/или в граф для решения следующих задач.

а) Решение задачи поиска кратчайших путей из одной вершины для ориентированных взвешенных графов.

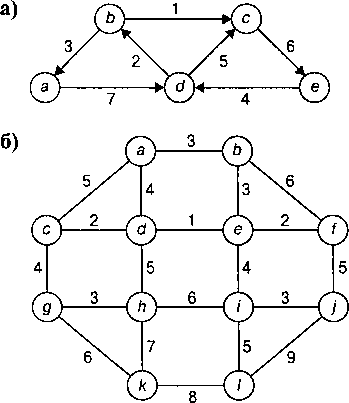
б) Поиск кратчайшего пути между двумя заданными вершинами взве­шенного ориентированного или неориентированного графа {задача

***поиска кратчайшего пути между парой вершин*** (single-pair short- est-path problem)).

в) Поиск кратчайшего пути к данной вершине из всех прочих вер­шин взвешенного ориентированного или неориентированного гра­фа {задача поиска кратчайших путей в одну вершину (single­destination shortest-paths problem)).

г) Решение задачи поиска кратчайших путей из одной вершины для графа с неотрицательными числами, присвоенными его вершинам (длина пути определяется как сумма чисел в вершинах, составляю­щих путь).

1. Решите следующие экземпляры задачи поиска кратчайших путей из одной вершины для вершины а в качестве исходной.



1. Приведите контрпример, который показывает, что алгоритм Дейкст­ры не может корректно работать со взвешенными связными графами с отрицательными весами.
2. Пусть Т — дерево, построенное алгоритмом Дейкстры в процессе ре­шения задачи поиска кратчайших путей из одной вершины для взве­шенного связного графа G. Истинно или ложно каждое из следующих утверждений?

а) Т — остовное дерево G.

б) Г — минимальное остовное дерево G.

1. Напишите псевдокод более простой версии алгоритма Дейкстры, ко­торая находит только расстояния (т.е. длины кратчайших путей, но не

сами эти пути) от данной вершины ко всем остальным вершинам графа, представленного весовой матрицей.

1. Докажите корректность алгоритма Дейкстры для графов с положитель­ными весами.
2. Разработайте алгоритм с линейным временем работы для решения задачи поиска кратчайших путей из одной вершины для ориентиро­ванных ациклических графов, представленных связанными списками смежности.

image200

1. Разработайте эффективный алгоритм для поиска длины самого длин­ного пути в ориентированном ациклическом графе. (Эта задача важна в связи с тем, что она определяет нижнюю границу общего времени, необходимого для завершения проекта, состоящего из заданий с огра­ничениями по предшествованию.)
2. Предположим, что у нас есть модель взвешенного связного графа, сде­ланная из шаров (представляющих вершины), соединенных веревками соответствующих длин (представляющих ребра).

а) Опишите, как можно решить задачу поиска кратчайших путей меж­ду парой вершин при помощи такой модели.

б) Опишите, как можно решить задачу поиска кратчайших путей из одной вершины при помощи такой модели.

1. Вернемся к упражнению 1.3.6 об определении наилучшего маршру­та для пассажира метрополитена, который бы позволял ему переме­щаться от одной станции до другой по развитой системе подземных коммуникаций, наподобие вашингтонской или лондонской. Напишите программу для этой задачи.
2. Деревья Хаффмана

Предположим, требуется закодировать текст, состоящий из символов неко­торого n-символьного алфавита, назначая каждому из символов текста неко­торую последовательность битов, именуемую кодом (codeword). Например, мы можем использовать кодирование фиксированной длины (fixed-length encoding), когда каждому символу назначается битовая строка одной и той же длины т (т ^ log2n). Это именно тот способ, который используется в стандартном се­мибитовом кодировании ASCII. Один из способов получить схему кодирования, которая в среднем дает более короткую битовую строку, основан на старой идее назначать более короткие коды чаще встречающимся символам, а более длинные — тем, что встречаются реже. (Эта идея, в частности, использована в телеграфных кодах, разработанных в средине XIX века Сэмюэлем Морзе (Samuel Morse). В этом

коде часто встречающимся буквам, таким как е (•) или а (• -), назначены короткие

последовательности точек и тире, в то время как редкие буквы, такие как q ( )

и z (— • •), имеют длинные последовательности.)

Использование кодирования переменной длины (variable-length encoding), при котором различным символам назначаются коды разной длины, создает пробле­мы, которых нет при кодировании постоянной длины, а именно: как определить, сколько битов кодированного текста представляют первый (или, в общем случае, г-ый) символ? Для того чтобы избежать этой сложности, мы можем ограничить­ся префиксными кодами (prefix code). При использовании префиксного кода ни один код не является префиксом другого кода. Следовательно, при таком коди­ровании мы просто сканируем битовую строку, пока не получим первую группу битов, являющуюся кодом некоторого символа, заменяем эти биты соответству­ющим символом и повторяем операцию, пока не будет достигнут конец битовой строки.

Если мы хотим создать бинарный префиксный код для некоторого алфавита, естественным представляется связать его символы с листьями бинарного дерева, в котором все левые ребра помечены 0, а правые — 1 (или наоборот). Код символа можно получить путем записи меток ребер на простом пути от корня дерева к листу символа. Поскольку простого пути к листу, который бы продолжался к другому листу, не существует, нет и кода, который бы был префиксом другого кода; таким образом, все такие деревья приводят к префиксному кодированию.

Как среди множества деревьев, которые могут быть построены таким обра­зом для алфавита с известной частотой употребления символов, найти дерево, которое назначает короткие строки часто встречающимся символам и длинные — редко встречающимся? Это можно сделать при помощи следующего жадного ал­горитма, открытого Дэвидом Хаффманом (David Huffman) во время его учебы в Массачуссетском технологическом институте [56].

**Алгоритм Хаффмана**

Шаг 1. Инициализируем п одноузловых деревьев и помечаем их символами ал­фавита. Записываем частоту каждого символа в корне его дерева в ка­честве веса дерева. (В общем случае вес дерева равен сумме частот, указанных в листьях дерева.)

Шаг 2. Повторяем следующую операцию до тех пор, пока не получим единое дерево. Находим два дерева с наименьшими весами (неоднозначности могут быть разрешены произвольным образом, однако см. упражне­ние 9.4.2) и делаем их левым и правым поддеревьями нового дерева, в корне которого записываем сумму их весов в качестве веса образо­ванного дерева.

Дерево, построенное по такому алгоритму, называется деревом Хаффмана (Huffman tree), а код, который оно определяет, — кодом Хаффмана (Huffman code).

Пример 1. Рассмотрим пятисимвольный алфавит {А, В, С, D, \_} со следующими вероятностями символов:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Символ | А | В | с | D |
| Вероятность | 0.35 | 0.1 | 0.2 | 0.2 0.15 |

Построение дерева Хаффмана для приведенных входных данных показано на рис. 9.11. В результате мы получаем следующие коды символов:

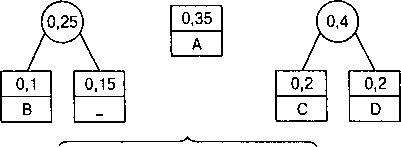
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Символ | А | В | С | D |  |
| Вероятность | 0.35 | 0.1 | 0.2 | 0.2 | 0.15 |
| Код | 11 | 100 | 00 | 01 | 101 |

Следовательно, DAD кодируется как 011101, а 10011011011101 декодируется как BAD\_AD.

Для данных вероятностей символов и полученных кодов математическое ожи­дание количества битов для кодирования одного символа составляет

2 • 0.35 + 3 • 0.1 + 2 • 0.2 + 2 • 0.2 + 3 • 0.15 = 2.25.

Если бы мы применили код фиксированной длины для того же алфавита, нам бы пришлось использовать как минимум три бита для каждого символа. Таким образом, в этом примере применение кода Хаффмана позволило достичь сте­пени сжатия (compression ratio), стандартной меры эффективности алгоритма сжатия информации, равной (3 — 2.25)/3 • 100% = 25%. Другими словами, мы должны ожидать, что кодирование Хаффмана, примененное к некоторому тексту на основе данного алфавита, использует на 25% меньше памяти, чем кодиро­вание фиксированной длины (большое количество экспериментов, выполненных с кодами Хаффмана, показывает, что степень сжатия в данной схеме обычно ока­зывается в диапазоне от 20% до 80%, в зависимости от характеристик сжимаемого файла). ■



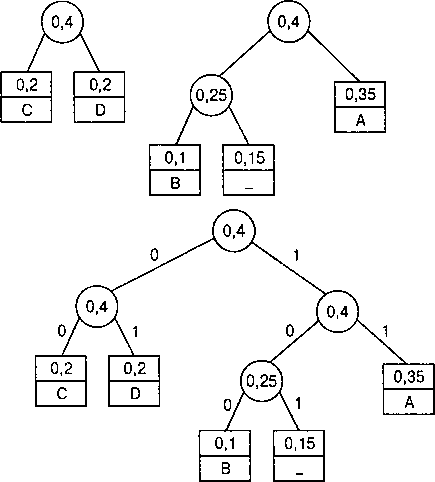
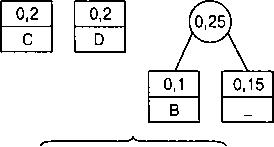


Рис. 9.11. Пример построения дерева кодирова­ния Хаффмана



0,15

0,35

0,1

|  |  |  |
| --- | --- | --- |
| 0,2 |  | 0,2 |
| С |  | D |

0,35

Кодирование Хаффмана — один из наиболее важных методов сжатия файлов. Кроме простоты и универсальности он обладает тем достоинством, что приводит к оптимальному (т.е. минимальной длины) кодированию (при условии, что вероят­ности появления символов независимы и известны заранее). Простейшая версия сжатия Хаффмана в действительности использует предварительное сканирование текста для подсчета частот появления в нем различных символов. Затем на ос­новании этих частот строится дерево Хаффмана, и с его помощью выполняется кодирование текста. Однако такая схема делает необходимым включение в зако­дированный текст информации о дереве кодирования для того, чтобы текст было можно декодировать. Преодолеть этот недостаток можно с помощью так называ­емого динамического кодирования Хаффмана (dynamic Huffman encoding), прикотором дерево кодирования обновляется всякий раз, когда из исходного текста считывается очередной символ (см., например, [101]).

Важно отметить, что алгоритм Хаффмана не ограничивается сжатием дан­ных. Предположим, у нас есть п положительных чисел tui, , wn, которые

назначены п листьям бинарного дерева, по одному на узел. Если мы определим взвешенную длину пути как сумму X^=i кщ, где k — длина простого пути от корня к г-му листу, то как можно построить бинарное дерево с минимальной взве­шенной длиной пути? Это — более общая задача, решаемая алгоритмом Хаффмана (в рассматриваемом случае кодирования и wi представляют собой, соответствен­но, длину кода и частоту появления г-го символа). Эта задача возникает во многих ситуациях, включающих принятие решения. Рассмотрим, например, игру с угады­ванием выбранного предмета из п возможных (например, целого числа от 1 до п) с помощью вопросов, ответы на которые должны быть “да” или “нет”. Различные стратегии, применяемые во время игры, могут быть смоделированы при помощи деревьев принятия решений (decision trees)[[60]](#footnote-60). Пример такого дерева для п = 4 приведен на рис. 9.12.

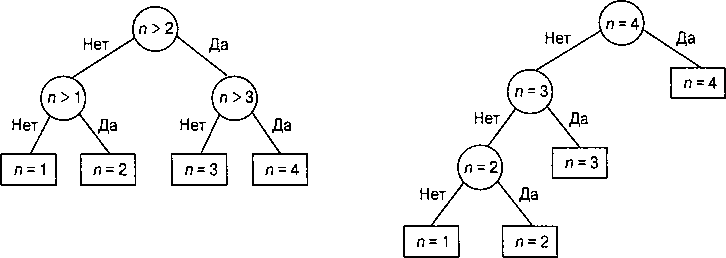


Рис. 9.12. Два дерева принятия решения для угадывания целого числа от 1 до 4

Длина простого пути от корня к листу в таком дереве равна количеству вопро­сов, требующихся для выбора числа, представленного в этом листе. Если число г выбирается с вероятностью р\*, то сумма kPi (гДе — длина пути от корня до г-го листа) определяет среднее количество вопросов, необходимых для “угадыва­ния” выбранного числа при использовании стратегии игры, представленной этим деревом принятия решения. Если каждое из чисел выбирается с одной и той же вероятностью, равной 1/п, то наилучшая стратегия состоит в последовательном исключении половины (или почти половины) кандидатов, как это делается при бинарном поиске. Однако в случае произвольных pi это может быть неподходя­щей стратегией (например, при п = 4, р\ = 0.1, р2 = 0.2, рз = 0.3 и р4 = 0.4 дерево с минимальной взвешенной длиной пути выглядит так, как показано на рис. 9.12 справа). Таким образом, для решения этой задачи в общем виде мы должны воспользоваться алгоритмом Хаффмана.

В заключение стоит вспомнить, что это второй случай, когда мы сталкива­емся с задачей построения оптимального бинарного дерева. В разделе 8.3 мы рассматривали задачу построения оптимального бинарного дерева поиска с поло­жительными числами (вероятностями поиска), назначенными каждому узлу де­

рева. В этом разделе, заданные числа назначаются только листьям. Такая задача оказывается более легкой: ее можно решить при помощи жадного алгоритма, в то время как первая задача решается более сложным алгоритмом динамического программирования.

Упражнения 9.4

1. а) Постройте код Хаффмана для следующих данных:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Символ | А | В | с | D |
| Вероятность | 0.4 | 0.1 | 0.2 | 0.15 0.15 |

б) Закодируйте при помощи полученного кода текст ABACABAD.

в) Декодируйте текст, кодирование которого с применением кода из части а) упражнения дает 100010111001010.

1. При передаче данных зачастую желательно использовать кодирование с минимальной дисперсией длин кодов (среди кодов с одинаковой сред­ней длиной кода). Вычислите среднее значение длины кода и ее диспер­сию для двух разных кодов Хаффмана, получающихся при различном разрешении неоднозначностей в процессе построения дерева Хаффма­на для следующих данных:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Символ | А | В | с | D | Е |
| Вероятность | 0.1 | 0.1 | 0.2. | 0.2 | 0.4 |

1. Скажите, справедливы ли следующие утверждения для любого кода Хаффмана.

а) Коды двух наименее часто встречающихся символов имеют одина­ковую длину.

б) Длина кода более часто встречающегося символа всегда меньше или равна длине кода менее часто встречающегося символа.

1. Какова максимально возможная длина кода при кодировании Хаффмана алфавита из п символов?
2. а) Напишите псевдокод алгоритма построения дерева Хаффмана.

б) Какому классу эффективности как функции размера алфавита п при­надлежит алгоритм построения дерева Хаффмана?

1. Покажите, что дерево Хаффмана можно построить за линейное время, если символы алфавита даны в отсортированном порядке их частот.
2. Какой алгоритм вы используете для получения кодов всех символов для заданного дерева Хаффмана? Какому классу эффективности как функции размера алфавита п принадлежит этот алгоритм?
3. Поясните, как можно генерировать код Хаффмана без явной генерации дерева кодирования Хаффмана.
4. а) Напишите программу, которая строит код Хаффмана для заданного

английского (русского, украинского или иного) текста и кодирует его.

б) Напишите программу для декодирования текста, закодированного с использованием кодов Хаффмана.

в) Поэкспериментируйте с вашей программой и определите типичный диапазон степени сжатия при помощи кодирования Хаффмана обыч­ного текста из примерно 1000 слов.

image205

г) Поэкспериментируйте с вашей программой и определите, насколько изменяется степень сжатия при переходе от использования действи­тельных частот символов в тексте к их стандартным оценкам.

1. Разработайте стратегию для минимизации математического ожидания количества вопросов в следующей игре, взятой в [39]. У вас имеется колода карт, состоящая из одной единицы, двух двоек, трех троек и т.д. до девяти девяток, всего 45 карт. Некто выбирает карту из перетасован­ной колоды. Вы должны определить ее, задавая вопросы, на которые можно отвечать только “да” или “нет”.

Резюме

* Жадный метод состоит в построении решения задачи оптимизации путем последовательности шагов, каждый из которых расширяет ча­стично построенное решение до тех пор, пока не будет достигнуто полное решение исходной задачи. На каждом шаге должен делаться выбор, являющийся допустимым, локально оптимальным и оконча­тельным.
* Алгоритм Прима представляет собой жадный алгоритм для построе­ния минимального остовного дерева взвешенного связного графа. Он работает путем присоединения к ранее построенному поддереву вер­шины, ближайшей к вершинам, уже находящимся в дереве.
* Алгоритм Крускала представляет собой еще один жадный алгоритм для построения минимального остовного дерева взвешенного связного графа. Он строит минимальное остовное дерево путем выбора ребер в возрастающем порядке их весов так, чтобы при этом не образовыва-

лись циклы. Эффективная проверка этого условия требует применения одного из так называемых алгоритмов объединения и поиска.

* Алгоритм Дейкстры решает задачу поиска кратчайших путей из одной (исходной) вершины ко всем другим вершинам взвешенного ориенти­рованного или неориентированного графа. Он работает подобно алго­ритму Прима, но сравнивает не длины ребер, а длины путей. Алгоритм Дейкстры всегда дает корректное решение для графа с неотрицатель­ными весами.
* Дерево Хаффмана представляет собой бинарное дерево, которое ми­нимизирует взвешенную длину пути от корня к листьям, содержащим множество предопределенных весов. Наиболее важным приложением деревьев Хаффмана являются коды Хаффмана.
* Код Хаффмана представляет собой оптимальную префиксную схему кодирования переменной длины, которая назначает символам битовые строки на основе частот появления этих символов в данном тексте. Это достигается путем жадного построения бинарного дерева, листья которого представляют символы алфавита, а ребра помечены нулями и единицами.

**Глава it**

Ограничения мощи алгоритмов

Разум различает возможное и невозможное; рассудок различает осмыс­ленное и бессмысленное. Однако возможное может оказаться бессмыс­ленным.

— Макс Борн (Мах Вот) (1882-1970) “Моя жизнь и взгляды” (My Life and My Views), 1968

В

предыдущих главах книги мы встретились с десятками алгоритмов для реше­ния множества задач. Избежать определения алгоритмов как инструментов для решения задач невозможно: это очень мощные инструменты, в особенности при работе на современных компьютерах. Но мощь алгоритмов не безгранична, и именно эти границы являются предметом рассмотрения в данной главе. Как мы увидим, некоторые задачи не могут быть решены ни одним алгоритмом. Некото­рые могут быть решены, но не за полиномиальное время работы. Но даже если задача может быть решена некоторыми алгоритмами за полиномиальное время, все равно обычно имеются нижние границы эффективности алгоритмов.

В разделе 10.1 мы рассмотрим методы получения нижних границ, т.е. оцен­ки минимально необходимого количества работы для решения задачи. В общем случае получить нетривиальную нижнюю границу, даже для задачи с простой формулировкой, обычно очень непросто. В отличие от определения эффективно­сти конкретного алгоритма, в этом случае нас интересует граница эффективности любого алгоритма — известного или неизвестного. Это также требует очень точно­го и аккуратного описания операций, которые может выполнять такой алгоритм. Если мы ошибемся в точном описании “правил игры”, то место нашим выводам может оказаться на свалке, как это случилось, например, с известным британским физиком лордом Кельвином, который в 1895 году заявил, что летающие машины тяжелее воздуха невозможны...

В разделе 10.2 мы познакомимся с деревьями'принятия решения. Этот метод позволяет кроме прочего, установить нижнюю границу эффективности алгорит­мов для сортировки и поиска в отсортированных массивах, основанных на опе­рации сравнения. В результате мы узнаем, можно ли разработать более быстрый алгоритм сортировки, чем сортировка слиянием, и является ли бинарный поиск самым быстрым алгоритмом поиска в отсортированном массиве (что вам под­сказывает ваша интуиция по поводу этих вопросов?). Кстати, деревья принятия решения оказываются отличными помощниками при решении некоторых голово­ломок, например задачи о поиске фальшивой монеты (см. раздел 5.5).

В разделе 10.3 рассматривается вопрос сложности: какие задачи можно, а ка­кие нельзя решить за полиномиальное время. Эта область теоретической информа­тики называется “теорией вычислительной сложности”. Здесь мы познакомимся только с основными элементами этой теории и неформально рассмотрим такие фундаментальные понятия, как Р, NP и iVP-полнота, включая самую важную нерешенную задачу теоретической информатики о соотношении задач классов Р и NP.

Последний раздел этой главы посвящен численному анализу. Эта область ин­форматики посвящена алгоритмам для решения задач “непрерывной” математи­ки — решения уравнений и систем уравнений, вычисление таких функций, как sin я и 1пх, вычисление интегралов и т.д. Природа этих задач накладывает два типа ограничений. Во-первых, большинство из них не может быть решено точно. Во-вторых, даже приближенное их решение приводит к работе с числами, кото­рые могут быть представлены в цифровых компьютерах только с ограниченным уровнем точности. Работа с приближенными числами без должной аккуратности может привести к очень неточным результатам. Мы увидим, что даже решение обычного квадратного уравнения на компьютере может сопровождаться значи­тельными трудностями и потребовать модификации канонической формулы для корней квадратного уравнения.

1. Доказательства нижних границ

Рассматривать эффективность алгоритмов можно двумя путями. Можно уста­новить класс асимптотической эффективности (скажем, для наихудшего случая) и посмотреть, где он находится в иерархии классов эффективности (см. раз­дел 2.2). Например, сортировка выбором, эффективность которой квадратична, является достаточно быстрым алгоритмом, в то время как алгоритм решения за­дачи о ханойских башнях работает очень медленно в силу своей экспоненциально- сти. Однако можно возразить, что такое сравнение алгоритмов сродни сравнению яблок с вишнями, поскольку эти алгоритмы предназначены для решения совер­шенно разных задач. Альтернативный, более “честный” подход состоит в ответе на вопрос о том, как соотносится эффективность конкретного алгоритма с други­ми алгоритмами для решения той же задачи. В этом случае сортировка выбором оказывается медленным алгоритмом, поскольку имеются алгоритмы сортировки со временем работы О (nlogn); алгоритм решения задачи о ханойской башне, с другой стороны, оказывается наиболее быстрым из возможных алгоритмов для решения поставленной задачи.

Когда мы хотим выяснить, как соотносится эффективность данного алгоритма с эффективностями других алгоритмов для решения той же задачи, желательно знать, какую наивысшую эффективность может иметь любой алгоритм, решающий рассматриваемую задачу. Знание такой нижней границы (lower bound) может под­сказать, на что мы можем надеяться при попытках получить более эффективный алгоритм для решения нашей задачи. Если такая граница плотная (tight), т.е. уже имеется алгоритм, принадлежащий тому же классу эффективности, что и нижняя граница, мы можем в лучшем случае получить только улучшение эффективности на постоянный множитель. Если же между эффективностью наиболее быстрого алгоритма и наилучшей нижней границей имеется “зазор”, то дверь для даль­нейшего возможного усовершенствования алгоритма остается не запертой: либо может существовать более быстрый алгоритм, соответствующий нижней границе, либо можно доказать существование лучшей нижней границы.

В этом разделе представлено несколько методов для определения нижних гра­ниц и проиллюстрировано их применение для конкретных примеров. Как и в слу­чае анализа эффективности конкретных алгоритмов в предыдущих разделах, сле­дует различать класс нижней границы и минимально необходимое количество определенных операций. Как правило, определение второй величины — задача существенно более сложная, чем первая. Например, мы можем сделать немед­ленный вывод о том, что любой алгоритм для поиска медианы п чисел должен иметь эффективность П (п) (почему?), но не так просто доказать, что любой ал­горитм для решения этой задачи, основанный на сравнениях, должен выполнить как минимум 3 (га — 1)/2 сравнений в наихудшем случае (при нечетном п).

**Тривиальные нижние границы**

Простейший метод получения класса нижней границы основан на подсчете количества элементов входных данных, которые следует обработать. Поскольку любой алгоритм должен как минимум “прочесть” все данные, с которыми он будет работать, и “записать” все выходные данные, такой подсчет приводит к триви­альной нижней границе (trivial lower bound). Например, любой алгоритм для ге­нерации всех перестановок п различных элементов должен принадлежать П (п!), поскольку размер выходных данных равен п\. Эта граница — плотная, так как хорошие алгоритмы для генерации перестановок затрачивают постоянное время для поиска каждой из них, за исключением первой (см. раздел 5.4).

В качестве другого примера рассмотрим вычисление полинома степени п

***р (х)*** = ***апхп*** + ***ап-\xn~l*** Н Ь ***а0***

в конкретной точке х для заданных коэффициентов ап, ап\_i,..., ао. Легко видеть, что все коэффициенты должны быть обработаны любым алгоритмом вычисления полинома. Если бы это было не так, то мы бы могли изменить этот необраба­тываемый коэффициент, что должно привести к изменению значения полино­ма в ненулевой точке Таким образом, любой алгоритм вычисления полинома должен иметь эффективность fi (п). Эта нижняя граница — плотная, поскольку имеется алгоритм вычисления справа налево (упражнение 6.5.2) и схема Горнера (раздел 6.5) — оба с линейным временем работы.

Аналогичным способом тривиальная граница для вычисления произведения двух матриц размером пхп оказывается равной fi (п2)> так как любой такой алгоритм должен обработать 2п2 элементов входных матриц и сгенерировать п2 элементов произведения. Однако неизвестно, является ли эта граница плотной.

Тривиальные нижние границы зачастую слишком малы, чтобы быть полез­ными. Например, тривиальная граница для задачи коммивояжера равна Q (п2). поскольку на вход подается п (п — 1)/2 расстояний между городами, а на выходе получается список из п + 1 городов, образующих оптимальный маршрут. Однако эта граница бесполезна, поскольку неизвестен алгоритм решения данной зада­чи, у которого время работы выражалось бы полиномиальной функцией любой степени.

Имеется еще одно препятствие для вывода значимой нижней границы этим методом. Оно связано с определением того, какая часть входных данных должна быть обработана любым алгоритмом решения рассматриваемой задачи. Например, поиск элемента с данным значением в отсортированном массиве не требует обра­ботки всех его элементов (почему?). В качестве еще одного примера рассмотрим задачу определения связности неориентированного графа, заданного его матрицей смежности. Естественно ожидать, что любой такой алгоритм должен проверить существование каждого из п{п — 1)/2 потенциальных ребер, но доказательство этого факта нетривиально.

**Информационно-теоретические доказательства**

В то время как описанный выше подход принимает во внимание размер выхо­да задачи, информационно-теоретический подход пытается установить нижнюю границу алгоритма на основе количества информации, которую он производит. Рассмотрим, например, хорошо известную игру с отгадыванием натурального числа от 1 до п при помощи вопросов, на которые можно давать ответ “да” или “нет”. Количество неопределенностей, которые должен разрешить любой алго­ритм для решения этой задачи, можно оценить как flog2 гг] — количество битов, необходимых для указания конкретного числа из п возможных. Каждый вопрос (вернее, ответы на них) можно рассматривать как получение не более одного бита информации о выходе алгоритма (загаданном числе). Следовательно, любой такой алгоритм должен выполнить как минимум flog2п\ шагов, перед тем как сможет определить загаданное число в наихудшем случае.

Подход, которым мы только что воспользовались, называется информацион­но-теоретическим доказательством (information-theoretic argument) из-за его связи с теорией информации. Как доказано, он очень полезен для поиска так на­зываемых информационно-теоретических нижних границ (information-theoretic lower bounds) для многих задач с использованием сравнений, включая сортировку и поиск. Идея, лежащая в основе этого подхода, может быть гораздо более точ­но осуществлена посредством механизма деревьев принятия решения (decision trees). Из-за важности этого метода он будет рассмотрен отдельно, в разделе 10.2.

**Доказательство “от противника”**

Вернемся еще раз к игре с угадыванием числа, на примере которой мы раз­бирали информационно-теоретический метод. Можно доказать, что любой алго­ритм, который решает эту задачу, должен в наихудшем случае задать как минимум [log2 п\ вопросов, если мы сыграем роль противника, который хочет, чтобы алго­ритм задал максимально возможное число вопросов. Противник начинает с рас­смотрения каждого из потенциально выбираемых чисел от 1 до п. (Конечно, это жульничество, если рассматривать игру, а не доказательство нашего утвержде­ния.) После каждого вопроса противник дает ответ, который оставляет для него наибольшее множество чисел, соответствующих ответам на все ранее заданные вопросы. (Такая стратегия оставляет противнику как минимум половину чисел, которые были у него перед последним вопросом.) Если остановить выполнение алгоритма до того, как размер множества окажется равным единице, противник сможет продемонстрировать число, которое является корректным входным дан­ным и которое алгоритм не смог определить. После этого показать, что требуется |"log2 гг] итераций для приведения п-элементного множества к одному элементу путем снижения его размера вдвое и округления в большую сторону — дело техни­ки. Следовательно, в наихудшем случае алгоритму требуется задать как минимум |~log2n] вопросов.

Этот пример иллюстрирует метод противника (adversary method) для вы­яснения нижних границ. Он основан на логике злого, но честного противника, который делает все, чтобы алгоритм выполнил как можно больше действий, но честность заставляет его согласовывать свои действия с уже сделанным выбором. В таком случае нижняя граница получается путем измерения количества работы, которая требуется для снижения размера потенциального входного множества до одного элемента по самому длинному пути.

В качестве еще одного примера рассмотрим задачу слияния двух отсортиро­ванных списков размером п элементов:

< ^2 < \* \* \* < 0"п и bi < 62 < \* \* • < Ьп

в один отсортированный список размером 2п. Для простоты полагаем, что все а и b различны, так что задача имеет единственное решение. Мы уже встречались с этой задачей при рассмотрении сортировки слиянием в разделе 4.1. Вспомним, что мы выполняли слияние путем многократного сравнения первых элементов в остающихся списках и вывода в результирующий список меньшего из них. Количество сравнений, выполняемых этим алгоритмом в наихудшем случае, равно 2п — 1.

Существует ли алгоритм, который мог бы выполнить слияние быстрее? От­вет на этот вопрос отрицательный. Кнут [67] использовал метод противника для доказательства того, что 2п — 1 — нижняя граница количества сравнений ключей, которые должен сделать любой алгоритм, основанный на сравнениях, для реше­ния поставленной задачи. Противник пользуется следующим правилом: отвечает “да” на вопрос а\* < bj тогда и только тогда, когда г < j. Это заставляет любой корректный алгоритм слияния выдать единственно возможный список, согласую­щийся с этим правилом:

*Ь\ < а\* < Ь2 < *а2 <* • • • < *Ъп < ап.*

Чтобы получить такой список, любой корректный алгоритм должен явным обра­зом сравнить 2п — 1 соседних пар его элементов, т.е. Ь\ — с аь а\ — с и т.д. Если одно из этих сравнений не будет сделано, например, а\ не будет сравнено с Ь2, то мы можем поменять эти ключи местами и получить последовательность

Ь\ < Ъ<1 < а\ < а2 < • • ■ < Ъп < ап,

согласованную со всеми сделанными сравнениями, но неотличимую от коррект­ной последовательности, приведенной выше. Следовательно, 2п — 1 в действи­тельности представляет собой нижнюю границу количества сравнений ключей, необходимых для алгоритма слияния.

Приведение задачи

Мы уже встречались с приведением задачи в разделе 6.6. Там мы рассмотрели метод получения алгоритма для решения задачи Р путем приведения ее к другой задаче Q, разрешимой при помощи известного алгоритма. Подобная идея при­ведения может использоваться и для поиска нижней границы. Чтобы показать, что задача Р как минимум столь же сложна, как и задача Q с известной нижней границей, мы должны привести Q к Р (не Р к Q). Другими словами, мы должны показать, что произвольный экземпляр задачи Q может быть преобразован (доста­точно эффективным способом) к экземпляру задачи Р, так что любой алгоритм, который решает задачу Р, будет также решать и задачу Q. Тогда нижняя граница для задачи Q будет нижней границей и для задачи Р. В табл. 10.1 перечислены несколько важных задач, которые часто используются для этой цели.

|  |  |  |
| --- | --- | --- |
| Задача | Нижняя граница | Плотность |
| Сортировка | П (n log п) | Да |
| Поиск в отсортированном массиве | П (log п) | Да |
| Задача единственности элемента | Q (n log п) | Да |
| Умножение n-значных целых чисел | П (п) | Неизвестно |
| Умножение квадратных матриц | П (п[[61]](#footnote-61)) | Неизвестно |

Мы определим нижние границы для сортировки и поиска в следующем раз­деле. Задача единственности элемента состоит в выяснении того, есть ли среди п заданных чисел дубли (мы уже сталкивались с этой задачей в разделах 2.3 и 6.1). Доказательство нижней границы для этой на вид простой задачи основано на очень сложном математическом анализе, который выходит за рамки нашей кни­ги (см., например, [90]). Что касается двух последних алгебраических задач, то их нижние границы, приведенные в табл. 10.1, тривиальны, но пока неизвестно, могут они быть улучшены или нет.

В качестве примера определения нижней границы методом приведения рас­смотрим задачу поиска евклидова минимального остовного дерева: требуется по­строить дерево минимальной длины, узлами которого являются данные п точек на декартовой плоскости. В качестве задачи с известной нижней границей вос­пользуемся задачей единственности элементов. Мы можем преобразовать любое множество Ж1,Ж2,..., яп из п действительных чисел во множество п точек на декартовой плоскости, просто добавляя к ним 0 в качестве координаты у: (xi,0), (a?2,0),..., (яп, 0). Пусть Т — минимальное остовное дерево, найденное для это­го множества точек. Поскольку Т должно содержать кратчайшее ребро, проверка, содержит ли Т ребро нулевой длины, отвечает на вопрос об единственности дан­ных чисел. Из этого приведения следует, что нижней границей задачи поиска евклидова минимального остовного дерева также является П (nlogn).

Поскольку сложность многих задач неизвестна, метод приведения часто ис­пользуется для сравнения относительной сложности задач. Например, формулы

***(.X* + *у)2 -(х- у)2*** 2

**х • у = И X = X • X**

показывают, что задачи вычисления произведения двух n-разрядных чисел и воз­ведение n-разрядного числа в квадрат принадлежат одному и тому же классу эффективности, несмотря на кажущуюся большую простоту второй задачи по сравнению с первой.

Для операций с матрицами имеется ряд аналогичных результатов. Например, умножение двух симметричных матриц принадлежит тому же классу сложности, что и произведение двух произвольных квадратных матриц. Этот результат осно-

ван на том наблюдении, что не только первая задача является частным случаем второй, но и задача перемножения двух произвольных матриц порядка п, скажем, А и В, может быть приведена к задаче перемножения двух симметричных матриц

0 Вт В 0

0 А Ат 0

***X =***

и У =

***зг***

где и Вт являются транспонированными матрицами А и В (т.е. Afj = А

и Bfj = Bji, а 0 означает нулевую матрицу размером пхп (все элементы которой равны 0). В самом деле,

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ’ 0 | А |  | 'о Вт' |  | АВ 0 |
| Ат | 0 |  | В 0 |  | 0 АТВТ |

XY =

откуда легко получить интересующее нас произведение матриц А В (да, мы два­жды перемножаем матрицы исходного размера, но это всего лишь небольшое техническое усложнение, не влияющее на класс сложности).

Хотя такие результаты и интересны, в разделе 10.3 мы покажем более важное применение метода приведения для сравнения сложности задач.

Упражнения 10.1

1. Докажите, что любой алгоритм поиска наибольшего среди п заданных чисел в худшем случае должен выполнить п — 1 сравнений.
2. Приведите доказательство методом противника того факта, что вре­менная эффективность любого алгоритма, проверяющего связность графа с п вершинами, равна Q (п2), при наличии единственной опера­ции, состоящей в выяснении наличия ребра между двумя вершинами графа. Является ли эта нижняя граница плотной?
3. Чему равно наименьшее число сравнений, необходимых алгоритму сортировки, основанному на операции сравнения, чтобы объединить два отсортированных списка размером пип|1 элементов соответ­ственно? Докажите корректность вашего ответа.
4. Найдите произведение матриц А и В путем преобразования к произ­ведению двух симметричных матриц, если

|  |  |  |
| --- | --- | --- |
| ’l -l" |  | ’ 0 l" |
|  | и В = |  |
| 2 3 |  | -1 2 |

1. а) Можно ли использовать формулы этого раздела, которые указывают

эквивалентность сложности задач умножения и возведения в квад­рат двух целых чисел для того, чтобы показать эквивалентность задач умножения и возведения в квадрат двух матриц?

б) Покажите, что умножение двух матриц порядка п может быть при­ведено к возведению в квадрат матрицы размером 2п.

image206

1. Найдите класс плотной нижней границы задачи поиска двух ближай­ших чисел среди п действительных чисел xi, Х2,..., хп.

10. Дана плитка шоколада, состоящая из п х т ячеек. Ее надо разломить на пт кусочков. Вы можете разламывать плитку только по прямым ли­ниям, и только одну плитку за раз. Разработайте алгоритм для решения поставленной задачи при помощи минимального количества разломов.

1. Деревья принятия решения

Многие важные алгоритмы, в особенности алгоритмы сортировки и поиска, работают посредством сравнения элементов входных данных. Мы можем изучать производительность таких алгоритмов при помощи устройства под названием дерево принятия решения (decision tree). В качестве примера на рис. 10.1 пред­ставлено дерево принятия решения для поиска минимума из трех чисел. Каждый внутренний узел бинарного дерева принятия решения представляет показанное в узле сравнение ключа, например к < к'. Левое поддерево узла содержит инфор­мацию о последующих сравнениях, выполняемых, если к < к'; правое поддеревосодержит аналогичную информацию для случая к > kf (для простоты в этом разделе мы считаем, что все входные элементы различны). Каждый лист пред­ставляет возможный выход алгоритма при обработке некоторых входных данных размера п. Заметим, что количество листьев может оказаться большим, чем ко­личество вариантов выходных данных, поскольку в некоторых алгоритмах один и тот же выход может быть достигнут разными цепочками сравнений (именно такой случай изображен на рис. 10.1). Однако важно отметить, что количество листьев должно быть не меньше возможных выходных данных. Работу алгоритма с конкретными входными данными размером п можно представить как проход по пути по дереву принятия решения от корня до листа; количество сравнений при выполнении алгоритма равно количеству ребер на указанном пути. Следова­тельно, количество сравнений в наихудшем случае равно высоте дерева принятия решения алгоритма.

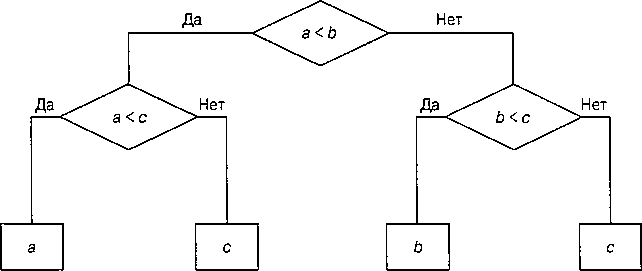


Рис. 10.1. Дерево принятия решения для поиска минимального из трех чисел

Главная идея этого метода заключается в наблюдении, что дерево с данным ко­личеством листьев, которое определяется количеством вариантов выходных дан­ных, должно быть достаточно высоким, чтобы содержать листья в требуемом количестве. В частности, нетрудно доказать, что для любого бинарного дерева с I листьями и высотой h

h> riog2n - (10.1)

В самом деле, бинарное дерево высотой h с наибольшим количеством листьев содержит все листья на последнем уровне (почему?). Следовательно, наибольшее количество листьев в таком дереве равно 2Н. Другими словами, 2Н ^ /, откуда непосредственно следует (10.1).

Неравенство (10.1) определяет нижнюю границу высоты бинарных деревьев принятия решения, а следовательно, количество сравнений, выполняемых в наи­худшем случае любым алгоритмом для решения рассматриваемой задачи, осно­ванным на сравнениях. Такая граница называется информационно-теоретиче­ской нижней границей (см. раздел 10.1). Мы проиллюстрируем этот метод ниже на примере двух важных задач: сортировки и поиска в отсортированном массиве.

Деревья принятия решения для алгоритмов сортировки

Большинство алгоритмов сортировки основано на использовании сравнений, т.е. они работают путем сравнения элементов сортируемого списка. Кроме того, за исключением бинарной сортировки вставкой (см. упражнение 5.1.9), для таких ал­горитмов базовой операцией является сравнение двух элементов. Следовательно, изучая свойства деревьев принятия решения для алгоритмов сортировки, осно­ванных на сравнении, мы можем получить важные нижние границы временной эффективности таких алгоритмов.

Выход алгоритма сортировки можно рассматривать как поиск перестановки индексов элементов входного списка, которая располагает элементы в возрастаю­щем порядке. Например, для выхода а < с < Ь, полученного при сортировке спис­ка а, Ь, с (см. рис. 10.2), интересующая нас перестановка — 1,3,2. Следовательно, количество возможных выходов при сортировке произвольного п-элементного списка равно п\.

***abc***

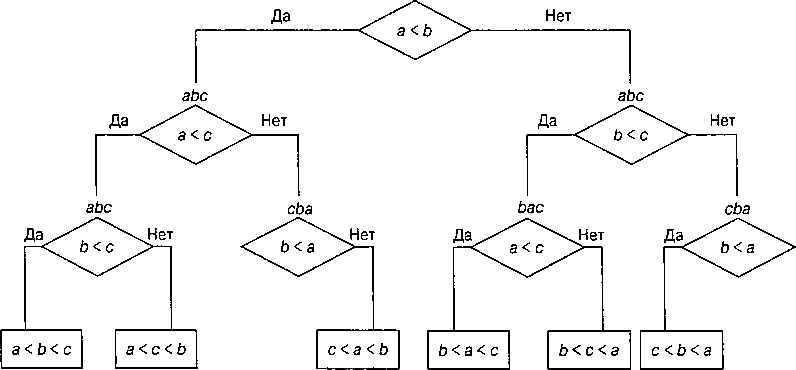


Рис. 10.2. Дерево принятия решения для сортировки выбором трехэлементного списка. Тройка над каждым узлом дерева указывает состояние сортируемого массива. Обратите внимание на два избыточных сравнения b < а с единственным возможным выходом, что связано с выполненными ранее сравнениями

Из неравенства (10.1) следует, что высота бинарного дерева принятия решения для произвольного алгоритма сортировки на основе сравнений (а следовательно, и количество сравнений в наихудшем случае, выполняемое таким алгоритмом) не

может быть меньше |"log2 п!]:

(10.2)

image209

Cworst (п) ^ [log2 п\]. Используя формулу Стирлинга для п!, получим

|"log2 n!] « log2 \/27тп ('п/е)п = п log2 п — п log2 е +

Другими словами, необходимо примерно п log2 п сравнений для сортировки про­извольного п-элементного списка любым алгоритмом сортировки, который осно­ван на использовании сравнений. Заметим, что сортировка слиянием выполняет примерно такое количество сравнений в худшем случае, а следовательно, является асимптотически оптимальной. Отсюда также следует плотность асимптотической нижней границы nlog2n, а значит, она не может быть существенно улучшена. Мы должны отметить, однако, что нижняя граница flog2 п!] может быть улучше­на для некоторых значений п. Например, [log2 12!] = 29, но доказано, что для сортировки массива из 12 элементов в худшем случае необходимо (и достаточно) 30 сравнений.

Мы можем также использовать деревья принятия решения и для анализа пове­дения алгоритма сортировки, использующего сравнения, в среднем случае. Можно вычислить среднее количество сравнений для конкретного алгоритма как сред­нюю глубину листьев его дерева принятия решения, т.е. среднюю длину пути от корня до листьев. Например, для сортировки трех элементов вставкой, де­рево принятия решения которой показано на рис. 10.3, это количество равно (2 + 3 + 3 + 2 + 3 + 3)/6 = 2§.

Стандартное предположение, что все п! выходов алгоритма сортировки рав­новероятны, приводит нас к следующей нижней границе среднего количества сравнений, выполняемых любым алгоритмом сортировки с использованием срав­нений п-элементного списка:

(10.3)

Cavg (n) ^ log2 п!.

Как мы видели ранее, эта нижняя граница равна примерно n log2 п. Вы може­те удивиться, что нижние границы в наихудшем и среднем случае практически идентичны. Вспомним, однако, что эти границы получены путем максимизации количества сравнений, выполняемых соответственно в наихудшем и в среднем случаях. Для отдельных алгоритмов сортировки эффективность в среднем слу­чае, конечно, может значительно превышать эффективность в наихудшем случае.

Деревья принятия решения для поиска в отсортированном массиве

В этом разделе мы покажем, как деревья принятия решения могут использо­ваться для определения нижних границ количества сравнений ключей при поиске

***abc***

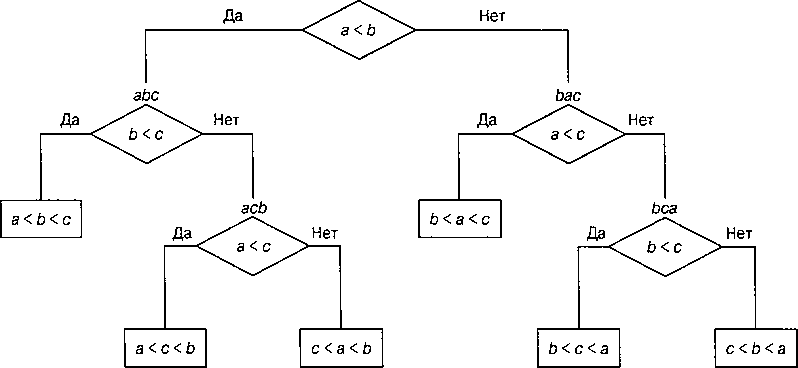


Рис. 10.3. Дерево принятия решения для сортировки трех элементов вставкой

в отсортированном массиве из п ключей А[0] < А[ 1] < ••• < А[п— 1]. Ос­новным алгоритмом для решения этой задачи является бинарный поиск. Как мы видели в разделе 4.3, количество сравнений, выполняемых бинарным поиском в наихудшем случае, C^corsi (п), определяется формулой

Cworst (п) = Uog2 п\ + 1 = Г1о§2 («. + 1)1 . (10.4)

Воспользуемся деревьями принятия решения для того, чтобы выяснить, является ли это количество сравнений наименьшим возможным.

Поскольку здесь мы имеем дело с тернарным сравнением, при котором срав­нение искомого ключа К с некоторым элементом А [г] дает один из ответов — К < А [г\, К = А [г] или К > А [г], естественным представляется применение тернарных деревьев принятия решения. На рис. 10.4 показано такое дерево для случая п = 4. Внутренние узлы этого дерева указывают элементы массива, срав­ниваемые с искомым ключом. Листья указывают либо найденный элемент при успешном поиске, либо найденный интервал, которому принадлежит искомый ключ, при неудачном поиске.

Любой алгоритм для поиска в отсортированном массиве с использованием тернарного сравнения можно представить при помощи тернарного дерева приня­тия решения, подобного приведенному на рис. 10.4. Для массива из п элементов все такие деревья будут иметь 2n + 1 листьев (п для успешных поисков ип+1 для неудачных). Поскольку минимальная высота h тернарного дерева с I листья­ми равна [log3 /], получаем следующую нижнюю границу количества сравнений в наихудшем случае:

**Cworst** ^ |"1°§з (2^ + 1)1 •

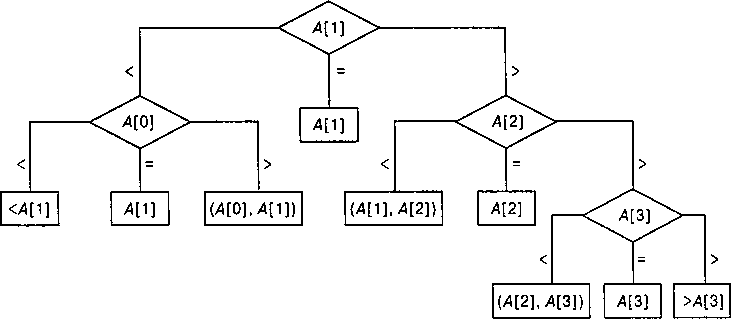


Рис. 10.4. Тернарное дерево принятия решения для бинарного поиска в четырех­элементном массиве

Эта нижняя граница меньше, чем количество сравнений при бинарном поиске в наихудшем случае, равное [log2 (n + 1)] для больших значений п (и не пре­вышает [log2 (n + 1)] для всех натуральных п — см. упражнение 10.2.7). Можем ли мы доказать лучшую нижнюю границу или бинарный поиск оптимален? Для получения лучшей нижней границы мы должны рассмотреть бинарное, а не тер­нарное дерево принятия решения, такое, как показано на рис. 10.5. Внутренние узлы такого дерева соответствуют тем же тернарным сравнениям, что и ранее, но теперь они служат завершающими узлами при успешном поиске. Листья, таким образом, представляют только неудачные поиски, и всего их п + 1 при поиске в n-элементном массиве.

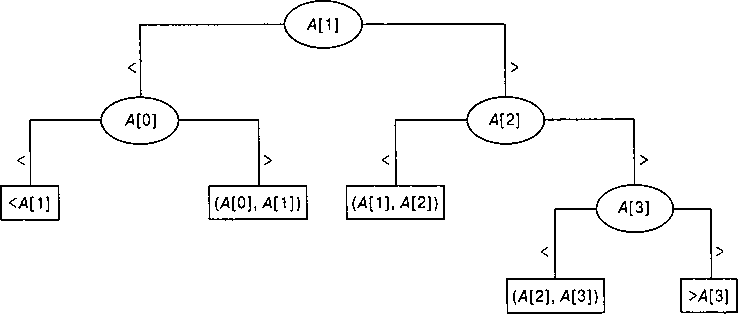


Рис. 10.5. Бинарное дерево принятия решения для бинарного поиска в четырех­элементном массиве

Сравнивая деревья принятия решения на рис. 10.4 и 10.5, мы видим, что бинарное дерево принятия решения — это просто тернарное дерево принятия решения с удаленными средними поддеревьями. Применение неравенства (10.1) к такому бинарному дереву принятия решения дает

CyjOrst (п) ^ [log2 (п + 1)1 . (10.5)

Это неравенство заполняет промежуток между нижней границей и количеством сравнений, выполняемых бинарным поиском в наихудшем случае, которое также равно [log2 (п + 1)]. Существенно более сложный анализ (см., например, [67], раздел 6.2.1) показывает, что при стандартных предположениях о параметрах по­иска бинарный поиск выполняет в среднем наименьшее количество сравнений. Среднее количество сравнений у этого алгоритма равно примерно log2 п — 1 для успешного и log2 (n + 1) для неудачного поисков.

Упражнения 10.2

1. Докажите методом математической индукции, что

а) h ^ flog2 Ц для любого бинарного дерева с высотой h и количеством листьев I.

б) h ^ flog31] для любого тернарного дерева с высотой h и количе­ством листьев I.

1. Рассмотрим задачу поиска медианы трехэлементного множества

{а, ***Ь, с}.***

а) Чему равна информационно-теоретическая нижняя граница алго­ритма для решения этой задачи, основанного на сравнении?

б) Изобразите дерево принятия решения для алгоритма, решающего указанную задачу.

в) Если количество сравнений в наихудшем случае у вашего алгорит­ма больше информационно-теоретической нижней границы, как вы думаете, существует ли алгоритм, соответствующий этой нижней границе или нет? (Либо найдите такой алгоритм, либо докажите невозможность его существования.)

1. Изобразите дерево принятия решения и найдите количество сравнений

ключей в наихудшем и среднем случае для

а) базовой трехэлементной пузырьковой сортировки;

б) улучшенной трехэлементной пузырьковой сортировки (которая пре­кращает работу, если при последнем проходе не было сделано ни одного обмена).

1. Разработайте алгоритм на основе сравнений для сортировки четырех­элементного массива с наименьшим возможным количеством сравне­ний элементов.
2. Разработайте алгоритм на основе сравнений для сортировки пятиэле­ментного массива с наименьшим возможным количеством сравнений элементов.
3. Изобразите бинарное дерево принятия решения для последовательного поиска в четырехэлементном упорядоченном списке.
4. Сравните две нижние границы поиска в отсортированном массиве — flog3 (2п + 1)] и flog2 (п + 1)] — и покажите, что

а) flog3 (2п + 1)] ^ flog2 (гг + 1)] для всех натуральных гг;

image213

б) |~log3 (2гг + 1)] < |~log2 (гг + 1)] для всех натуральных п > щ.

1. Усложненная задача поиска фальшивой монеты. Имеется гг ^ 3 иден­тично выглядящих монет; либо все они подлинные, либо одна из них фальшивая. Кроме того, неизвестно, легче фальшивая монета подлин­ной или тяжелее. У вас есть рычажные весы, при помощи которых вы можете сравнить вес любых двух множеств монет. Т.е. вы можете определить, весит ли некоторое множество монет столько же, боль­ше или меньше другого множества, но не можете сказать, насколько. Задача состоит в том, чтобы определить, все ли монеты подлинные, и если не все, то найти фальшивую и выяснить, легче она подлинной или тяжелее.

а) Докажите, что любой алгоритм для решения этой задачи должен выполнить как минимум [log3 (2п + 1)] взвешиваний в наихудшем случае.

image214

б) Изобразите дерево принятия решения для алгоритма, который ре­шает эту задачу для п = 3 монет за два взвешивания.

1. а) Докажите, что не существует алгоритма, который решает задачу из

упражнения 8 для п = 4 за два взвешивания.

б) Изобразите дерево принятия решения для алгоритма, который ре­шает задачу из упражнения 8 для п = 4 за два взвешивания при помощи дополнительной монеты, о которой достоверно известно, что она подлинная.

в) Изобразите дерево принятия решения для алгоритма, который ре­шает задачу из упражнения 8 для п — 12 за три взвешивания без использования дополнительных монет.

1. Турнирное дерево представляет собой полное бинарное дерево, отра­жающее результаты турнира с выбыванием: его листья представляют п игроков, участвующих в турнире, а каждый внутренний узел — побе-

дителя в игре между игроками, представленными дочерними узлами.

Следовательно, победитель турнира представлен корнем дерева.

а) Чему равно общее количество игр, сыгранных в таком турнире?

б) Сколько всего раундов в таком турнире?

в) Разработайте эффективный алгоритм для определения игрока, ока­зывающегося на втором месте, на основании информации, полу­ченной в ходе проведения турнира. Сколько дополнительных игр требует провести ваш алгоритм?

* 1. Р, NP и iVP-полные задачи

При изучении вычислительной сложности задач первое, на что смотрят и уче­ные, и практики в области информатики, — может ли данная задача быть разре­шена при помощи некоторого алгоритма за полиномиальное время.

Определение 1. Мы говорим, что алгоритм решает задачу за полиномиальное время, если его временная эффективность в наихудшем случае принадлежит клас­су О (р (п)), где р (п) — полином от размера входных данных п. (Обратите вни­мание, что, поскольку мы используем запись с большим “О”, задачи, решаемые, скажем, за логарифмическое время, решаются также и за полиномиальное время.) Задачу, которая может быть решена за полиномиальное время, будем называть легкой (tractable), а задачу, которая не может быть решена за полиномиальное время, — трудной (intractable). ■

Имеется ряд причин для определения трудности задач таким образом. Во- первых, записи в табл. 2.1 и их обсуждение в разделе 2.1 говорят о том, что мы не можем решить произвольные экземпляры трудных задач за реальное время, за исключением только очень малых экземпляров. Во-вторых, хотя время работы может очень сильно отличаться для разных степеней в формуле 0(Р(п)), име­ется очень мало практичных полиномиальных алгоритмов со степенью полинома больше трех. Кроме того, полиномы, ограничивающие время работы алгоритма, обычно не содержат очень больших коэффициентов. В-третьих, полиномиальные функции обладают многими удобными свойствами; в частности, как сумма, так и композиция двух полиномов всегда остаются полиномами. В-четвертых, выбор этого класса привел к разработке теории вычислительной сложности (computa­tional complexity), которая классифицирует задачи в соответствии со сложностью их решения. Согласно этой теории, трудность остается одной и той же для всех основных моделей вычислений и всех разумных схем кодирования входной ин­формации рассматриваемой задачи. В этом разделе мы только слегка затронем основные понятия и идеи теории сложности. Если вас интересует более полное формальное изложение этой теории, вы без труда найдете массу литературы, по­священной этой теме (например, [110, 86]).

Р и ЛГР-задачи

Большинство задач, рассматривавшихся в этой книге, могут быть решены некоторым алгоритмом за полиномиальное время. Сюда входит вычисление про­изведения и наибольшего общего делителя двух целых чисел, сортировка, поиск (ключа в списке или образца в тексте), проверка связности и ацикличности графа, поиск минимального остовного дерева и кратчайших путей во взвешенном графе. (Вы без труда добавите свои примеры в этот список.) Говоря нестрого, задачи, решаемые за полиномиальное время, можно рассматривать как множество, ко­торое ученые-теоретики в области информатики называют Р. Более формальное определение включает в Р только задачи принятия решения (decision problems), представляющие собой задачи, ответ на которые — “да” либо “нет”.

Определение 2. Класс Р представляет собой класс задач принятия решения, ко­торые могут быть решены (детерминистическим) алгоритмом за полиномиальное время. Этот класс задач называется полиномиальным. I

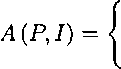
Ограничение класса Р только задачами принятия решения можно объяснить следующими причинами. Во-первых, разумно исключить задачи, не разрешимые за полиномиальное время из-за экспоненциально большого размера выходных данных. Такие задачи возникают естественным путем — например, генерация всех подмножеств данного множества или всех перестановок п различных эле­ментов, — но очевидно, что по этой причине они просто не могут быть решены за полиномиальное время. Во-вторых, многие важные задачи, не являющиеся задача­ми принятия решения в своей естественной формулировке, могут быть приведены к ряду проблем принятия решения, которые проще изучить. Например, вместо вы­яснения, какое наименьшее количество цветов надо для раскраски вершин графа, так, чтобы никакие две смежные вершины не были одного цвета, мы можем вы­яснять, существует ли такая раскраска вершин графа не более чем т цветами

при т — 1,2, (Задача раскраски вершин т цветами называется задачей т-

раскраски (га-coloring problem).) Первое значение т в этом ряду, для которого решение задачи га-раскраски имеет решение, дает ответ на оптимизационную задачу раскраски графа.

Естественно задаться вопросом:, не может ли каждая задача принятия реше­ния быть решена за полиномиальное время? Оказывается, нет. В действительности некоторые задачи принятия решения не могут быть решены вообще, никаким ал­горитмом. Такие задачи называются неразрешимыми (undecidable). Знаменитыйпример такой задачи был приведен Тьюрингом (Turing) в 1936 году.[[62]](#footnote-62) Это знаме­нитая задача останова (halting problem): для данной компьютерной программы и входных данных определить, завершится ли выполнение программы или она будет выполняться бесконечно.

Вот на удивление короткое доказательство этого замечательного факта. От противного предположим, что А — алгоритм, который решает задачу останова, т.е. для любой программы Р и входных данных I



1, если программа Р завершается при входных данных /;

0, если программа Р не завершается при входных данных I.

Мы можем рассматривать программу Р как входные данные для нее самой и ис­пользовать выход алгоритма А для пары (Р, Р) для построения программы Q следующим образом:

завершается, если А (Р, Р) = 0, те\*, если программа Р не q \_ < завершается при входных данных Р;

не завершается, если А (Р, Р) = 1, т.е. если программа Р k завершается при входных данных Р.

Тогда, подставляя вместо Р программу Q, получим

завершается, если A(Q,Q) = 0, т.е. если программа Q не ^ завершается при входных данных Q;

не завершается, если A (Q, Q) = 1, т.е. если программа Q k завершается при входных данных Q.

Мы пришли к противоречию, поскольку ни один из двух выходов программы Q невозможен, — что и завершает доказательство.

Существуют ли задачи разрешимые, но трудные? Да, но количество известных примеров невелико, в особенности тех, которые возникают естественным путем, а не построены в качестве теоретического доказательства.

Однако имеется большое количество важных задач, для которых не найден ал­горитм с полиномиальным временем работы, но и не доказана невозможность его существования. Классическая монография [40] содержит список из несколькихсотен таких задач из различных областей информатики, математики и исследова­ния операций. Вот только несколько из наиболее известных задач, попадающих в эту категорию.

Гамильтонов цикл. Определить, имеется ли в данном графе гамиль­тонов цикл (путь, который начинается и заканчивается в одной и той же вершине и проходит по все остальным вершинам ровно по одному разу).

Задача коммивояжера. Найти кратчайший маршрут по п городам с из­вестными положительными целыми расстояниями между ними (найти кратчайший гамильтонов цикл в полном графе с положительными це­лыми весами).

Задача о рюкзаке. Найти подмножество с наибольшей стоимостью из п предметов с заданными положительными целыми весами и стоимостя­ми, которое может быть помещено в рюкзак с заданной положительной целой емкостью.

Задача о разделении. Даны п положительных целых чисел; требуется определить, можно ли разделить их на два непересекающихся подмно­жества с одинаковыми суммами.

Упаковка корзин. Даны п предметов, размеры которых представляют собой положительные рациональные числа, не превышающие 1. Их надо разместить в наименьшее количество корзин размером 1.

Раскраска графа. Для данного графа найти его хроматическое число (наименьшее количество цветов, которыми можно раскрасить вершины графа так, чтобы никакие две смежные вершины не были окрашены в один и тот же цвет).

Целочисленное линейное программирование. Найти максимальное (или минимальное) значение линейной функции нескольких целочис­ленных переменных при условии выполнения конечного множества ограничений в виде линейных равенств и/или неравенств.

Некоторые из этих задач являются задачами принятия решения и, само собой, являясь таковой, не имеют двойственной задачи принятия решения (как, напри­мер, задача га-раскраски для задачи раскраски графа). Общее у всех этих задач то, что они обладают экспоненциальным (или еще более быстрым) ростом коли­чества вариантов выбора решения с ростом размера задачи п. Заметим, однако, что имеется ряд задач с тем же свойством, но решаемых за полиномиальное вре­мя. Например, задача об эйлеровом цикле, т.е. о существовании цикла, который проходит по всем ребрам данного графа ровно по одному разу, может быть реше­на за время О (п2) путем проверки в дополнение к связности графа, все ли его

вершины имеют четную степень. Этот пример особенно поразителен: интуитив­но совершенно не ожидаешь того, что задача обхода всех ребер по одному разу (эйлеров цикл) настолько проще кажущейся похожей задачи о цикле, обходящем по одному разу все вершины (гамильтонов цикл).

Еще одно общее свойство огромного большинства задач принятия решения заключается в том, что при том, что решение задач может быть вычислительно сложным, проверка предложенного решения обычно достаточно проста и может быть выполнена за полиномиальное время (такие решения можно представить как случайным образом генерируемые кем-то и предлагаемые нам для проверки их корректности). Например, легко проверить, является ли предложенный список вершин гамильтоновым циклом для данного графа с п вершинами. Все, что для этого надо, — убедиться, что список содержит п + 1 вершин графа, что первые п вершин в списке различны, а последняя совпадает с первой, и что каждая пара соседних вершин в списке соединяется ребром. Это наблюдение привело ученых к понятию недетерминистического алгоритма.

Определение 3. Недетерминистическим алгоритмом (nondeterministic algo­rithm) называется двухэтапная процедура, которая получает в качестве входа эк­земпляр / задачи принятия решения и делает следующее.

Недетерминистический этап (“угадывания”): генерируется произвольная стро­ка 5, которая может рассматриваться как кандидат в решение данной задачи I (но может оказаться и полной ерундой).

Детерминистический этап (“проверки”): детерминистический алгоритм полу­чает / и S в качестве входных данных и выдает “да”, если S представляет собой решение экземпляра I. (Если S не является решением /, алгоритм либо возвра­щает “нет”, либо может вообще не завершить работу.) ■

Мы говорим, что недетерминистический алгоритм решает задачу принятия решения тогда и только тогда, когда для каждого экземпляра задачи с ответом “да” он возвращает при некотором выполнении “да” (другими словами, требу­ется, чтобы недетерминистический алгоритм был способен как минимум один раз “угадать” решение, и был способен проверить его корректность. И, конеч­но, мы не хотим, чтобы он мог ответить “да” для экземпляра, ответ для которого должен быть “нет”). Наконец, недетерминистический алгоритм является недетер­министическим полиномиальным (nondeterministic polynomial), если временная эффективность этапа проверки полиномиальная.

Теперь определим класс iVP-задач.

Определение 4. Класс NP — это класс задач принятия решения, которые могут быть решены недетерминистическим полиномиальным алгоритмом. Этот класс задач называется недетерминистическим полиномиальным (nondeterministic polynomial). ■

Большинство задач принятия решения принадлежат классу NP. Прежде всего этот класс включает все задачи класса Р: PC NP, Это соотношение истинно, так как если задача принадлежит классу Р, мы можем использовать детерминисти­ческий полиномиальный алгоритм, который решает ее, на этапе проверки неде­терминистического алгоритма, просто проигнорировав строку 5, генерируемую на этапе угадывания. Но, кроме того, класс NP включает также задачу поиска гамильтонова цикла, задачу о разделении, задачу о рюкзаке, задачу коммивояже­ра и сотни других сложных комбинаторных задач оптимизации, перечисленных в [40]. С другой стороны, задача останова находится среди тех редких задач при­нятия решения, о которых известно, что они не входят в класс NP.

Это приводит к наиболее важному открытому вопросу теоретической инфор­матики: является ли класс Р истинным подмножеством NP или на самом деле оба эти класса совпадают? Мы можем записать этот вопрос как

***P = NP.***

Давайте обсудим эти два варианта. Р = NP должно приводить к тому, что каж­дая из многих сотен сложных комбинаторных задач принятия решения может быть решена алгоритмом с полиномиальным временем работы, хотя ученые не в состоянии найти такие алгоритмы несмотря на многолетние усилия. Кроме того, многие хорошо известные задачи принятия решения являются NP-полными (NP- complete). Говоря нестрого, iVP-полная задача — это задача класса 7VP, которая так же сложна, как и любая другая задача этого класса, поскольку по определению любая другая задача класса NP может быть приведена к ней за полиномиальное время (что схематически показано на рис. 10.6).

Вот более строгое определение этой концепции.

Определение 5. Задача принятия решения D\ называется полиномиально при­водимой (polynomially reducible) к задаче принятия решения D2, если имеется функция £, которая преобразует экземпляры D\ в экземпляры D<i так, что

1. t отображает все экземпляры D\ с положительным ответом на экзем­пляры £>2 с положительным ответом, и все экземпляры D\ с отрица­тельным ответом на экземпляры с отрицательным ответом;
2. t вычислима при помощи алгоритма с полиномиальными временем работы. ■

Из этого определения непосредственно следует, что если задача D\ полино­миально приводима к некоторой задаче Д2, которая может быть решена за поли­номиальное время, то задача D\ также может быть решена за полиномиальное время (почему?).

Л/Р-задачи

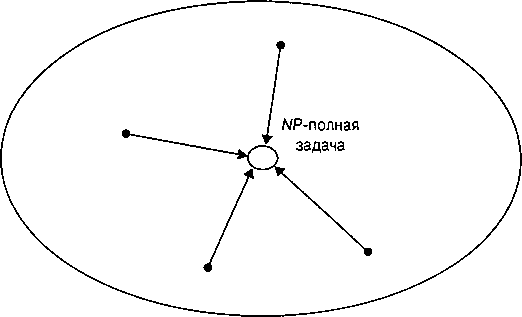


Рис. 10.6. Понятие iVP-сложной задачи. Стрелками пока­зано приведение iVP-задач к iVP-полной задаче за поли­номиальное время

ЛГР-полные задачи

Введем еще одно важное понятие теории вычислительной сложности — NP- полноты.

Определение 6. Задача принятия решения D является NP-полной (iVP-complete), если

1. она принадлежит классу NP;
2. любая задача в NP полиномиально приводима к D. ■

Тот факт, что тесно связанные задачи принятия решения полиномиально при­водимы одна к другой, не слишком удивителен. Например, докажем, что задача поиска гамильтонова цикла полиномиально приводима к версии принятия реше­ния задачи коммивояжера. Последнюю можно сформулировать как задачу опре­деления существования гамильтонова цикла в полном графе с положительными целыми весами, полная длина которых не превышает заданного положительного целого числа га. Мы можем отобразить граф G данного экземпляра задачи о га­мильтоновом цикле на полный взвешенный граф G', представляющий экземпляр задачи коммивояжера, назначая вес, равный 1, каждому ребру из G, и добавляя ребра весом 2 между всеми парами вершин, несмежных в G. В качестве верхней границы га длины гамильтонова цикла возьмем п = га, где п — количество вер­шин в G (и G')- Очевидно, что такое преобразование может быть выполнено за полиномиальное время.

Пусть G — экземпляр задачи о гамильтоновом цикле с положительным от­ветом. Тогда G имеет гамильтонов цикл, и его образ в G' имеет длину п, давая

образ положительного экземпляра версии принятия решения задачи коммивояже­ра. И обратно: если вС'у нас есть гамильтонов цикл длиной не более п, то его длина должна быть в точности п (почему?), а следовательно, цикл должен состо­ять из ребер, имеющихся в G, что дает обратное отображение экземпляра версии принятия решения задачц коммивояжера с положительным ответом на экземпляр задачи о гамильтоновом цикле с положительным ответом. Полнота доказана.

Понятие iVP-полноты требует, однако, полиномиальной приводимости всех задач в NP, как известных, так и неизвестных, к рассматриваемой задаче. При огромном количестве задач принятия решения вызывает изумление тот факт, что были найдены конкретные примеры iVP-полных задач. Тем не менее этот мате­матический подвиг был независимо совершен Стефеном Куком (Stephen Cook) из США и Леонидом Левиным из СССР.[[63]](#footnote-63) В своей статье [30], датированной 1971 го­дом, Кук показал, что так называемая задача CNF-выполнимости является NP- полной. Эта задача работает с булевыми выражениями. Каждое булево выражение может быть представлено в нормальной конъюнктивной форме, такой как следу­ющее выражение, в котором участвуют три булевы переменные х\, х2 и хз и их отрицания, обозначаемые как х\, х2 и хз соответственно:

(xi V х2 V х2) & (xi V х2) & (xi V х2 V Хз).

В задаче CNF-выполнимости спрашивается, можно ли назначить переменным в булевом выражении значения true и false так, чтобы результат вычисления всего выражения был равен true. (Легко видеть, что для приведенной формулы это возможно: если х\ — true, х2 = true и хз = false, все выражение принимает значение true.)

С того времени как Кук и Левин нашли первую iVP-полную задачу, найдены сотни, если не тысячи других примеров iVP-полных задач. В частности, широко известные задачи (или их версии принятия решения), упомянутые выше, — задача о гамильтоновом цикле, задача коммивояжера, разделение множества, упаковка корзин и раскраска графа — все они являются iVP-полными задачами. Известно, однако, что если Р ф NP, то должны существовать АР-задачи, которые не являются ни P-задачами, ни iVP-полными задачами.

Много лет ведущим кандидатом в примеры такой задачи была задача опре­деления того, является ли заданное число простым или составным. В 2002 году профессор Маниндра Агравал (Manindra Agrawal) и его студенты Нирай Каял (Neeraj Kayal) и Нитин Саксена (Nitin Saxena) из Индийского института техно­логий в Капуре объявили об открытии детерминистического полиномиального алгоритма для проверки простоты [3]. Их алгоритм, однако, не решает задачу разложения больших составных чисел на множители, которая представляет собой центральную часть широко используемого метода шифрования, называющегося алгоритмом RSA [96].

Показать, что задача принятия решения является iVP-полной, можно в два этапа. Сначала надо показать, что рассматриваемая задача является iVP-задачей, т.е. что случайным образом сгенерированная строка может быть проверена на пригодность в качестве решения задачи за полиномиальное время. Обычно этот этап весьма прост. Второй этап заключается в том, чтобы показать, что любая за­дача из множества NP приводима к рассматриваемой задаче за полиномиальное время. Благодаря транзитивности полиномиального приведения для выполнения этого этапа достаточно показать, что известная iVP-полная задача может быть приведена к данной за полиномиальное время (см. рис. 10.7). Хотя такое преобра­зование может потребовать немалой изобретательности, это несравнимо проще, чем доказывать существование преобразования для каждой задачи из NP. На­пример, если мы уже знаем, что задача о гамильтоновом цикле является NP- полной, из ее полиномиальной приводимости к версии принятия решения задачи коммивояжера доказывает, что последняя также является iVP-полной задачей (по­сле простой проверки того, что версия принятия решения задачи коммивояжера принадлежит классу NP).

А/Р-задачи

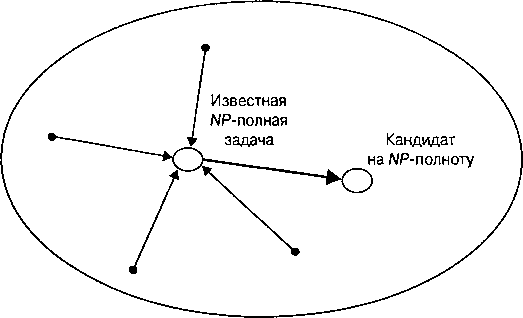


Рис. 10.7. Доказательство iVP-полноты приведением

Непосредственно из определения iVP-полноты следует, что если будет найден детерминистический алгоритм решения одной из iVP-полных задач, то все задачи в NP могут быть решены за полиномиальное время при помощи детерминисти­ческого алгоритма, следовательно, Р = NP. Другими словами, нахождение алго­ритма с полиномиальным временем работы для одной iVP-полной задачи будет

означать, что не существует качественного различия между сложностью проверки предлагаемого решения и поиска его за полиномиальное время для подавляюще­го большинства задач принятия решения всех видов. Такое следствие заставляет большинство ученых верить, что Р ф NP, хотя никто не нашел математического доказательства этой интригующей гипотезы. Интересно, что в интервью с авто­рами книги о жизни и открытиях 15 выдающихся ученых-кибернетиков [106] Кук не смог сказать ничего определенного о решении этой дилеммы, в то время как Левин утверждал, что следует ожидать, что Р = NP.

?

Каким бы ни был окончательный ответ на вопрос Р = NP, на сегодняш­ний день знание того, что задача является iVP-полной, имеет важные практиче­ские следствия. Это означает, что, столкнувшись с NP-полной задачей, не стоит устраивать революцию в информатике, разрабатывая полиномиальный алгоритм для всех ее экземпляров. Вместо этого следует сконцентрироваться на нескольких подходах поиска уменьшения сложности стоящей перед вами задачи. Эти подходы будут вкратце рассмотрены в следующей главе.

Упражнения 10.3

1. Игра в шахматы может рассматриваться как следующая задача приня­тия решения: зная, чей ход, определить для данной корректной позиции шахматных фигур, какая сторона может выиграть. Является ли эта за­дача принятия решения разрешимой?
2. Задача может быть решена при помощи алгоритма со временем работы О (nlog2n). Какие из следующих утверждений истинны?

а) Задача легкая.

б) Задача трудная.

в) Ни одно из приведенных утверждений.

1. Приведите примеры следующих графов или поясните, почему такие примеры не могут существовать:

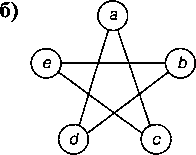
а) графа с гамильтоновым циклом, но без эйлерова цикла;

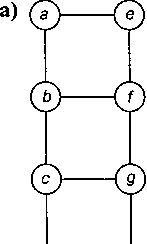
б) графа с эйлеровым циклом, но без гамильтонова цикла;

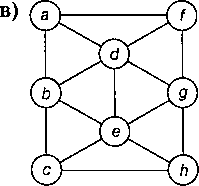
в) графа как с гамильтоновым, так и с эйлеровым циклом;

г) графа с циклом, включающим все вершины, но не являющимся ни гамильтоновым, ни эйлеровым.

1. Найдите хроматическое число каждого из показанных графов.







**© ©**

1. Разработайте алгоритм с полиномиальным временем работы для зада­чи 2-раскраски графа: определить, могут ли вершины данного графа быть раскрашены не более чем двумя цветами так, чтобы никакие две смежные вершины не были окрашены в один и тот же цвет.
2. Рассмотрим следующий алгоритм грубой силы для решения задачи о составных числах: последовательно проверяем все целые числа от 2 до [п/2\ в качестве возможных делителей п. Если на одно из них п делится нацело, возвращаем ответ “да” (т.е. число составное), если не делится ни на одно — возвращаем ответ “нет”. Почему этот алгоритм не делает задачу принадлежащей классу Р?
3. Сформулируйте версию принятия решения для каждой из следующих задач и набросайте полиномиальный алгоритм для проверки, являет­ся ли предложенное решение корректным решением задачи или нет. (Считаем, что гГредлагаемое решение представляет корректные вход­ные данные для вашего алгоритма верификации.)

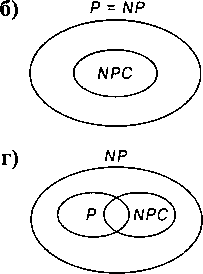
а) Задача о рюкзаке

б) Задача об упаковке корзин

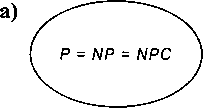
1. Покажите, что задача о разделении полиномиально приводима к версии принятия решения задачи о рюкзаке.
2. Покажите, что следующие три задачи полиномиально приводимы одна к другой.

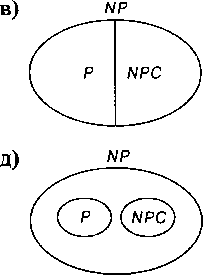
Определить для графа G — (V,E) и положительного целого т ^ \V\, содержит ли граф G клику (clique) размером т и более. (Кликой раз­мером к графа является его полный подграф из к вершин.) Определить для графа G = (V, Е) и положительного целого т ^ \V|, имеется ли вершинное покрытие (vertex cover) графа G размером не более га. (Вершинным покрытием размером к графа G — (V, Е) является подмножество V' С V такое, что \Vf\ = к и для каждого ребра (u, v) € Е как минимум одна из вершин и или v принадлежит V'.)

Определить для графа G = (V, Е) и положительного целого т ^ \V\, содержит ли граф G независимое множество (independent set) разме­ром йе менее га. (Независимым множеством размером к графа G = (У,Е) является подмножество V1 С V такое, что \V'\ = к и для всех u,v €Vf вершины ииу не являются смежными в G.)



1. Какие из приведенных диаграмм не противоречат текущим знаниям о классах сложности Р, NP и NPC (iVP-полных задач)?





Ш

11. Король Артур ожидает на ежегодный обед в Камелоте 150 рыцарей. К сожалению, некоторые из рыцарей в ссоре друг с другом (и король Артур знает, кто с кем). Король хочет рассадить гостей за круглым сто­лом так, чтобы никакие два рыцаря, находящиеся в ссоре друг с другом, не оказались рядом.

а) Какая стандартная задача может использоваться для моделирования задачи короля Артура?

б) В качестве исследовательского проекта найдите доказательство того, что задача короля Артура имеет решение, если каждый рыцарь не находится в ссоре как минимум с 75 другими рыцарями.

* 1. Численные алгоритмы

Численный анализ обычно описывается как раздел информатики, посвящен­ный алгоритмам для решения математических задач. Такое определение требует важного пояснения: рассматриваемые задачи являются задачами “непрерывной” математики — решение уравнений и систем уравнений, вычисление таких функ­ций, как sin х и In я, вычисление интегралов и т.д. — в противоположность задачам дискретной математики, работающим с такими структурами, как графы, деревья, перестановки и сочетания. Наш интерес к эффективным алгоритмам для реше­ния математических задач обусловлен тем, что эти задачи возникают в качестве моделей многих явлений реальной жизни как в природе, так и в обществе. Числен­ный анализ является основной областью исследований, изучения и применения информатики. С проникновением компьютеров в бизнес и повседневную жизнь, где приходится иметь дело в основном с хранением и получением информации, относительная важность численного анализа сокращалась последние 30 лет. Од­нако его приложения, усиленные мощью современных компьютеров, продолжают распространяться во всех областях фундаментальных исследований и техноло­гий. Итак, в какой бы области огромного мира современной информатики не находились ваши интересы, очень важно понимать специфику задач непрерывной математики.

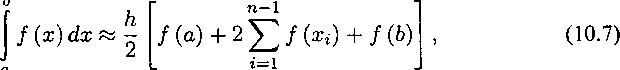
Здесь мы не будем останавливаться на различных трудностях моделирова­ния — задачи описания явлений реального мира математическими формулами. Считая, что это уже сделано, с какими основными препятствиями при решении математических задач нам придется столкнуться? Главное препятствие заклю­чается в том, что большинство задач численного анализа не может быть решено точно.[[64]](#footnote-64) Они должны решаться приближенно, и это обычно делается путем замены бесконечного объекта конечным приближением. Например, значение ех в данной точке х может быть вычислено путем аппроксимации бесконечного ряда Тейлора вблизи х = 0 конечной суммой его первых элементов, называющейся полиномом Тейлора п-ой степени:

:2 Хп

(10.6)

**ж х х в^1 + х + - + --- + -.**

В качестве другого примера определенный интеграл функции может быть аппрок­симирован конечной взвешенной суммой его значений в соответствии с формулой трапеций, которую вы можете помнить с лекций по вычислительной математике:



а

где h= (Ь — а)/п, я\* = а + ih при г = 0,1,..., п (рис. 10.8).

Ошибки таких приближений называются ошибками усечения[[65]](#footnote-65) (truncating ег-

гог). Одной из важных задач в численном анализе является оценка величины

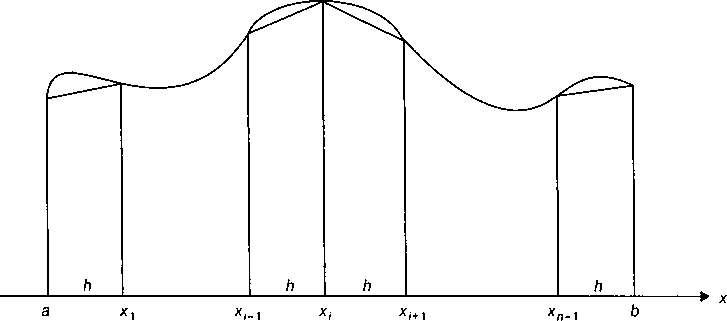


Рис. 10.8. Формула трапеций для вычисления определенного интеграла

ошибок усечения. Обычно это делается с помощью соответствующего математи­ческого инструментария. Например, для приближения (10.6) имеем:

***М***

***х"***

1П+1

е\*-

(10.8)

+

(п + 1)!

га!

*1* + Х+*2*\ +

***х***

где М = max на отрезке с конечными точками 0 и х. Эта формула позволяет определить степень полинома Тейлора, необходимую для гарантирования задан­ной степени точности приближения (10.6).

Например, если мы хотим вычислить е0 5 по формуле (10.6) и гарантировать ошибку усечения не выше 10-4, то можем поступить следующим образом. Во- первых, оценим значение М из формулы (10.8):

М = шах ^ е0'5 < 2.

0^<0.5

Используя эту границу и требуемую точность 10-4, из (10.8) получим

***М***

**п+1**

**-4**

П+1

|0.5|

< 10

<

(п + 1)

(га + 1)!

0.5

Для решения последнего неравенства можно вычислить несколько первых зна­чений

2 \_, 2~п

1. 5n+1 =

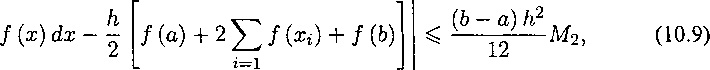
(га + 1)!

(га + 1)!’

что позволит наити наименьшее значение га, для которого выполняется это нера­венство: оно равно 5.

Аналогично, для приближения (10.7) стандартная граница ошибки усечения дается неравенством

ь



п— 1

**а**

где М2 = тах|/"(х)| на интервале а < х ^ Ъ. Вам придется использовать это неравенство в упражнениях к данному разделу.

Второй тип ошибки, именуемой ошибкой округления (round-off error), вызван ограниченной точностью, с которой действительные числа могут быть представ­лены в цифровом компьютере. Эти ошибки возникают не только для всех ирра­циональных чисел (которые по определению требуют бесконечного количества цифр для точного представления), но и для многих рациональных чисел. В аб­солютном большинстве случаев действительные числа представляются как числа с плавающей точкой

(10.10)

zb G?iG?2 • • • dp \* j

где В — основание системы счисления, обычно 2 или 16 (или, для глупых каль­куляторов — 10), di, б?2, • • •, dp — цифры (0 ^ di < В для г = 1, 2,... ,р и d\ > 0, если только само число не равно 0), представляющие дробную часть числа и име­нуемые мантиссой (mantissa); Е — целый показатель степени, или экспонента (exponent) с диапазоном значений, приблизительно симметричным относитель­но 0.

Точность представления числа с плавающей точкой зависит от количества значащих цифр (significant digits) р в представлении (10.10). Большинство ком­пьютеров позволяют использовать два или даже три уровня точности: однократ­ную точность (single precision) (обычно от 6 до 7 значащих десятичных цифр), двойную точность (double precision) (от 13 до 14 значащих десятичных цифр) и расширенную точность (extended precision) (от 19 до 20 значащих десятичных цифр). Использование арифметики с большей степенью точности замедляет вы­числения, но позволяет преодолеть некоторые проблемы, связанные с ошибками округлений. Может потребоваться использовать более высокую точность только на некоторых этапах работы алгоритма.

Как и в случае любого типа приближений, важно различать абсолютную ошибку (absolute error) и относительную ошибку (relative error) представления числа а\* его приближением а:

(10.11)

Абсолютная ошибка = \а — а\*\,

(10.12)

Относительная ошибка =

image227

(Относительная ошибка не определена при а\* = 0.)

Очень большие и очень маленькие числа не могут быть представлены в ариф­метике с плавающей точкой из-за явлений, называющихся соответственно пе­реполнением (overflow) и опустошением, или антипереполнением (underflow). Типичные примеры переполнения возникают при перемножении больших чисел или делении на очень маленькие числа. Иногда таких проблем можно избежать, просто внеся небольшие изменения в порядок вычисления выражения (например, (1029 • II30)/1230 = 1029 • (11/12)30), заменяя выражение эквивалентным (напри­мер, вычисление С200 не как 100!/(2! (100 — 2)!), а как (100 • 99)/2) или вычисляя логарифм выражения вместо его значения.

Опустошение происходит, когда результат операции представляет собой нену­левую дробь со столь малым значением, что она не может быть представлена ненулевым числом с плавающей точкой. Обычно такие числа заменяются нулем, но при этом аппаратным обеспечением генерируется специальный сигнал, указы­вающий, что произошло опустошение.

Важно помнить, что кроме неточностей представления чисел арифметические операции, выполняемые компьютером, также не всегда точны. В частности, вы­читание двух близких чисел с плавающей точкой может вызвать резкий рост относительной ошибки. Это явление носит название потери значащих разрядов (subtractive cancellation).

Пример 1. Рассмотрим два иррациональных числа

а\* = тг = 3.14159265... и /Г = тг - 6 • 10"7 - 3.14159205

представленных числами с плавающей точкой а = 0.3141593\* 101 и /3 = 0.3141592\*

• 101 соответственно. Относительные ошибки этих приближений малы:

[а —а\*1 \_ 0.0000003... 4 7

а\* 7г < 3

И \Р-Р\*\ \_ 0.00000005... 1 7

0\* 7Г — 6 \* 10-7 < 3

соответственно. Относительная ошибка представления разностиу\* = а\* — /3\* раз­ностью представлений числами с плавающей точкой 7 = а — (3 равна

|7 — 7\*| 10"6-6\*10-7 2 7\* “ 6-10-7 “ 3’

что является очень большим значением для относительной ошибки, несмотря на достаточно точные приближения а и (3. ■

Заметим, что можно столкнуться со значительным увеличением ошибки округ­ления, если разность с пониженной точностью будет использована в качестве де­лителя (мы уже сталкивались с этой проблемой при рассмотрении метода исклю­чения Гаусса в разделе 6.2 и воспользовались для ее решения выбором ведущего элемента). Многие численные алгоритмы выполняют для типичных входных дан­ных тысячи или даже миллионы арифметических операций, и распространение ошибок округления в них становится основным вопросом как с теоретической, так и с практической точек зрения. Для некоторых алгоритмов распространение ошибок округления идет с нарастанием. Это крайне нежелательное свойство чис­ленного алгоритма называется нестабильностью (instability). Некоторые задачи демонстрируют настолько высокую степень чувствительности к изменениям вход­ных данных, что невозможно разработать стабильный алгоритм для их решения. Такие задачи называются плохо обусловленными (ill-conditioned).

Пример 2. Рассмотрим следующую систему двух линейных уравнений с двумя неизвестными:

1.001\*+ 0.999?/= 2,

1. 999\* + 1.001?/ = 2.

Ее единственным решением является \* = 1, у — 1. Чтобы увидеть, насколько эта система чувствительна к малым изменениям в правой части, рассмотрим систему линейных уравнений с той же матрицей коэффициентов и слегка измененной правой частью:

1.001\*+ 0.999?/= 2.002,

1. 999\* + 1.001?/ = 1.998.

Единственным решением данной системы является \* = 2 и у = 0, что очень далеко от решения предыдущей системы линейных уравнений. Обратите внима­ние, что матрица коэффициентов близка к сингулярной (почему?). Следовательно, малые изменения коэффициентов этой системы линейных уравнений могут при­вести к отсутствию решения или наличию бесконечного количества решений, в зависимости от коэффициентов в правой части. Более строгое и подробное рассмотрение оценки степени обусловленности матрицы коэффициентов можно найти в соответствующих учебниках (например, [41]). ■

Мы завершим раздел рассмотрением хорошо известной задачи поиска дей­ствительных корней квадратного уравнения

(10.13)

ах2 + Ъх + с = 0

для произвольных действительных коэффициентов а, Ъ и с (а ф 0). В соответствии со школьным учебником по алгебре, уравнение (10.13) имеет действительные корни тогда и только тогда, когда его дискриминант D — Ъ2 — 4ас неотрицателен, и эти корни находятся по формуле

—Ъ ± у/b2 — 4ас \*1,2 = Та • (10.14)

Хотя формула (10.14) дает полное решение поставленной задачи с точки зре­ния математика, это еще не полное решение с точки зрения разработчика ал­горитма. Первое серьезное препятствие заключается в вычислении квадратного корня. Даже для большинства целых D у[Т) является иррациональным числом, которое может быть вычислено только приближенно. Имеется метод вычисления квадратных корней, существенно лучший, чем тот, о котором обычно рассказы­вают в школе (он следует из метода Ньютона, очень важного алгоритма для решения уравнений, который мы рассмотрим в разделе 11.4). Этот метод генери­рует последовательность {хп} приближений y/D, где D — неотрицательное число, в соответствии с формулой

***1 ( D\***

£n+i = g Гхп + — ) при п = 0,1,..., (10.15)

где начальное приближение хо может быть выбрано (среди прочих возможных значений) как хо = (1 + D)/2. Нетрудно доказать, что последовательность (10.15) убывающая (если D ф 1) и всегда сходится в y/D. Мы можем прекратить генера­цию элементов либо когда разность между двумя последовательными элементами станет меньше предопределенной допустимой ошибки е > 0

**Хп** З'п-Ь 1

либо когда станет достаточно близким к D. Последовательность приближе­ний (10.15) очень быстро сходится к у[Т) для большинства значений D. В част­ности, можно доказать, что если 0.25 < D < 1, то требуется не более четырех итераций, чтобы гарантировать, что

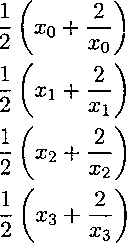
хп - Vd\ < 4 • 10"15,

а мы всегда можем масштабировать данное значение d в интервал [0.25,1) по формуле d = D2P, где р — четное целое число.

Пример 3. Давайте применим метод Ньютона для вычисления у/2 (для просто­ты игнорируем масштабирование). Мы округляем числа до шестого знака после

запятой и используем стандартный символ = для указания округления.

Ж, = J (1 + 2) = 1.500000,



XI = ; = 1.416667,

х2 = = 1.414216,

хз = = 1.414214,

х4 — = 1.414214.

На этом мы остановимся, так как х4 = хз = 1.414214, следовательно, все осталь­ные приближения будут такими же. Точное значение у/2 равно 1.41421356.... ■

Есть ли другие препятствия, кроме вычисления квадратного корня, при ис­пользовании формулы (10.14)? Оказывается, да. Среди прочего мы сталкиваемся

с потерей значащих разрядов. Если Ь2 гораздо больше 4ас, то значение %/Ь2 — 4ас

оказывается очень близким к |Ь|, и корень, вычисляемый по формуле (10.14), может иметь большую относительную ошибку.

Пример 4. Будем следовать статье Джорджа Форсайта (George Forsythe)[[66]](#footnote-66) [38] и рассмотрим уравнение

х2 - 105х + 1 = 0.

Его истинными корнями до одиннадцатой значащей цифры являются

х\ = 99999.999990 их^ 0.000010000000001.

Если мы воспользуемся формулой (10.14) и выполним все вычисления с исполь­зованием десятичной арифметики с плавающей точкой со, скажем, семью знача­щими цифрами, то получим

(~Ъ)2 = 0.1000000 • 10й, 4ас = 0.4000000 • 101, D = 0.1000000 -10й, \/D = 0.1000000 • 10[[67]](#footnote-67),

— 6 -J- \/ D а

**X! =** = 0.1000000 • 106,

2 а

—Ъ — \fD .

***12 =—2^~*** = °'

В то время как относительная ошибка приближения к х\ очень мала, для второго корня она оказывается очень большой:

1\*2 - \*51 \_ ч

5

Z2

т.е. составляет 100%. ■

Чтобы избежать возникновения большой относительной ошибки, можно вос­пользоваться другой формулой, полученной следующим образом:

***—Ъ*** + ***у/Ъ2 —*** 4 ***ас***

***XI =*** —

2 а

*(^—Ь* + *у/b2* — 4*ас^ (^—Ь — у/b2 —* 4*ас^*

2*а (^—Ь — у/Ъ2 — Aacj* 2 с

***—Ь — у/b2 —*** 4 ***ас***

лишенной опасности потери значащих разрядов в знаменателе, если Ъ > 0. Что касается Х2, то его можно вычислить по стандартной формуле

***—Ъ —*** *у/b2* — 4 ***ас*** \*2 = ***2а***

без опасности потери значащих разрядов при положительных значениях Ъ. Случай Ъ < 0 симметричен: мы можем воспользоваться формулами

*—Ь* + *у/Ъ2* — 4 ***ас ХХ =*** ***2а***

И 2 с

*х2 =*

***-Ъ*** + ***\/Ь2 -*** 4***ас***

(Случай 6 = 0 можно отнести к любому из приведенных.)

Имеется ряд иных препятствий при использовании формулы (10.14), которые связаны с ограничениями арифметики с плавающей точкой: если о слишком мало, деление на него может вызвать переполнение; похоже, нет никакого способа из­бежать потери значащих разрядов при вычислении 62 — 4ос, кроме использования вычислений с двойной точностью, и т.д. Все эти проблемы были преодолены

Вильямом Каганом (William Kahan) из университета в Торонто, и его алгоритм рассматривается как значительное достижение в истории численного анализа.

Будем надеяться, что этот краткий обзор пробудил в вас достаточный интерес для поиска дополнительной информации в книгах, посвященных численным ал­горитмам. В следующей главе мы рассмотрим еще одну тему — три классических метода решения уравнений с одним неизвестным.

Упражнения 10.4

1. В некоторых учебниках число значащих цифр в приближении числа а\* числом а определяется как наибольшее неотрицательное целое к, для которого

image229

Сколько, согласно этому определению, значащих цифр в приближении числа 7г значением

а) 3.1415?

б) 3.1417?

1. Известно, что а = 1.5 является приближением некоторого числа а\* с абсолютной ошибкой, не превышающей 10-2. Найдите

а) диапазон возможных значений а\*;

б) диапазон относительных ошибок такого приближения.

1. Найдите приближенное значение /е — 1.648721..., полученное при помощи полинома Тейлора пятой степени около 0, и вычислите ошибку усечения такой аппроксимации. Согласуется ли полученный результат с теоретическим предсказанием, сделанным в тексте раздела?
2. Выведите формулу трапеций (10.7).
3. Воспользуйтесь формулой трапеций при п = 4 для приближенного вы­числения указанных ниже определенных интегралов. Найдите ошибку усечения для каждого приближения и сравните ее с вычисляемой по формуле (10.9).

б) J ***х ldx***

1

1. **J *x2dx*** о
2. На сколько отрезков следует разделить диапазон интегрирования при вычислении J0 esmxdx по формуле трапеций, чтобы ошибка усечения была гарантированно меньше 10-4? меньше 10\_6?
3. Решите две системы линейных уравнений и укажите, являются ли они хорошо или плохо обусловленными.

1 з

а) 2х + 5у = 7 б) 2х + 5у = 7

2х + 5.000001у = 7.000001 2х + 4.999999у = 7.000002

1. Напишите программу для решения квадратного уравнения ах2 + Ъх + + с = 0.
2. а) Докажите, что для любого неотрицательного числа D последова­

тельность, полученная методом Ньютона для вычисления y/D, яв­ляется строго убывающей и сходится к y/D для любого значения начального приближения хо > y/D.

б) Докажите, что если 0.25 ^ D < 1 и х0 = (1 + Z>)/2, то потребуется не более четырех итераций метода Ньютона, чтобы гарантировать, что

Хп-y/D < 4- НГ15.

1. Примените четыре итерации метода Ньютона для вычисления у/3 и оце­ните абсолютную и относительную погрешность полученного прибли­жения.

Резюме

* Для данного класса алгоритмов для решения определенной задачи нижняя граница указывает наилучшую возможную эффективность, ко­торую может иметь любой алгоритм этого класса.
* Тривиальная нижняя граница основана на подсчете количества элемен­тов входных данных задачи, которые должны быть обработаны, и ко­личестве выходных элементов, которые должны быть произведены.
* Информационно-теоретическая нижняя граница обычно получается при помощи механизма деревьев принятия решения. Этот метод осо­бенно полезен для алгоритмов сортировки и поиска, основанных на сравнениях. В частности, любой алгоритм сортировки на основе срав­нений должен выполнить как минимум [log2 n!] ~ п log2 п сравнений ключей в наихудшем случае, а любой алгоритм поиска в отсортиро­ванном массиве на основе сравнений в наихудшем случае должен вы­полнить как минимум [log2 (n + 1)] сравнений ключей.
* Метод противника для установления нижней границы основан на сле­довании логике злонамеренного противника, который направляет алго­ритм по наиболее длинному с точки зрения времени работы пути.
* Нижняя граница может быть установлена при помощи приведения, т.е. путем приведения задачи с известной нижней границей к рассмат­риваемой задаче.
* Теория вычислительной сложности классифицирует задачи в соответ­ствии с их вычислительной сложностью. Принципиальное разделение задач — на легкие и сложные, которые могут и не могут быть решены за полиномиальное время. По техническим причинам теория сложно­сти работает с задачами принятия решения, т.е. задачами, на которые нужно дать ответ да/нет.
* Задача останова представляет собой пример неразрешимой задачи принятия решения, т.е. она не может быть решена никаким алгоритмом.
* Р представляет собой класс всех задач принятия решения, которые могут быть решены за полиномиальное время. NP — это класс задач, включающий все задачи, для которых за полиномиальное время можно проверить корректность предложенного случайным образом решения.
* Многие важные задачи из NP (такие как задача о гамильтоновом цик­ле) являются TVP-полными: все другие задачи из множества NP при­водимы к такой задаче за полиномиальное время. Первое доказатель­ство iVP-полноты было опубликовано С. Куком для задачи о CNF- выполнимости.
* Неизвестно, является ли Р истинным подмножеством NP или Р = NP. Это наиболее важный нерешенный вопрос теоретической информатики. Открытие алгоритма с полиномиальным временем ра­боты для любой из тысяч известных iVP-полных задач докажет, что Р = NP.
* Численный анализ является разделом информатики, посвященным ре­шению задач “непрерывной” математики. При решении таких задач возникают два основных типа ошибок — ошибки усечения (или мето­да), и ошибки округления. Ошибки усечения связаны с заменой бес­конечных объектов их конечными приближениями, а ошибки округле­ния — с неточностями представления чисел в цифровых компьютерах.
* Потеря значащих разрядов происходит вследствие вычитания двух близких чисел с плавающей точкой и может привести к резкому уве­личению относительной ошибки округления, поэтому таких ситуаций следует избегать (либо изменяя вид выражения, либо используя более высокую точность при вычислении такой разности).
* Написание универсальной программы для решения квадратного урав­нения ах2 + Ъх + с = 0 — сложная задача. Вычисление квадратного корня осуществляется при помощи метода Ньютона; избежать потери значащих разрядов можно, используя различные формулы для корней уравнения (в зависимости от того, положительно или отрицательно значение Ъ) и вычисляя дискриминант Ъ2 — 4ас с удвоенной точностью.

**Глава**

Преодоление ограничений

Продолжайте искать новые идеи даже среди успешно использующихся другими. Идея будет оригинальна применением к задаче, над которой вы работаете.

— Томас Эдисон (Thomas Edison) (1847-1931)

К

ак вы узнали из предыдущей главы, многие задачи трудно решить алгорит­мически. В то же время многие из них настолько важны, что мы не можем просто оставить все как есть и ничего не предпринимать. В этой главе будут рассмотрены несколько способов работы с такими сложными задачами.

В разделах 11.1 и 11.2 рассматриваются два метода разработки алгоритмов — поиск с возвратом (backtracking) и метод ветвей и границ (branch-and-bound) — которые зачастую делают возможным решение как минимум некоторых больших экземпляров сложных комбинаторных задач. Обе стратегии можно рассматри­вать как усовершенствование исчерпывающего перебора, рассмотренного в раз­деле 3.4. В отличие от исчерпывающего перебора указанные методы строят кан­дидатов в решения по одному компоненту и оценивают частично построенные решения: если потенциальных значений оставшихся компонентов, которые мог­ли бы привести к корректному решению, не имеется, то оставшиеся компоненты решения не генерируются. Такой подход делает возможным решение некоторых больших экземпляров сложных комбинаторных задач, хотя в наихудшем случае мы все равно сталкиваемся с экспоненциальным ростом, присущим исчерпываю­щему перебору.

Оба метода основаны на построении дерева пространства состояний (state- space-tree), узлы которого отражают конкретные выборы, сделанные для компо­нентов решения. Оба метода останавливаются в узле, если можно гарантировать, что невозможно получить решение, рассматривая решения, соответствующие по­томкам данного узла. Между собой методы отличаются природой задач, к которым они могут быть применены. Метод ветвей и границ применим только к оптими­зационным задачам, поскольку основан на вычислении границ возможных значе­ний целевой функции. Метод поиска с возвратом чаще применим к задачам, не являющимся задачами оптимизации, но не ограничивается ими. Другое отличие между этими методами заключается в порядке генерации узлов дерева простран­ства состояний. При поиске с возвратом это дерево обычно строится в глубину (аналогично поиску в глубину). Метод ветвей и границ может генерировать узлы согласно различным правилам; наиболее естественным является так называемое правило выбора наилуч;шего варианта, о котором рассказывается в разделе 11.2.

В разделе 11.3 мы отходим от идеи точного решения задачи. Представленные здесь алгоритмы решают задачи приближенно, но быстро. В частности, мы рас­смотрим некоторые приближенные алгоритмы для задачи коммивояжера и задачи о рюкзаке. При рассмотрении задачи коммивояжера мы познакомимся с простым жадным алгоритмом и алгоритмом, основанным на тесной связи между минималь­ным остовным деревом и кратчайшим гамильтоновым циклом. При рассмотрении задачи о рюкзаке мы сначала познакомимся с жадным алгоритмом, а затем - с параметрическим семейством полиномиальных алгоритмов, которые позволяют получить сколь угодно хорошие приближения.

Раздел 11.4 посвящен алгоритмам решения нелинейных уравнений. После краткого обсуждения этой очень важной задачи мы рассмотрим три классических метода для поиска приближенных значений корней: метод деления пополам, метод ложной позиции и метод Ньютона.

1. Поиск с возвратом

В книге (в частности, в разделах 3.4 и 10.3) мы уже сталкивались с задача­ми, в которых требовалось найти элемент с некоторыми свойствами из области, растущей экспоненциально (или даже быстрее) с ростом размера входных дан­ных: гамильтонов цикл среди всех перестановок вершин графа, наиболее ценное подмножество предметов в экземпляре задачи о рюкзаке и т.п. Из раздела 10.3 вы узнали, почему многие такие задачи вряд ли могут когда-либо быть решены за полиномиальное время. Вспомним также, что в разделе 3.4 мы выяснили, что такие задачи могут быть решены (по крайней мере в принципе) методом исчер­пывающего перебора. Метод исчерпывающего перебора предполагает генерацию всех решений-кандидатов и выбор из них корректного решения (или решений) с требуемыми свойствами.

Поиск с возвратом представляет собой более интеллектуальный вариант этого подхода. Основная идея состоит в построении решения по одному компоненту и выяснении, может ли дальнейшее построение привести к корректному реше­нию. Если продолжить построение можно без нарушения ограничений задачи, оно продолжается путем выбора первого допустимого варианта для следующего компонента. Если таких вариантов нет, то никакие варианты для всех оставших­ся компонентов не рассматриваются. Алгоритм в этой ситуации возвращается к последнему построенному компоненту и заменяет его следующим допустимым компонентом этого уровня.

Удобно реализовать работу такого алгоритма при помощи построения дере­ва рассмотренных вариантов выбора, которое называется деревом пространства состояний (state-space tree). Его корень представляет начальное состояние перед началом поиска. Узлы на первом уровне дерева представляют варианты выбора, сделанные для первого компонента решения, узлы на втором уровне — варианты выбора второго компонента и т.д. Узел дерева пространства состояний называется обещающим (promising), если он соответствует частично построенному решению, которое может привести к полному решению; в противном случае он называется бесперспективным (unpromising). Листья представляют собой либо бесперспек­тивные тупики, либо полные решения, найденные алгоритмом. В большинстве случаев дерево пространства состояний для алгоритма поиска с возвратом стро­ится способом поиска в глубину. Если текущий узел является обещающим, его до­черний узел генерируется путем добавления первого из остающихся корректных вариантов для следующего компонента решения, и алгоритм переходит к работе с этим дочерним узлом. Если текущий узел бесперспективен, алгоритм возвраща­ется к родительскому узлу и рассматривает очередной возможный вариант для его последнего компонента; если такого варианта нет, алгоритм поднимается вверх по дереву на один уровень и продолжает работу. Наконец, по достижении полного решения задачи, алгоритм либо завершает работу (если требуется только одно решение), либо продолжает поиск прочих возможных решений.

Задача о п ферзях

В качестве первого примера рассмотрим постоянную любимицу авторов учеб­ников — задачу о п ферзях. Задача заключается в размещении п ферзей на шахмат­ной доске размером п х п так, чтобы никакие два ферзя не угрожали друг другу, находясь на одной горизонтали, вертикали или диагонали. Для п — 1 задача име­ет тривиальное решение; легко убедиться, что для п = 2ип = 3 решений не существует. Поэтому начнем с задачи с четырьмя ферзями и решим ее с помощью поиска с возвратом. Поскольку каждый ферзь должен находиться на собственной горизонтали, все, что надо, — это указать вертикаль для каждого ферзя на доске, показанной на рис. 11.1.

Начинаем с пустой доски и помещаем ферзя 1 в первую возможную позицию на его горизонтали; это позиция на вертикали 1. Затем мы размещаем ферзя 2 в первую допустимую позицию (после неудачных попыток размещения на верти­калях 1 и 2) на вертикали 3, в квадрате (2,3). Это тупиковая позиция, поскольку при ней оказывается невозможно разместить третьего ферзя так, чтобы он не был под боем. Соответственно, алгоритм осуществляет возврат к предыдущему состоянию и помещает ферзя в позицию (2,4). После этого третий ферзь может

12 3 4

Ферзь 1 Ферзь 2 Ферзь 3 Ферзь 4

Рис. 11.1. Доска для задачи о че­тырех ферзях

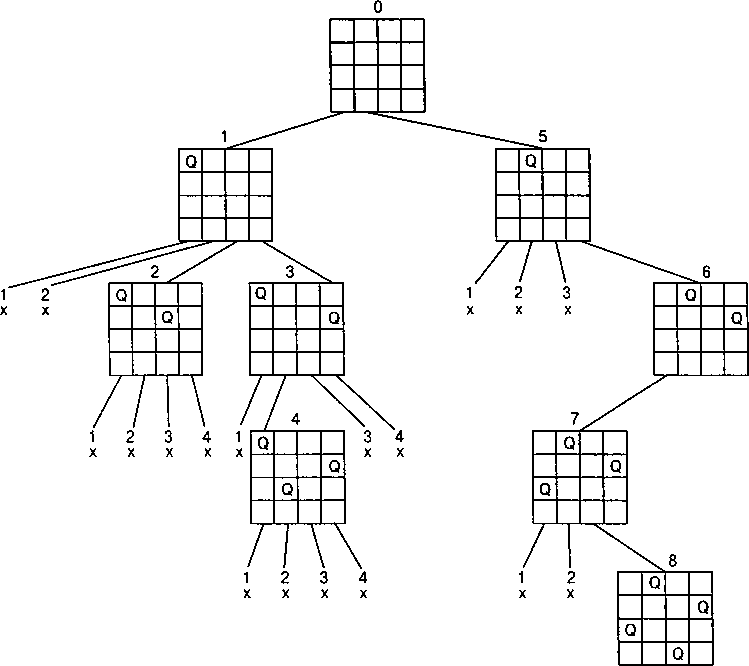
быть помещен только в позицию (3,2), что приводит к очередному тупику. После этого алгоритм осуществляет возврат к ферзю 1 и помещает его в очередной до­пустимой позиции (1, 2). Ферзь 2 может быть размещен только в позиции (2,4), ферзь 3 — только в позиции (3,1), а ферзь 4 — в позиции (4,3), что дает реше­ние поставленной задачи. Соответствующее описанному поиску решения дерево пространства состояний показано на рис. 11.2.

Если требуется найти другие решения поставленной задачи (сколько всего их у задачи о четырех ферзях?), алгоритм может просто продолжить операции с листа, в котором оказалось найдено решение. В качестве альтернативы можно использовать симметрию шахматной доски.

Задача о гамильтоновом цикле

В качестве следующего примера рассмотрим задачу поиска гамильтонова цик­ла графа, показанного на рис. 11.3а.

Без потери общности можно считать, что если гамильтонов цикл существует, то он начинается в вершине а. Соответственно, мы делаем вершину а корнем де­рева пространства состояний (рис. 11.3б). Первый компонент будущего решения, если таковое существует, является первой промежуточной вершиной строящего­ся гамильтонова цикла. Используя алфавитный порядок для разрешения неодно­значности при выборе вершин, смежных с вершиной а, мы выбираем вершину Ь. После вершины Ъ алгоритм обращается к вершинам с, d, е и, наконец, к /, которая является тупиком. Соответственно алгоритм возвращается к вершине с — первой вершине, которая позволяет получить отличное от рассмотренного решение. Пе­реход от с к е не приводит к решению, и алгоритм должен вернуться к вершине Ь. Из вершины b мы попадаем в вершину /, а за ней — в вершины е, с и d, из которой можно вполне законным путем вернуться в а, что дает гамильтонов цикл а, Ь, /, е, с, d, а. Если мы хотим найти еще какой-то гамильтонов цикл, то можем продолжить процесс возврата из листа, соответствующего найденному решению.



Решение

Рис. 11.2. Дерево пространства состояний задачи о четырех ферзях. Крестиками помечены неудачные попытки разместить ферзей на указанных вертикалях. Числа вверху указывают порядок генерации узлов

Задача о сумме подмножества

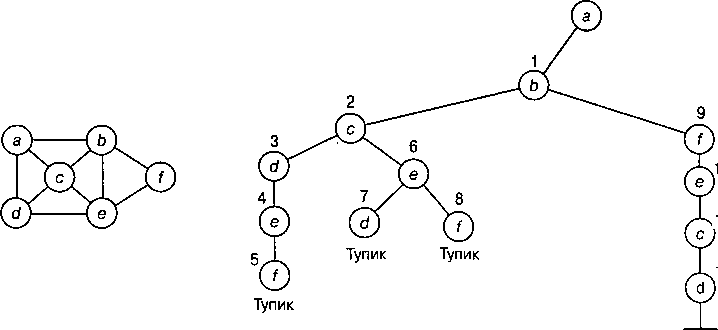
Последним примером применения метода поиска с возвратом будет задача о сумме подмножества (subset-sum problem): требуется найти подмножество данного множества S = {si,..., sn}, состоящего из га натуральных чисел, та­кое, что сумма его элементов равна заданному натуральному числу d. Например, для S = {1,2,5,6,8} и d — 9 имеется два решения — {1,2,6} и {1,8}. Само собой, некоторые экземпляры данной задачи могут не иметь решений.

Удобно отсортировать элементы множества в возрастающем порядке. Будем считать, что

**Sl ^ S2 ^ ^ Sn.**

Дерево пространства состояний может быть построено как бинарное дерево, как показано на рис. 11.4 для экземпляра S = {3,5,6,7} и d = 15. Корень дерева представляет начальную точку, в которой не принято решение ни по одному из элементов множества. Его левый и правый дочерние узлы представляют, соответ­ственно, включение s i в искомое подмножество и отсутствие в нем этого элемента. Аналогично, переход влево от узла на первом уровне означает включение S2 в ис­комое подмножество, а переход вправо — отсутствие этого элемента в искомом множестве. Таким образом, путь от корня к узлу на г-ом уровне дерева указывает, какие из первых г чисел будут включены в подмножество, представленное этим узлом.

о



10

,11

12

а

Решение

а)

б)

Рис. 11.3. а) Граф. б) Дерево пространства состояний\* для поиска гамильтонова

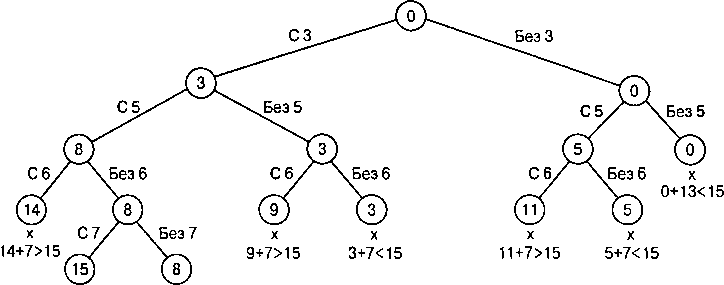
цикла. Числа над узлами указывают порядок генерации узлов

Мы записываем значение s', суммы этих чисел, в узле. Если s' равно с?, мы получаем решение поставленной задачи. Мы можем либо сообщить о нем и пре­кратить работу алгоритма, либо, если надо найти все решения, выполнить оче­редной возврат к родительскому узлу и продолжить работу. Если s' не равно d, мы можем завершить работу с узлом как с бесперспективным при выполнении любого из двух условий:

s' + Si+i > d (сумма s' слишком велика),

*п*

s' + Sj < d (сумма s' слишком мала). .7=1+1



Решение \* \_

8< 15

Рис. 11.4. Полное дерево пространства состояний алгоритма поиска с возвра­том, примененного к экземпляру задачи о сумме подмножества S = {3,5, б, 7} и d = 15. Число внутри узла представляет собой сумму элементов, включенных в подмножество, представленное этим узлом. Неравенства под листьями указы­вают причину прекращения их обработки

Общие замечания

Большинство алгоритмов поиска с возвратом удовлетворяют следующему опи­санию. Выход алгоритма поиска с возвратом можно рассматривать как кортеж из п элементов (х\, Х2, •. •, хп), где каждый элемент х\* является элементом некоторо­го конечного линейного упорядоченного множества Si. Например, для задачи о п ферзях каждое Si представляет собой множество целых чисел (номер вертикали) от 1 до п. Может потребоваться, чтобы кортеж удовлетворял некоторым допол­нительным ограничениям (например, чтобы ни один ферзь не угрожал другому в задаче о п ферзях). В зависимости от задачи все кортежи-решения могут быть одной длины (как, например, в задачах о п ферзях или о гамильтоновом цикле) или иметь разные размеры (как в задаче о сумме подмножества). Алгоритм поиска с возвратом генерирует, явно или неявно, дерево пространства состояний; его узлы представляют частично построенные кортежи с первыми г элементами, опреде­ленными предыдущими действиями алгоритма. Если такой кортеж (xi, Х2, . • •, Xi) не является решением, алгоритм ищет следующий элемент в Si+ь который со­гласуется со значениями из (xi, Х2,..., х\*) и ограничениями задачи, и добавляет его в кортеж в качестве (г + 1)-го элемента. Если такой элемент не существует, алгоритм возвращается к рассмотрению следующего значения Xi, и т.д.

Для запуска алгоритма поиска с возвратом приведенный псевдокод должен быть вызван для г = 0; X [1..0] представляет пустой кортеж.

Алгоритм Backtrack (X [1 ..г])

// Шаблон обобщенного алгоритма поиска с возвратом // Входные данные: Массив Х[1.л\, определяющий первые г

II обещающих компонентов решения

// Выходные данные: Все кортежи, представляющие решения // задачи

if Х[1..г\ является решением write Х[1.л] else // См. упражнение 8

for (для) каждого элемента х е 5г+ь согласующегося с Х[1.л] и удовлетворяющего ограничениям do Х[г + 1] <— х Backtrack(X[l..i + 1])

Успешность решения небольших экземпляров трех различных задач ранее в этом разделе не должна ввести вас в заблуждение и заставить сделать неверный вывод о том, что поиск с возвратом — очень эффективный метод. В наихудшем случае ему может потребоваться сгенерировать всех возможных кандидатов из экспоненциально (или еще быстрее) растущего пространства состояний решаемой задачи. Надежда остается на то, что алгоритм поиска с возвратом будет способен обрезать достаточное количество ветвей дерева пространства решений до того, как исчерпает отведенное ему время или память (или и то, и другое). Успешность этой стратегии колеблется в очень широких пределах, и не только от задачи к задаче, но и между экземплярами одной и той же задачи.

Имеется несколько хитростей, которые помогают снизить размер дерева про­странства состояний. Одна из них — воспользоваться часто присущей комбина­торным задачам симметрией. Например, доска из задачи о п ферзях обладает несколькими симметриями, так что некоторые решения могут быть получены из других с помощью отражений или поворотов. Отсюда, в частности, следует, что нам не надо рассматривать размещение первого ферзя на последних \п/2J верти­калях, поскольку любое решение с первым ферзем в клетке (1, г), [гг/2] ^ г ^ гг, может быть получено путем отражения (какого?) из решения, в котором первый ферзь находится в клетке (1,п — г + 1). Это наблюдение снижает размер дерева примерно наполовину. Второй трюк заключается в предварительном назначении значений одному или нескольким компонентам решения, как было сделано в при­мере с гамильтоновым циклом. Предварительная сортировка в примере задачи о сумме подмножества продемонстрировала потенциальные достоинства еще од­ной возможности — перегруппировки данных экземпляра задачи.

Было бы крайне желательно оценить размер дерева пространства состояний алгоритма поиска с возвратом. Однако, как правило, это слишком сложная для ана­литического решения задача. Кнут [62] предложил генерировать случайный путь от корня к листу и использовать информацию о количестве выборов, доступных в процессе генерации пути, для оценки размера дерева. В частности, пусть с\ — количество значений первого компонента х\9 которые согласуются с ограничени­ями задачи. Мы случайным образом выбираем одно из этих значений (с равной вероятностью 1/ci) для перехода к одному из с\ дочерних по отношению к корню узлов. Повторяя эту операцию для С2 возможных значений £2, которые согла­суются с х\ и другими ограничениями, мы переходим в один из С2 возможных дочерних по отношению к рассматриваемому узлов. Этот процесс продолжается до тех пор, пока после случайного выбора значений для xi, Х2, • • •, хп не будет достигнут лист. Считая, что узлы на уровне г имеют в среднем по ^ дочерних узлов, общее количество узлов в дереве можно оценить как

**1 + С\ + С**1**С**2 **+ ■ ' • + С**1**С**2 **• • • On.**

Генерируя несколько таких оценок и вычисляя их среднее, можно получить полез­ную информацию о реальном размере дерева, хотя стандартное отклонение этой случайной величины может быть достаточно большим.

В заключение следует сделать три замечания по поводу поиска с возвратом. Во-первых, этот метод обычно он применяется для сложных комбинаторных за­дач, для точного решения которых, возможно, не существует эффективных алго­ритмов. Во-вторых, в отличие от исчерпывающего поиска, который чрезвычайно медленно работает для всех экземпляров задачи, поиск с возвратом как минимум оставляет надежду на то, что решение некоторых экземпляров нетривиальных раз­меров будет найдено за приемлемое время. Это в особенности справедливо для оптимизационных задач, где идея поиска с возвратом может получить дальней­шее развитие при помощи вычисления качества частично построенных решений. Как именно это делается, рассказано в следующем разделе. И, в-третьих, даже если поиск с возвратом не удаляет ни одного элемента из пространства состояний задачи и в результате генерирует все его элементы, то он все равно обеспечивает особый метод генерации, который может быть ценен сам по себе.

Упражнения 11.1

1. а) Продолжите поиск с возвратом в задаче о четырех ферзях, начатый

в тексте раздела, и найдите второе решение этой задачи,

б) Поясните, как можно использовать симметрию доски для поиска второго решения задачи о четырех ферзях.

1. а) Каким будет последнее решение задачи о пяти ферзях, найденное

алгоритмом поиска с возвратом?

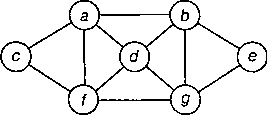
б) Воспользуйтесь симметрией доски для поиска как минимум четы­рех других решений данной задачи.

1. а) Реализуйте алгоритм поиска с возвратом для задачи о п ферзях

на языке программирования по вашему выбору. Выполните вашу программу для различных значений п и выясните количество уз­лов в соответствующих деревьях пространств состояний. Сравните полученные значения с количеством решений-кандидатов, генери­руемых алгоритмом исчерпывающего перебора,

б) Для каждого значения п, для которого выполняется программа из части а данного упражнения, оцените размер дерева пространства состояний методом, описанным в тексте раздела 11.1, и сравните полученную оценку с реальным количеством узлов дерева.

1. Примените поиск с возвратом к задаче о гамильтоновом цикле в сле­дующем графе:



1. Примените поиск с возвратом для решения задачи о 3-раскраске графа, показанного на рис. 11.3а.
2. Сгенерируйте все перестановки множества {1,2,3,4} методом поиска с возвратом.
3. а) Примените поиск с возвратом для решения следующего экземпляра

задачи о сумме подмножества: S = {1,3,4,5} и d— 11.

б) Будет ли алгоритм поиска с возвратом корректно работать при ис­пользовании только одного из двух неравенств для завершения об­работки узла как бесперспективного?

1. Общий шаблон алгоритма поиска с возвратом, приведенный в тексте раздела 11.1, работает корректно только в случае, если не существует решения, представляющего собой префикс другого решения задачи. Измените псевдокод таким образом, чтобы он корректно работал и для задач с решениями такого рода.
2. Напишите программу, реализующую алгоритм поиска с возвратом

а) для задачи о гамильтоновом цикле;

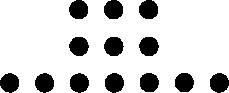
image234

б) для задачи га-раскраски графа.

1. Игра Hi-Q. В этой игре-головоломке участвуют 32 шашки, изначально размещенные на доске так, как показано ниже. Каждая шашка может перемещаться, перепрыгивая через непосредственного соседа по вер­тикали или горизонтали на пустое место; шашка, через которую был

• • •

осуществлен прыжок, удаляется с доски. Цель игры — удалить с доски 31 шашку так, чтобы последняя шашка оказалась в центре доски.



**• • • О • • •**

image236

Разработайте и реализуйте алгоритм поиска с возвратом для решения этой головоломки.

1. Метод ветвей и границ

Вспомним основную идею метода поиска с возвратом, рассматривавшегося в предыдущем разделе, — обрезка ветви дерева пространства состояний зада­чи, как только можно сделать вывод, что она не ведет к решению. Эта идея может быть усилена при работе с оптимизационными задачами, которые долж­ны минимизировать или максимизировать целевую функцию, обычно при нали­чии некоторых ограничений (длина маршрута, стоимость выбранных предметов, стоимость назначений и т.п.). Заметим, что в стандартной терминологии задач оптимизации допустимое решение (feasible solution) представляет собой точку в пространстве состояний задачи, которая удовлетворяет всем ее ограничениям (например, гамильтонов цикл в задаче коммивояжера, подмножество предметов с общим весом, не превышающим емкость рюкзака), в то время как оптимальное решение (optimal solution) является допустимым решением с наилучшим значени­ем целевой функции (например, кратчайший гамильтонов цикл, наиболее ценное подмножество предметов, помещающихся в рюкзак).

По сравнению с методом поиска с возвратом метод ветвей и границ требует:

* способа получить для каждого узла дерева пространства состояний границу наилучшего значения целевой функции[[68]](#footnote-68) для всех решений, которые могут быть получены путем дальнейшего добавления компо­нентов к частичному решению, представленному узлом;
* значение наилучшего решения, полученного к этому моменту.

Если такая информация доступна, мы можем сравнивать значение границы узла со значением наилучшего решения, полученного к этому моменту: если зна­чение границы не лучше значения уже имеющегося наилучшего решения — т.е. не меньше в случае задачи минимизации или не больше в случае задачи максими­зации, — то такой узел является бесперспективным и его обработка может быть завершена (иногда говорят, что эта ветвь обрезается), поскольку ни одно получа­емое из него решение не может оказаться лучше того, что уже имеется. В этом заключается основная идея метода ветвей и границ.

В общем случае мы завершаем путь поиска в текущем узле дерева простран­ства состояний алгоритма ветвей и границ по одной из трех следующих причин.

* Значение границы узла не лучше значения наилучшего решения, най­денного к этому моменту.
* Узел не представляет допустимых решений из-за нарушения ограниче­ний, налагаемых задачей.
* Подмножество допустимых решений, представляемое узлом, состоит из одного элемента (следовательно, дальнейший выбор невозможен) — в этом случае мы сравниваем значение целевой функции для этого допустимого решения со значением целевой функции наилучшего по­лученного к настоящему моменту решения и обновляем последнее те­кущим, если новое решение оказывается лучше.

Задача о назначениях

Проиллюстрируем метод ветвей и границ, применяя его к задаче о назначени­ях, когда имеется п работников, которым надо выполнить п заданий, и надо найти распределение заданий по работникам с наименьшей общей стоимостью. С этой задачей мы познакомились в разделе 3.4, где она решалась методом исчерпы­вающего перебора. Вспомним, что экземпляр задачи о назначениях определяется матрицей С размером пхп, так что мы можем сформулировать задачу следующим образом: выбрать по одному элементу в каждой строке матрицы так, чтобы ника­кие два выбранных элемента не располагались в одном столбце, а общая сумма выбранных элементов была минимальной. Мы продемонстрируем решение этой задачи при помощи метода ветвей и границ на том же небольшом экземпляре задачи, который рассматривался в разделе 3.4:

|  |  |  |  |
| --- | --- | --- | --- |
| Задание 1 | Задание 2 Задание 3 | Задание 4 |  |
| 9 | 2 7 | 8 | Работник а |
| 6 | 4 3 | 7 | Работник Ъ |
| 5 | 8 1 | 8 | Работник с |
| 7 | 6 9 | 4 | Работник d |

Как найти нижнюю границу стоимости оптимального решения без реального решения задачи? Можно сделать это разными путями. Например, ясно, что сто­имость любого решения, включая оптимальное, не может быть меньше суммы наименьших элементов в каждой из строк матрицы. Для приведенного приме­ра эта сумма равна 2+3+1+4— 10. Важно отметить, что это значение не является суммой ни одного из допустимых выборов (3 и 1 находятся в одном столбце матрицы), это просто нижняя граница стоимости любого выбора, соот­ветствующего ограничениям задачи. Мы можем (и будем) применять такие же рассуждения и к частично построенным решениям. Например, для любого допу­стимого решения, которое выбирает из первой строки 9, нижняя граница равна 9 + 3 + 1 + 4-17.

Перед тем как приступить к построению дерева пространства состояний, на­до сделать еще один комментарий. Он касается порядка генерации узлов дерева. Вместо генерации одного дочернего узла по отношению к последнему обещающе­му, как это делалось при поиске с возвратом, мы будем генерировать все дочерние узлы для наиболее обещающего среди незавершенных листьев текущего дере­ва (незавершенные, т.е. все еще обещающие листья иногда называют живыми (live)). Как можно определить, какие из узлов являются наиболее обещающими? Это можно сделать, сравнивая нижние границы живых листьев. Разумно рассмат­ривать как наиболее обещающий узел с наилучшей границей, хотя это, конечно, не препятствует тому, что оптимальное решение будет в конечном итоге принад­лежать иной ветви дерева пространства состояний. Такая стратегия называется методом ветвей и границ с выбором наилучшего варианта (best-first branch- and-bound).

Вернемся к экземпляру задачи о назначениях, с которой мы сталкивались ра­нее, и начнем с корня, который соответствует отсутствию какого-либо выбора элементов матрицы стоимости. Как уже говорилось, значение нижней границы (которое мы обозначим как lb) для корня равно 10. Узлы на первом уровне де­рева соответствуют четырем элементам (заданиям) в первой строке матрицы, по­скольку все они могут быть потенциальными первыми компонентами решения (рис. 11.5).

Итак, мы имеем четыре живых листа (узлы от 1 до 4), которые могут содер­жать оптимальное решение. Наиболее обещающим среди них является узел 2, поскольку он имеет наименьшее значение нижней границы. Следуя стратегии вы­бора наилучшего варианта, исследуем сперва эту ветвь, перед тем как перейти к остальным. Из рассматриваемого листа выходят три ветви, соответствующие выбору элемента из второй строки, не находящегося во втором столбце, и пред­ставляющие три различные задания, назначенные работнику Ъ (рис. 11.6).

Из шести листьев (узлы 1, 3, 4, 5, 6 и 7), которые могут содержать оптималь­ное решение, мы вновь выбираем лист с наименьшим значением нижней грани­цы, а именно — узел 5. Сначала рассмотрим выбор третьего элемента из строки с (т.е. назначение задания 3 работнику с) — при этом у нас не остается иного вы­бора, кроме назначения задания 4 работнику d, что дает нам лист 8 (рис. 11.7),

О

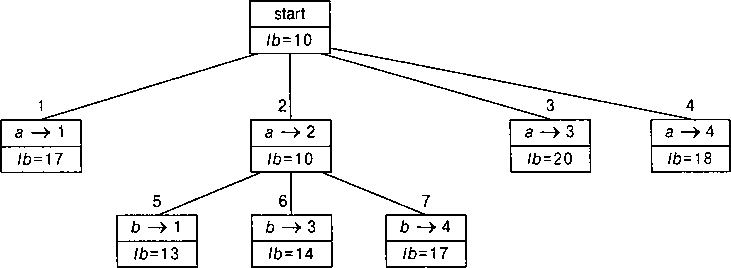


Рис. 11.6. Уровни 0, 1 и 2 дерева пространства состояний для экземпляра за­дачи о назначениях, решаемой методом ветвей и границ с выбором наилучшего варианта

о

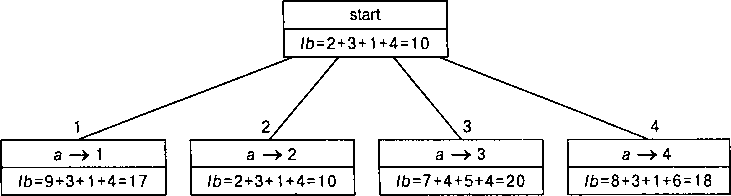


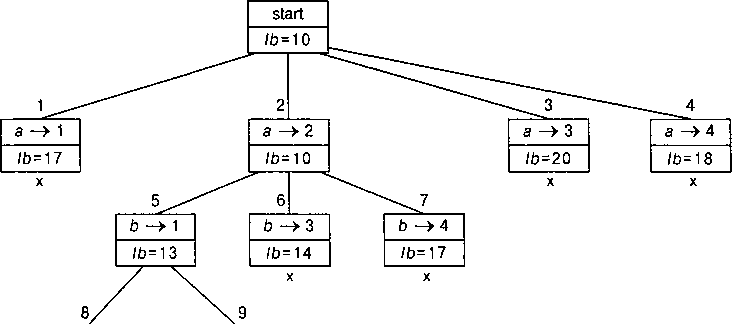
Рис. 11.5. Уровни 0 и 1 дерева пространства состояний для экземпляра задачи о назначениях, решаемой методом ветвей и границ с выбором наилучшего ва­рианта. Числа над узлами указывают порядок генерации последних. Поля узла указывают номер назначенного задания работнику а и значение нижней границы lb для данного узла

который соответствует допустимому решению {а —> 2, Ъ —► 1, с —^ 3, с? —^ 4} с об­щей стоимостью 13. “Сестринский” узел 9 соответствует допустимому решению {а —► 2, Ъ —► 1, с —> 4, d —> 3} с общей стоимостью 25. Поскольку эта стоимость больше стоимости решения, соответствующего узлу 8, работа с узлом 9 просто прекращается (если бы его стоимость была меньше 13, пришлось бы заменить ин­формацию о найденном наилучшем решении данными, соответствующими этому листу).

Теперь, если мы обратимся к каждому из пяти листьев последнего дерева пространства состояний (узлы 1, 3, 4, 6 и 7 на рис. 11.7), то обнаружим, что их значения нижних границ не меньше значения целевой функции наилучшего решения, обнаруженного к настоящему времени (равного 13, в листе 8). Сле­довательно, работа с ними прекращается, и решение, представленное листом 8, является оптимальным решением поставленной задачи.

Рис. 11.7. Полное дерево пространства состояний для экземпляра задачи о назна­чениях, решаемой методом ветвей и границ с выбором наилучшего варианта

о



с —» 4

d —^ 4 сУ-> 3

**cosf=13 *cost = 25***

Решение Худшее решение

Перед тем как распрощаться с задачей о назначениях, напомним самим се­бе еще раз, что, в отличие от следующего примера, для данной задачи имеется полиномиальный алгоритм решения под названием Венгерский метод (см., напри­мер, [87]). В свете наличия такого эффективного алгоритма решение проблемы о назначениях методом ветвей и границ следует рассматривать как учебный при­мер, а не как практическую рекомендацию.

Задача о рюкзаке

Давайте применим метод ветвей и границ к решению задачи о рюкзаке. С этой задачей мы также познакомились в разделе 3.4: дано п предметов с ве­сами гУ1,..., wn и ценами г>1,..., vn, а также рюкзак, выдерживающий вес W. Требуется найти подмножество предметов, которое можно разместить в рюкзаке и которое имеет при этом максимальную цену. Оказывается удобным упорядочить предметы в убывающем порядке по их удельной цене (отношению цены к весу), с разрешением неоднозначностей произвольным образом:

***V\/w\* ^ *V****2****/W****2* **^ ^ *vn/wn.***

Естественной структурой дерева пространства состояний для данной задачи является бинарное дерево, построенное следующим образом (рис. 11.8). Каждый узел на уровне 0 < г ^ гг представляет все подмножества из п элементов, кото­рые включают определенный выбор из первых г упорядоченных элементов. Этотчастичный выбор однозначно определяется путем от корня к узлу: ветвь, идущая влево, указывает на включение очередного элемента в подмножество, в то время как правая ветвь указывает на отсутствие элемента в подмножестве. Мы записы­ваем общий вес w и общую стоимость v выбора, соответствующего узлу, вместе с верхней границей иЪ значения для любого подмножества, которое может быть получено путем добавления некоторых элементов (возможно, никаких) к этому выбору.

о

w=0, v=0 ub=100

|  |  |  |
| --- | --- | --- |
|  |  | С1^ |
|  | 1 |  |
|  | w=4, v=40 |  |
|  | ub=76 |  |
| С 2/ |  | \Без 2 |
| 3 / |  | \4 |
| w= 11 |  | w=4, v=40 |
|  |  | ub=70 |

Недопустимо

Без 1

w=0,v=0 ub=60

Меньше, чем в узле 8



image241

|  |  |  |
| --- | --- | --- |
| iv=9, v=65 |  | w=4, v=40 |
| i/b=69 |  | ub=64 |

Меньше, чем в узле 8

image242

|  |  |  |
| --- | --- | --- |
| CsJ  II  £ |  | w=9,v=65 |
|  |  | ub=65 |

х Решение

Недопустимо

Рис. 11.8. Дерево пространства состояний алгоритма ветвей и границ для экземпляра задачи о рюкзаке

Простым способом вычисления верхней границы ub является добавление к об­щей стоимости уже выбранных элементов v произведения оставшейся емкости рюкзака W — w и наибольшего значения удельной стоимости среди оставшихся элементов, которое равно Vi+\/wi+\\

***ub = v + (W-w)(vi+i/wi+i).*** (11-1)

В качестве конкретного примера применим метод ветвей и границ к тому же экземпляру задачи о рюкзаке, который мы решали в разделе 3.4 методом исчер­пывающего перебора (здесь мы переупорядочили элементы в порядке убывания их удельных стоимостей). Емкость рюкзака W = 10.

|  |  |  |  |
| --- | --- | --- | --- |
| Предмет | Вес | Стоимость | Удельная стоимость |
| 1 | 4 | 40 | 10 |
| 2 | 7 | 42 | 6 |
| 3 | 5 | 25 | 5 |
| 4 | 3 | 12 | 4 |

В корне дерева пространства состояний не выбран ни один элемент. Следо­вательно, как общий вес, так и общая стоимость выбранных элементов равны 0. Значение верхней границы, вычисленное по формуле (11.1), равно 100. Узел 1, левый дочерний узел корня, представляет подмножество, состоящее из одного предмета, 1; общий вес и стоимость в этом узле равны, соответственно, 4 и 40, а значение верхней границы — 40 + (10 — 4) • 6 = 76. Узел 2 представляет под­множество, которое не включает предмет 1, так что в этом узле w = 0, v = 0 и иЪ — 0 + (10 — 0) • 6 = 60. Поскольку узел 1 имеет большую верхнюю границу, чем узел 2, он является более обещающим для данной задачи максимизации, и мы начинаем ветвление с узла 1. Его дочерние узлы — 3 и 4 — представляют подмно­жества с элементом 1 и с и без элемента 2, соответственно. Поскольку общий вес любого подмножества, представленного узлом 3, превосходит емкость рюкзака, работа с этим узлом завершается. У узла 4 те же значения общего веса и общей стоимости, что и у родительского, так что значение верхней границы у этого узла иЪ = 40+(10 — 4)-5 — 70. Из узлов 2 и 4 для дальнейшего ветвления мы выбираем узел 4 (почему?) и получаем узлы 5 и 6, включающий и не включающий, соответ­ственно, предмет 3. Общие вес и стоимость и значение верхней границы для этих узлов вычисляются точно так же, как и ранее. Ветвление из узла 5 дает узел 7, который дает недопустимое решение, и узел 8, представляющий подмножество {1,3}. (Поскольку никаких дополнительных предметов нет, верхняя граница для узла 8 просто равна сумме стоимостей указанных предметов.) Оставшиеся живые узлы 2 и 6 имеют меньшие значения верхней границы, чем решение, представ­ленное узлом 8. Следовательно, работа с этими узлами завершается, и множество {1,3} из узла 8 является оптимальным решением задачи.

Решение задачи о рюкзаке методом ветвей и границ имеет весьма необыч­ные характеристики. Обычно внутренние узлы дерева пространства состояний не определяют точку пространства поиска задачи, поскольку некоторые из компонен­тов решения остаются неопределенными (см., например, дерево для задачи о на­значениях, рассматривавшейся в предыдущем подразделе). В задаче же о рюкзаке каждый узел дерева представляет подмножество данных предметов. Этот факт можно использовать для обновления информации о наилучшем подмножестве после генерации каждого нового узла дерева. Для рассмотренного экземпляра это означает, что мы могли бы прекратить работу с узлами 2 и 6 еще до генерации уз­ла 8, так как значения верхних границ рассматриваемых узлов меньше стоимости подмножества в узле 5, равной 65.

Задача коммивояжера

Мы сможем применить метод ветвей и границ к экземпляру задачи коммивоя­жера, если найдем подходящий метод оценки нижней границы длины маршрута. Одна очень простая нижняя граница может быть получена путем поиска наи­меньшего элемента в матрице расстояний между городами и умножении его на количество городов п. Однако имеется менее очевидная и более информативная нижняя граница, не требующая большого количества вычислений. Нетрудно пока­зать (упражнение 11.2.8), что можно вычислить нижнюю границу длины I любого маршрута следующим образом. Для каждого г-го города (1 ^ г ^ п) находим сум­му si расстояний от города г до двух ближайших городов, после чего вычисляем сумму п этих чисел и делим результат на 2. Если все расстояния — целые числа, округляем результат до ближайшего целого:

1Ъ= [s/2] . (11.2)

Например, для экземпляра задачи, показанного на рис. 11.9л, формула (11.2) дает

lb = [[(1 + 3) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)]/2] = 14.

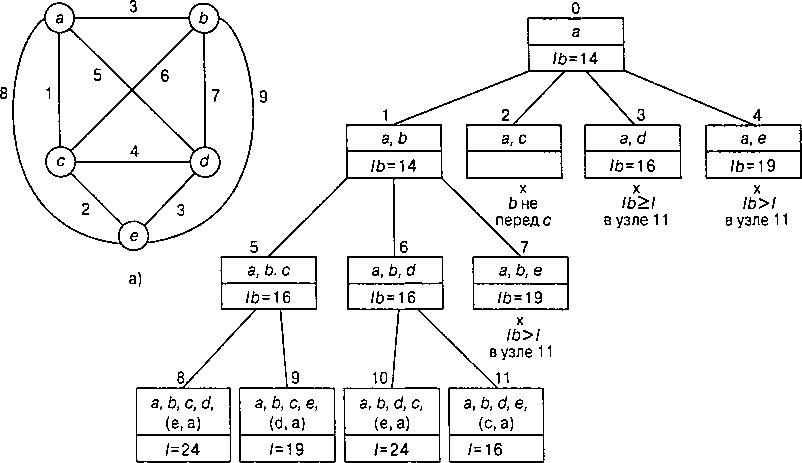
Кроме того, для любого подмножества маршрутов, которое должно включать неко­торые ребра данного графа, мы можем соответственно изменить нижнюю границу

1. . Например, для всех гамильтоновых циклов графа на рис. 11.9л, которые должны включать ребро (а, с?), мы получим следующую нижнюю границу пу­тем суммирования длин двух кратчайших ребер, инцидентных каждой из вершин (с необходимым включением ребер (а, 6) и (d, а)):

|"[(1 + 5) + (3 + 6) + (1 + 2) + (3 + 5) + (2 + 3)]/2] = 16.

Теперь применим алгоритм ветвей и границ с ограничивающей функцией (11.2) для поиска кратчайшего гамильтонова цикла графа, показанного на рис. 11.9л. Чтобы уменьшить количество потенциальной работы, воспользуемся двумя на­блюдениями, сделанными в разделе 3.4. Во-первых, без потери общности мы мо­жем рассматривать только маршруты, начинающиеся в а. Во-вторых, поскольку граф неориентированный, мы можем генерировать только маршруты, в которых Ъ посещается перед с. И в дополнение: после посещения п — 1 = 4 городов нет выбора, кроме как посетить оставшийся город и вернуться в начало маршрута. Дерево пространства состояний для данного алгоритма, примененного к графу на рис. 11.9л, показано на рис. 11.96.

Комментарии, которые были сделаны в конце предыдущего раздела о силь­ных и слабых сторонах поиска с возвратом, в той же мере применимы и к методу



Первый Лучший Худший Оптимальный

маршрут маршрут маршрут маршрут

Рис. 11.9. а) Взвешенный граф. б) Дерево пространства состояний алгоритма ветвей и гра­ниц для данного графа. Список вершин в узле указывает начальную часть гамильтонова цикла, представленного узлом

б)

ветвей и границ. Самое главное, — что оба эти метода позволяют решать мно­гие большие экземпляры сложных комбинаторных задач. Однако, как правило, почти невозможно предсказать, какие именно экземпляры окажутся решаемы за реальное время, а какие — нет.

Использование дополнительной информации, такой как симметрия доски в иг­ре, могут расширить диапазон решаемых экземпляров задач. Алгоритм ветвей и границ зачастую может быть ускорен, если знать значение целевой функции для некоторого нетривиального допустимого решения. Такую информацию мо­жет оказаться возможным получить до начала построения дерева пространства состояний, скажем, воспользовавшись конкретными данными, а для некоторых задач даже сгенерировав допустимое решение случайным образом. Такое реше­ние тут же может использоваться в качестве наилучшего, полученного к данному моменту, что позволяет не ожидать, пока ветвление приведет к какому-либо до­пустимому решению.

В противоположность поиску с возвратом решение задачи методом ветвей и границ включает как необходимость выбора порядка генерации узлов, так и по­иска хорошей функции для вычисления границ. Хотя правило выбора наилучшего

варианта, использовавшееся в этом разделе, представляет собой достаточно ра­зумный подход, оно может и не приводить к решению быстрее, чем другие стра­тегии. (Кстати, область кибернетики, посвященная искусственному интеллекту, среди прочего интересуется стратегиями разработки деревьев пространств состо­яний.)

Поиск хорошей функции для вычисления границ — задача обычно непро­стая. С одной стороны, требуется, чтобы эта функция была легко вычислимой. С другой, она не может быть слишком упрощенной — в противном случае она не сможет выполнять свою основную задачу отсечения как можно большего коли­чества ветвей дерева пространства состояний. Поиск компромисса между этими конкурирующими требованиями может потребовать интенсивных экспериментов с широким диапазоном экземпляров рассматриваемой задачи.

Упражнения 11.2

1. Какую структуру данйых вы бы предложили-использовать для отслежи­вания живых узлов в алгоритме ветвей и границ с выбором наилучшего варианта?
2. Решите тот же экземпляр задачи о назначениях, который решен в тексте раздела, при помощи алгоритма ветвей и границ с выбором наилучшего варианта, но с функцией для вычисления границ, использующей не строки, а столбцы матрицы.
3. а) Приведите пример входных данных для алгоритма ветвей и границ

для задачи о назначениях, соответствующий наилучшему случаю,

б) Сколько узлов окажется в дереве пространства состояний в наи­лучшем случае алгоритма ветвей и границ, примененного к задаче о назначениях?

1. Напишите программу для решения задачи о назначениях методом вет­вей и границ. Поэкспериментируйте с программой, чтобы определить средний размер матрицы стоимости, для которого задача решается на вашем компьютере за одну минуту.
2. Решите следующую задачу о рюкзаке методом ветвей и границ:

|  |  |  |
| --- | --- | --- |
| Предмет | Вес | Стоимость |
| 1 | 10 | 100 |
| 2 | 7 | 63 |
| 3 | 8 | 56 |
| 4 | 4 | 12 |

Емкость рюкзака W = 16.

1. а) Предложите более сложную функцию для вычисления границ в за­

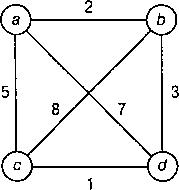
даче о рюкзаке, чем использованная в тексте раздела,

б) Воспользуйтесь вашей функцией в алгоритме ветвей и границ, при­мененном к экземпляру задачи из упражнения 5.

1. Напишите программу для решения задачи о рюкзаке методом ветвей и границ.
2. а) Докажите корректность нижней границы, вычисляемой по формуле
3. для экземпляров задачи коммивояжера с целой симметричной матрицей расстояний между городами,

б) Как нужно изменить нижнюю границу (11.2) для несимметричной матрицы расстояний?

1. Примените алгоритм ветвей и границ к решению задачи коммивояжера для приведенного графа (мы решали эту задачу в разделе 3.4 методом исчерпывающего перебора).



1. В качестве исследовательского проекта напишите реферат об исполь­зовании деревьев пространств состояний для программирования таких игр, как шахматы, шашки и крестики-нолики. При работе над рефе­ратом вы должны ознакомиться с двумя основными алгоритмами — минимаксным и альфа-бета отсечения.
2. Приближенные алгоритмы для iVP-сложных задач

В этом разделе мы рассмотрим, как работать со сложными задачами комби­наторной оптимизации, такими как задача коммивояжера или задача о рюкзаке. Как указывалось в разделе 10.3, версии принятия решения этих задач являются iVP-полными. Оптимизационные версии таких сложных задач попадают в класс NP-сложных (iVP-hard) — задач, которые имеют сложность как минимум ту же,

что и Л^Р-полные задачи.[[69]](#footnote-69) Следовательно, нет известных алгоритмов с полино­миальным временем работы, которые бы решали указанные задачи, и имеются серьезнейшие теоретические причины, по которым следует ожидать, что таких алгоритмов не существует в принципе. Как же тогда подходить к этим задачам, многие из которых весьма важны с практической точки зрения?

Если экземпляр задачи очень мал, мы можем решить его при помощи исчер­пывающего перебора (см. раздел 3.4). Некоторые из таких задач можно решить при помощи динамического программирования, как было показано в разделе 8.4. Но даже когда эти подходы применимы в принципе, на практике они ограничены только относительно малыми экземплярами. Открытие метода ветвей и границ оказалось переломным моментом, поскольку этот метод сделал возможным полу­чить решение для многих больших экземпляров сложных задач комбинаторной оптимизации за приемлемое время. Однако обычно гарантировать такую хорошую производительность невозможно.

Имеется радикально иной подход к сложным оптимизационным задачам — решать их приближенно при помощи быстрого алгоритма. Этот подход в осо­бенности подходит для приложений, где достаточно хорошего, но не обязательно оптимального решения. Кроме того, в реальных приложениях мы зачастую рабо­таем с неточными данными. В этих условиях может оказаться вполне разумным получение приближенного решения.

Хотя диапазон сложности приближенных алгоритмов весьма широк, многие из них представляют собой жадные алгоритмы, основанные на некоторой специ­фичной для данной предметной области эвристике. Эвристика (heuristic) — это основанные на здравом смысле правила, вытекающие из опыта, а не из стро­гих математически доказанных положений. Иллюстрацией этого понятия может служить, например, выбор ближайшего непосещенного города в задаче коммиво­яжера. Мы рассмотрим алгоритм, основанный на этой эвристике, позже в данном разделе.

Конечно, если мы используем алгоритм, выходом которого является прибли­жение настоящего оптимального решения, надо знать, насколько точным является это приближение. Мы можем измерять точность приближенного решения sa за­дачи минимизации некоторой функции / величиной относительной ошибки этого приближения

***re{Sa) f(s***\*) ’

где s\* — точное решение задачи. В качестве альтернативы, поскольку re (sa) = = / isa)/f (s\*) — 1, мы можем в качестве меры точности sa использовать отно­шение точности (accuracy ratio)

/ы

/ («\*)‘

***Г {Sa)*** =

Заметим, что для единообразия отношение точности приближенного решения за­дачи максимизации зачастую вычисляется как

{ а> /w

чтобы это отношение, как и для задачи минимизации, было не меньше 1.

Очевидно, что чем ближе значение г (sa) к 1, тем лучшим является приближе­ние решения. Наилучшая (т.е. наинизшая) верхняя граница возможных значений r(sa), взятая по всем экземплярам задачи, называется коэффициентом произ­водительности (performance ratio) алгоритма и обозначается Ra- Коэффициент производительности служит в качестве основной меры, указывающей качество приближенного алгоритма. Желательно, конечно, иметь алгоритм со значением Ra по возможности близким к 1, но, к сожалению, как мы увидим, некоторые простые алгоритмы имеют неограниченно большие коэффициенты производи­тельности Ra — оо). Это не означает, что такие алгоритмы нельзя использовать; надо просто быть осторожными с результатами их работы.

Мы говорим, что приближенный алгоритм с полиномиальным временем рабо­ты является с-приближенным алгоритмом (с-approximation algorithm), если его коэффициент производительности не превышает с, т.е. для любого экземпляра рассматриваемой задачи

/ (sa) ^ с/ (s\*).

Еще одним важным фактом, о котором надо помнить при решении слож­ных задач комбинаторной оптимизации, заключается в следующем. Хотя уровень сложности большинства таких задач тот же, что и у полиномиально приводи­мых к ним задач, эта эквивалентность не распространяется на приближенные алгоритмы. Как мы увидим, поиск приближенного решения с разумной степенью точности гораздо проще для одних из таких приводимых задач, чем для других.

Приближенный алгоритм для решения задачи коммивояжера

В разделе 3.4 задачу коммивояжера мы решали путем исчерпывающего пере­бора. В разделе 10.3 мы выяснили, что это одна из iVP-полных задач, а в раз­деле 11.2 увидели, как ее экземпляры могут быть решены при помощи метода ветвей и границ. Здесь мы рассмотрим два простых приближенных алгоритма для решения этой задачи.

**Алгоритм ближайшего соседа**

Этот простой жадный алгоритм основан на эвристике ближайшего соседа: всегда идти в ближайший непосещенный город.

Шаг 1. Выбрать произвольный город в качестве начального.

Шаг 2. Повторять следующую операцию до тех пор, пока не будут посещены все города: идти в непосещенный город, ближайший к последнему посещенному (неоднозначности разрешаются произвольным образом). Шаг 3. Вернуться в начальный город.

Пример 1. Для экземпляра задачи, представленного на рис. 11.10, с а в каче­стве начальной вершины, алгоритм ближайшего соседа приводит к следующему маршруту (гамильтонову циклу): sa \ а — Ъ — с — d — а длиной 10.

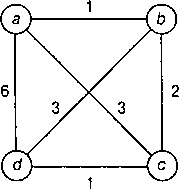


Рис. 11.10. Экземпляр задачи коммивояжера для иллюстрации алгоритма ближайшего соседа

Оптимальным решением, как легко проверить исчерпывающим перебором, является маршрут s\* : а — Ъ — d — с — а длиной 8. Таким образом, отношение точности этого приближения равно

r(s 12 \_ 1 25

**° *а)~ f(s\*)~*** 8 **-1-25**

(т.е. маршрут sa на 25% длиннее оптимального маршрута s\*). ■

К сожалению, кроме простоты, алгоритм ближайшего соседа ничем хорошим не отличается. В частности, в общем случае нельзя ничего сказать о точности получаемых им решений, так как он может заставить нас идти по очень длинному ребру на последнем этапе пути. В самом деле, если в примере 1 мы изменим вес ребра (a, d) с 6 на произвольное сколь угодно большое число w ^ 6, то алгоритм все равно будет давать в качестве решения маршрут а — b — с — d — а, длина которого равна 4 + w, в то время как оптимальный маршрут a — b — d — c — a будет иметь длину 8. Следовательно,

***Г (sa) =***

/ (sa) 4 + Wи мы можем сделать это отношение произвольно большим, выбирая подходящее значение w. Следовательно, для этого алгоритма Ra = оо.

Это, конечно, неприятное известие. Как мы увидим в конце этого подразде­ла, это связано не с простотой алгоритма ближайшего соседа, а со сложностью приближенного решения задачи коммивояжера. Однако имеется важное подмно­жество экземпляров этой задачи, называющихся евклидовыми, для которых мы можем сделать нетривиальное заключение о точности алгоритма ближайшего со­седа. Это экземпляры, в которых расстояния между городами удовлетворяют сле­дующим естественным условиям:

* ***неравенство треугольника***

d [г, j] ^ d [г, k] + d [fc, j] для любой тройки городов г, j и к

(непосредственное расстояние между городами г и j не может превос­ходить длину пути из города г в город j через некоторый промежуточ­ный город к);

* ***симметрия***

d[hj] — d [7, г] для любой пары городов г и j

(расстояние от г до j равно расстоянию от j до г).

Можно доказать (см., например, [95]), что хотя коэффициент производитель­ности алгоритма ближайшего соседа остается неограниченным для евклидовых экземпляров, отношение точности для любого такого экземпляра с п > 2 городами удовлетворяет неравенству

j^y<^(riog2nl + l),

где f (sа) и / (s\*) — соответственно, длина маршрута, полученного при помощи алгоритма ближайшего соседа, и длина кратчайшего маршрута.

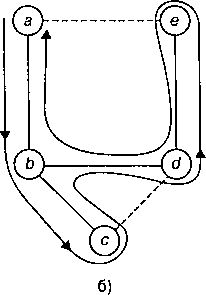
**Алгоритм двойного обхода дерева**

Теперь давайте рассмотрим простой приближенный алгоритм с конечным от­ношением производительности для евклидовых экземпляров задачи коммивояже­ра. Этот алгоритм использует связь между гамильтоновым циклом и остовными деревьями одного и того же графа.

Шаг 1. Построить минимальное остовное дерево графа, соответствующего дан­ному экземпляру задачи коммивояжера.

Шаг 2. Начиная с произвольной вершины, обойти минимальное остовное дерево, записывая пройденные вершины.

Шаг 3. Просканировать список, полученный на шаге 2, и убрать из него все повторяющиеся вершины, за исключением вершины в конце списка. Вершины, остающиеся в списке, образуют гамильтонов цикл, который и является выходом данного алгоритма.



Пример 2. Применим Зтот алгоритм к графу, показанному на рис. 11.11 а. Мини­мальное остовное дерево этого графа состоит из ребер (а, Ь), (Ь, с), (Ь, d) и (d, е) (см. рис. 11.11 б). Двойной обход минимального остовного дерева , начатый в вер­шине /, дает список вершин а, Ь, с, Ь, d, е, d, b, а.

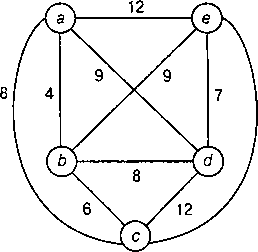


Рис. 11.11. Иллюстрация работы алгоритма двойного обхода дере­ва. а) Граф. б) Обход минимального остовного дерева с удалением повторяющихся вершин

Удаляя из списка вторую вершину Ь, вторую вершину d и третью вершину Ь, мы получим гамильтонов цикл а, Ь, с, d, е, а длиной 41. ■

Маршрут, полученный в примере 2, не оптимальный. Хотя рассмотренный экземпляр достаточно мал, чтобы можно было найти решение исчерпывающим перебором или методом ветвей и границ, мы воздержимся от этого для большей обобщенности рассмотрения. Как правило, мы не знаем, какова на самом деле длина оптимального маршрута, а потому не можем вычислить и отношение точ­ности / (sa)/f (5\*). Однако в случае алгоритма двойного обхода дерева мы можем как минимум оценить его сверху, если рассматриваемый граф — евклидов.

Теорема 1. Алгоритм двойного обхода дерева является 2-приближенным алго­ритмом решения задачи коммивояжера с евклидовыми расстояниями.

Доказательство. Очевидно, что алгоритм двойного обхода дерева полиноми­ален, если воспользоваться на первом шаге эффективным алгоритмом поиска минимального остовного дерева, таким как алгоритм Прима или Крускала. Намнадо показать, что для любого евклидового экземпляра задачи коммивояжера дли­на маршрута sa, полученного при помощи алгоритма двойного обхода дерева, не более чем в два раза превосходит длину оптимального маршрута s\*, т.е.

/(VK2/(S\*).

Поскольку удаление любого ребра из s\* дает остовное дерево Т с весом w(T), который должен быть не меньше веса минимального остовного дерева графа w (Т\*), получаем неравенство

/(s\*) >w(T)

Из этого неравенства следует, что

2/ (s\*) > 2w (Т\*) = длина пути, полученного на шаге 2 алгоритма.

Все возможные сокращения пути на шаге 3 алгоритма, приводящие к получению маршрута sa, не могут увеличить общую длину обхода евклидова графа, т.е.

Длина пути, полученного на шаге 2 ^ Длина маршрута sa.

Объединяя два последние неравенства, получим неравенство

**2** ***f(s\*)>f(sa),***

которое на самом деле несколько более строгое чем то, которое нам требовалось доказать. ■

Для евклидовой задачи коммивояжера имеются приближенные алгоритмы с лучшим коэффициентом производительности. Например, алгоритм Кристофи- деса (Christofides’ algorithm), который также основан на применении минималь­ных остовных деревьев, но более интеллектуальным и сложным способом, чем ал­горитм двойного обхода дерева, имеет коэффициент производительности 1.5 [40].

Можно ли надеяться найти полиномиальный приближенный алгоритм для задачи коммивояжера с конечным коэффициентом производительности для всех экземпляров задачи? Как показывает следующая теорема [100], ответ отрицателен, если только не выполняется соотношение Р = NP.

Теорема 2. Если Р ф NP, то не существует с-приближенного алгоритма для за­дачи коммивояжера, т.е. не существует приближенного алгоритма с полиномиаль­ным временем работы для решения этой задачи такого, что для всех экземпляров

***f(s***a) ^ ***cf(s\*)***

для некоторой константы с.

Доказательство. Будем вести доказательство от противного. Предположим, что такой приближенный алгоритм А и константа с существуют (без потери общности можно считать, что с — натуральное число). Покажем, что такой алгоритм можно использовать для решения задачи о гамильтоновом цикле за полиномиальное вре­мя. Мы воспользуемся вариацией преобразования, использованного в разделе 10.3 для приведения задачи о гамильтоновом цикле к задаче коммивояжера. Пусть G — произвольный граф с п вершинами. Отобразим граф G на полный взвешенный граф G', назначая вес 1 для каждого из его ребер и добавляя ребра весом сп + 1 между каждой парой вершин, не смежных в G. Если G содержит гамильтонов цикл, то его длина в G равна п; следовательно, этот цикл является точным реше­нием s\* задачи коммивояжера для графа G'. Заметим, что если sa — приближенное решение, полученное для графа G' алгоритмом А, то / (sa) ^ сп по предположе­нию. Если G не имеет гамильтонова цикла, то кратчайший маршрут в Gf содер­жит как минимум одно ребро весом сп + 1, а следовательно, / (sa) ^ / (s\*) > сп. С учетом двух полученных неравенств мы можем решить задачу о гамильтоновом цикле для графа G за полиномиальное время, отображдя граф G на G', применяя алгоритм А для получения кратчайшего маршрута в G' и сравнивая его длину с сп. Поскольку задача о гамильтоновом цикле iVP-полная, мы получаем проти­воречие, если только не выполняется соотношение Р — NP. ■

**Приближенные алгоритмы для задачи о рюкзаке**

Еще одна широко известная iVP-сложная задача — задача о рюкзаке, с которой мы познакомились в разделе 3.4. Дано п предметов с весами ..., wn и ценами г>1,..., vn, а также рюкзак, выдерживающий вес W. Наша задача — найти подмно­жество предметов, которое можно разместить в рюкзаке и которое имеет при этом максимальную цену. Мы видели, как можно решить эту задачу методом исчер­пывающего перебора (раздел 3.4), динамического программирования (раздел 8.4) и ветвей и границ (раздел 11.2). Теперь мы будем решать эту задачу при помощи приближенных алгоритмов.

**Жадные алгоритмы для задачи о рюкзаке**

Можно рассмотреть несколько жадных подходов к данной задаче. Один из них состоит в выборе предметов в убывающем порядке по их весам; беда в том, что более тяжелые предметы могут не быть наиболее ценными в множестве. Другой вариант состоит в выборе предметов в порядке уменьшения их стоимости, од­нако он не гарантирует эффективное использование емкости рюкзака. Можно ли найти жадную стратегию, которая бы принимала во внимание как вес, так и сто­имость предметов? Да, можно: вычисляя удельную стоимость предметов Vi/wi, г = 1,2,... ,п и выбирая предметы в порядке уменьшения удельной стоимости (в действительности мы уже использовали этот подход при разработке алгоритма ветвей и границ для задачи о рюкзаке в разделе 11.2). Вот как выглядит алгоритм, основанный на этой жадной эвристике.

Шаг 1. Вычислим удельные стоимости всех предметов множества г\* = vi/wi, г = 1,2,... ,п.

Шаг 2. Отсортируем предметы в невозрастающем порядке по их удельным сто­имостям, вычисленным на шаге 1 (неоднозначности разрешаются про­извольным образом).

Шаг 3. До тех пор пока в отсортированном списке не останется ни одного пред­мета, повторяем следующие действия: если текущий предмет помеща­ется в рюкзак, мы помещаем его туда; в противном случае переходим к следующему предмету.

Пример 3. Давайте рассмотрим экземпляр задачи о рюкзаке емкостью 10 и сле­дующей информацией о предметах:

|  |  |  |
| --- | --- | --- |
| Предмет | Вес | Стоимость |
| 1 | 7 | 42 |
| 2 | 3 | 12 |
| 3 | 4 | 40 |
| 4 | 5 | 25 |

Вычислим удельные стоимости и отсортируем предметы в невозрастающем порядке их удельных стоимостей, что даст нам следующую таблицу:

|  |  |  |  |
| --- | --- | --- | --- |
| Предмет | Вес | Стоимость | Стоимость  Вес |
| 3 | 4 | 40 | 10 |
| 1 | 7 | 42 | 6 |
| 4 | 5 | 25 | 5 |
| 2 | 3 | 12 | 4 |

Жадный алгоритм выбирает предмет 3 с весом 4, пропускает предмет 1 с ве­сом 7, затем выбирает предмет 4 с весом 5 и пропускает предмет 2 с весом

1. Полученное решение оказывается оптимальным для данного экземпляра зада­чи (см. раздел 11.2, где этот же экземпляр задачи решен методом ветвей и границ). ■

Может быть, жадный алгоритм всегда приводит к оптимальному решению? Конечно же, нет — если бы это было так, мы бы имели полиномиальный алгоритм для TVP-сложной задачи. Приведенный далее пример показывает, что при помощи этого алгоритма нельзя получить гарантированную конечную верхнюю границу точности.

Пример 4. Емкость рюкзака W > 2.

|  |  |  |  |
| --- | --- | --- | --- |
| Предмет | Вес | Стоимость | Стоимость  Вес |
| 1 | 1 | 2 | 2 |
| 2 | W | W | 1 |

Поскольку предметы уже расположены в требуемом порядке, алгоритм выби­рает первый из них и пропускает второй; общая стоимость подмножества равна при этом 2. Оптимальным же является выбор второго предмета стоимостью W. Следовательно, отношение точности г (sa) этого приближенного решения равно W/2 — величине, не ограниченной сверху. ■

Этот алгоритм очень легко модифицировать, получив приближенный алго­ритм с конечным коэффициентом производительности. Все, что для этого надо, — выбирать лучшее из двух решений: одно из них получается при помощи жадного алгоритма, а второе — из одного предмета наибольшей стоимости, который может поместиться в рюкзаке (заметим, что второй вариант в последнем примере оказы­вается лучше первого). Нетрудно доказать, что коэффициент производительности такого усовершенствованного жадного алгоритма равен 2. Таким образом, сто­имость оптимального подмножества s\* не более чем в два раза больше стоимости подмножества sa, полученного при помощи усовершенствованного жадного алго­ритма, причем 2 — наименьший множитель, для которого можно сформулировать такое утверждение.

Поучительно также рассмотреть непрерывную версию задачи о рюкзаке. В этой версии мы можем класть в рюкзак произвольные части предметов. В таком случае представляется естественной следующая модификация жадного алгоритма.

Шаг 1. Вычислим удельные стоимости всех предметов множества предметов ri = vi/wi, г = 1,2,... ,n.

Шаг 2. Отсортируем предметы в невозрастающем порядке по их удельным сто­имостям, вычисленным на шаге 1 (неоднозначности разрешаются про­извольным образом).

Шаг 3. До тех пор пока рюкзак не будет полностью заполнен или в отсортиро­ванном списке не останется ни одного предмета, повторяем следующие действия: если текущий предмет полностью помещается в рюкзак, мы берем его и переходим к следующему предмету; в противном случае мы берем только ту часть, которая может поместиться в рюкзаке (до конца заполнив его), и на этом завершаем работу.

Например, для использованного в примере 3 экземпляра задачи с четырьмя предметами данный алгоритм выбирает предмет 3 с весом 4 и 6/7 предмета 1 весом 7, после чего рюкзак оказывается заполненным.

Вас не должно удивлять, что этот алгоритм всегда приводит к оптимальному решению непрерывной версии задачи о рюкзаке. В самом деле, все предметы от­сортированы по их эффективности использования емкости рюкзака. Если первый предмет в отсортированном списке имеет вес w\ и стоимость v\, то ни одно реше­ние не может использовать w\ единиц емкости рюкзака с большей стоимостью, чем v\. Если нельзя заполнить рюкзак первым предметом или его частью полно­стью, мы должны добавить максимально возможное количество второго предмета, и т.д. По сути, это набросок строгого доказательства корректности описанного ал­горитма, которое остается читателю в качестве упражнения.

Заметим также, что стоимость оптимального решения непрерывной версии задачи о рюкзаке может служить верхней границей для дискретной версии того же экземпляра задачи. Это наблюдение обеспечивает лучший способ вычисле­ния верхней границы при решении дискретной задачи о рюкзаке методом ветвей и границ, чем тот, который использовался в разделе 11.2.

Схемы приближений

Вернемся к дискретной задаче о рюкзаке. Для этой задачи, в отличие от задачи коммивояжера, существуют полиномиальные схемы приближений (approximation

schemes), которые представляют собой параметрические семейства алгоритмов,

(к)

позволяющие получить приближения с предопределенным уровнем точности:

image248

15а I !

~~—~~ ^ 1 Н— для любого экземпляра размера п, f(s\*) к

где к — целый параметр из диапазона 0 ^ к < п. Первая схема приближений была предложена С. Сахни (S. Sahni) в 1975 году [99]. Этот алгоритм генерирует все подмножества из к или меньшего количества предметов и для каждого из под­множеств, которое помещается в рюкзак, добавляет оставшиеся предметы так же, как это делает описанный ранее жадный алгоритм (т.е. в невозрастающем порядке их удельных стоимостей). Подмножество с наибольшей суммарной стоимостью, полученное таким методом, и является выходом данного алгоритма.

Пример 5. Небольшой пример схемы приближения с к = 2 показан на рис. 11.12. В качестве оптимального решения данного экземпляра задачи алгоритм возвра­щает подмножество {1,3,4}. ■

Вряд ли на вас произвел большое впечатление приведенный пример, и это понятно. Важность этой схемы носит в основном теоретический, а не практиче­ский характер. Она заключается в том факте, что в дополнение к приближению оптимального решения с заданным уровнем точности временная эффективность данного алгоритма полиномиально зависит от п. Действительно, общее количе­ство подмножеств, генерируемых перед добавлением дополнительных элементов,

|  |  |  |  |
| --- | --- | --- | --- |
| Предмет | Вес Стоимость | Стоимость  Вес |  |
| 1 | 4 40 | 10 |  |
| 2 | 7 42 | 6 |  |
| 3 | 5 25 | 5 |  |
| 4 | 1 4 | 4 |  |
|  | Емкость W = 10 |  |  |
|  | а) |  |  |
| Подмножество | Добавляемые предметы Стоимость | | |
| 0 | 1,3,4 |  | 69 |
| {1} | 3,4 |  | 69 |
| {2} | 4 |  | 46 |
| {3} | 1,4 |  | 69 |
| {4} | 1,3 |  | 69 |
| {1,2} | Недопустимо |  |  |
| {1,3} | 4 |  | 69 |
| {1,4} | 3 |  | 69 |
| {2,3} | Недопустимо |  |  |
| {2,4} |  |  | 46 |
| {3,4} | 1 |  | 69 |

***б)***

Рис. 11.12. Пример применения схемы приближения Сахни при к = 2. а) Экземпляр задачи, б) Подмножества, генери­руемые алгоритмом

равно

**jz= о j=о i=o i=o**

Для каждого из этих подмножеств требуется время О (п), чтобы найти его возможное расширение. Таким образом, эффективность данного алгоритма — О (fcnfc+1). Обратите внимание: в то время как эффективность схемы Сахни поли­номиально зависит от п, от к она зависит экспоненциально. Более сложные схемы приближений, называющиеся полностью полиномиальными схемами (fully poly­nomial schemes), лишены этого недостатка. Среди ряда книг, в которых рассматри­ваются эти алгоритмы, хочется выделить монографию С. Мартелло (S. Martello) и П. Тота (P. Toth) [77], которая содержит богатый материал, посвященный задаче о рюкзаке.

Упражнения 11.3

1. а) Примените алгоритм ближайшего соседа к экземпляру задачи, опре­деленному приведенной ниже матрицей расстояний. Начните марш­рут с города 1, считая, что города пронумерованы от 1 до 5.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 14 | 4 | 10 | сю |
| 14 | 0 | 5 | 8 | 7 |
| 4 | 5 | 0 | 9 | 16 |
| 10 | 8 | 9 | 0 | 32 |
| оо | 7 | 16 | 32 | 0 |

б) Вычислите отношение точности этого приближенного решения.

1. а) Напишите псевдокод алгоритма ближайшего соседа. Считайте, что

входными данными алгоритма является матрица расстояний разме­ром пхп.

б) Какова временная эффективность алгоритма ближайшего соседа?

1. Примените алгоритм двойного обхода дерева к графу на рис. 11.11а с обходом минимального остовного дерева, который начинается в вер­шине а, но отличается от обхода, показанного на рис. 11.116. Рав­на ли длина полученного маршрута длине маршрута, показанного на рис. 11.116?
2. Докажите, что удаление повторяющихся вершин, используемое алго­ритмом двойного обхода дерева, не может привести к увеличению дли­ны маршрута в евклидовом графе.
3. К какому классу временной эффективности принадлежит жадный ал­горитм для решения задачи о рюкзаке?
4. Докажите, что коэффициент производительности Ra усовершенство­ванного жадного алгоритма для задачи о рюкзаке равен 2.
5. Рассмотрим жадный алгоритм для задачи об упаковке корзин, кото­рый называется алгоритмом первого подходящего (first-fit algorithm): помещаем каждый предмет в порядке поступления в первую корзину,

в которую он может поместиться; если таких корзин нет, — помещаем его в новую корзину и добавляем ее в список корзин.

а) Примените этот алгоритм к экземпляру si = 0.4, S2 = 0.7, S3 = 0.2, S4 = 0.1, S5 = 0.5 и определите, является ли полученное решение оптимальным.

б) Определите эффективность этого алгоритма в наихудшем случае.

в) Докажите, что описанный алгоритм является 2-приближенным ал­горитмом.

1. Приближенный алгоритм первого подходящего с убыванием (first-fit decreasing algorithm) для задачи об упаковке корзин начинается с сор­тировки предметов в невозрастающем порядке по их размерам, после чего работает так же, как и алгоритм первого подходящего.

а) Примените этот алгоритм к экземпляру s\ — 0.4, S2 = 0.7, S3 = 0.2, S4 = 0.1, S5 = 0.5 и определите, является ли полученное решение оптимальным.

б) Всегда ли описанный алгоритм дает Оптимальное решение? Обос­нуйте ваш ответ.

в) Докажите, что описанный алгоритм является 1.5-приближенным ал­горитмом.

г) Проведите эксперимент по выяснению того, какой из алгоритмов — первого подходящего или первого подходящего с убыванием — дает более точное приближение для случайно выбранного экземпляра задачи.

1. а) Разработайте простой 2-приближенный алгоритм для поиска мини­

мального вершинного покрытия (minimal vertex cover) — вершин­ного покрытия с минимальным количеством вершин — для данного графа.

б) Рассмотрим следующий приближенный алгоритм для поиска мак­симального независимого множества (maximal independent set) — независимого множества с наибольшим количеством вершин — для данного графа: применим 2-приближенный алгоритм из части а упражнения и выведем все вершины, не входящие в полученное вер­шинное покрытие. Можно ли утверждать, что этот алгоритм также является 2-приближенным алгоритмом?

10. а) Разработайте жадный алгоритм с полиномиальным временем рабо­ты для задачи о раскраске графа,

б) Покажите, что коэффициент производительности вашего прибли­женного алгоритма неограниченно велик.

11.4 Алгоритмы для решения нелинейных уравнений

В этом разделе мы рассмотрим несколько алгоритмов для решения нелиней­ных уравнений с одним неизвестным

/(х) = 0. (11.3)

Имеется несколько причин такого выбора среди прочих областей численного ана­лиза. Во-первых, это очень важная задача как с практической, так и с теоретиче­ской точки зрения. Она возникает в качестве математической модели множества явлений в науке и технике, как непосредственно, так и опосредованно (вспомним, например, что стандартный метод поиска точек экстремума функции / (х) основан на поиске критических точек, являющихся корнями уравнения /' (х) = 0). Во-вто­рых, она представляет наиболее доступный раздел численного анализа и в то же время демонстрирует его типичные инструменты и понятия. В-третьих, некоторые методы решения уравнений близки алгоритмам поиска в массиве, а следователь­но, являются примером применения общих методов проектирования алгоритмов к задачам непрерывной математики.

Начнем с часто встречающегося заблуждения, имеющего корни в курсе мате­матики средней школы и заключающегося в уверенности, что любое уравнение можно решить “разложением” или по готовой формуле. Но дело в том, что все встречающиеся в школе уравнения решаются таким образом только потому, что они представляют собой результат тщательного отбора среди всех уравнений та­ких, которые можно решить указанными способами. В общем случае мы не в со­стоянии точно решить уравнение, и требуется алгоритм для его приближенного решения.

Это справедливо даже для решения квадратного уравнения

ах2 + Ъх + с = 0, поскольку стандартная формула для его корней

—Ъ ± у/Ь2 — 4ас Ж1’2 = 2а

требует вычисления квадратного корня, что для большинства положительных чи­сел может быть сделано только приближенно. Кроме того, как говорилось в разде­ле 10.4, эта каноническая формула должна быть модифицирована для того, чтобы избежать возможной потери точности.

А что можно сказать о формулах для корней полиномов более высоких степе­ней? Формулы для корней полиномов третьей и четвертой степени существуют, но они слишком громоздки, чтобы иметь практическое значение. Для полиномов более высоких степеней не может существовать формул, которые бы включали только коэффициенты полиномов, арифметические операции и радикалы. Этот замечательный результат был впервые опубликован в 1799 году итальянским ма­тематиком и физиком Паоло Руффини (Paolo Ruffini) (1765-1822) и вновь от­крыт четверть века спустя норвежским математиком Нильсом Абелем (Niels Abel) (1802-1829) и получил дальнейшее развитие в работах французского математика Эвариста Галуа (Evariste Galois) (1811-1832)[[70]](#footnote-70).

Невозможность такой формулы вряд ли можно рассматривать как большую неприятность. Как отметил в своей диссертации в 1801 году великий немецкий математик Карл Фридрих Гаусс (Carl Friedrich Gauss) (1777-1855), алгебраическое решение уравнения ничуть не лучше разработки символа для корня такого урав­нения и утверждения, что уравнение имеет корень, равный этому символу [84].

Мы можем интерпретировать решение уравнения (11.3) как точку, в которой график функции f (х) пересекается с осью абсцисс х. В этом разделе мы рас­смотрим три алгоритма, использующие данную интерпретацию. Конечно, график / (ж) может пересекать ось абсцисс в одной точке (например, хг = 0), многих или даже в бесконечном количестве точек (например, sin х = 0) или не пересекать ее вообще (ех + 1 = 0). Соответственно, уравнение (11.3) будет иметь один корень, несколько корней или не иметь корней вовсе. Неплохо перед тем как начинать поиск корней функции, набросать ее график. Это может помочь определить ко­личество корней и их приближенное местоположение. В общем случае неплохо выделить корни, т.е. определить интервалы, содержащие по одному корню рас­сматриваемого уравнения.

**Метод деления пополам**

Этот алгоритм основан на том наблюдении, что график непрерывной функции должен пересечь ось абсцисс между двумя точками а и Ъ как минимум один раз, если значения функции имеют в этих точках разные знаки (рис. 11.13).

Корректность этого наблюдения доказывается в качестве теоремы в курсе вы­числительной математики, так что здесь мы просто воспользуемся этим резуль­татом. На нем основан следующий алгоритм решения уравнения (11.3), называю­щимся методом деления пополам (bisection method). Начиная с отрезка [а, Ь\, на концах которого / (х) имеет разные знаки, алгоритм вычисляет среднюю точку Xmid = (а+ Ь)/2. Если / (хш{&) = 0, корень найден, и алгоритм завершает работу. В противном случае он продолжает поиск корня либо на отрезке [a, xmid], либо

***fix)***

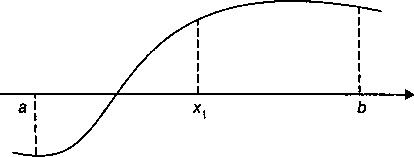


Рис. 11.13. Первая итерация метода деления по­полам: х\ — средняя точка отрезка [а, 6]

на отрезке [хт^,Ь] — в зависимости от того, на какой из двух половин отрезка функция / (х) имеет разные знаки ца концах нового отрезка.

Поскольку мы не можем ожидать, что алгоритм “наткнется” на точное решение уравнения и завершит работу, нужен иной критерий для его завершения. Можно остановиться, когда отрезок [ап,Ьп], содержащий некоторый корень я\*, станет столь мал, что можно гарантировать, что абсолютная ошибка приближения х\* величиной хп (средней точкой отрезка) меньше некоторого заранее выбранного значения е > 0. Поскольку хп — средняя точка отрезка [ап, Ьп\ и я\* лежит внутри этого отрезка, мы имеем

image250

Следовательно, мы можем завершать работу алгоритма, как только {Ъп — ап)/2 < < е, или, что то же самое,

***хп- ап< е.*** (11-5)

Нетрудно доказать, что

\хп - Х\*\ ^ bl Ql для п = 1,2,.... (11.6)

***2й***

Из этого неравенства следует, что последовательность приближений {хп} может быть сделана сколь угодно близкой к х\* выбором достаточно большого значения

п. Другими словами, мы можем сказать, что {хп} сходится к корню х\*. Заметим,

однако, что, поскольку все цифровые компьютеры представляют малые значения нулем (см. раздел 10.4), утверждение о сходимости верно в теории, но не всегда верно на практике. В действительности, если мы выберем значение е меньшим, чем определенное пороговое для данной машины значение, алгоритм никогда не завершится! Еще одним источником потенциальных сложностей являются ошиб­ки округления при вычислении значений рассматриваемой функции. Таким об­разом, желательно включать в программу, реализующую метод деления пополам, ограничение на количество итераций, которое может выполнить алгоритм.

Вот псевдокод метода деления пополам.

**Алгоритм Bisection** (/ **(я), а, Ь, eps, N)**

II Реализует метод деления пополам для поиска корня // уравнения /(я) = 0

// Входные данные: Два действительных числа а и Ъ, а < Ь,

// непрерывная на [а, Ь] функция /(я),

**// /(а) \* /(6) < 0,**

// верхняя граница абсолютной ошибки eps > 0,

// верхняя граница количества итераций N

// Выходные данные: Приближенное (или точное) значение я // корня уравнения на отрезке (а, Ь) или

// отрезок, содержащий корень (если

// достигнут предел количества итераций)

п <— 1 // Количество итераций

while n ^ TV do

я \*— (cl + Ь)/2 if я — а < eps return я jval <- /(ж) if jval = 0 return я if Jval \* /(а) < 0 b я else а я

71 71 + 1

return “Предел количества итераций”, а, Ь

С помощью неравенства (11.6) можно заранее определить количество итера­ций, достаточное (по крайней мере теоретически) для достижения требуемой сте­пени точности. Выбрав п достаточно большим, чтобы выполнялось неравенство (Pi - сь\)/2п < е, т.е.

n > log2 bl **£ Ql**, (11.7)

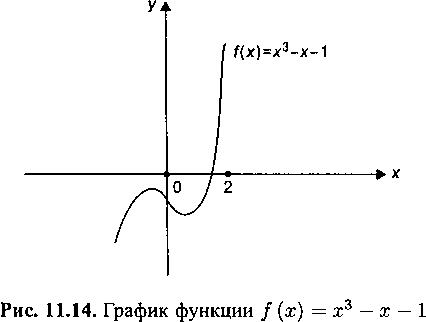
мы получим достаточное для обеспечения заданной точности количество ите­раций.

Пример 1. Рассмотрим уравнение

х3 -X- 1 = 0. (11.8)

Оно имеет один действительный корень (см. график функции / (х) — х3 —

— х — 1 на рис. 11.14). Поскольку / (0) < 0 и / (2) > 0, корень должен находиться в интервале (0,2). Выбрав допустимую ошибку е = 10 2, из неравенства (11.7) мы получим, что потребуется п > log2 (2/10-2), или п > 8 итераций.



В таблице на рис. 11.15 показаны результаты пошагового выполнения алго­ритма деления пополам для решения уравнения (11.8). Так мы получаем = = 1.3203125 в качестве приближенного значения корня х\* уравнения (11.8), га­рантируя при этом, что

1. 3203125 — ж\*| < кг2.

Кроме того, приняв во внимание знаки левой части уравнения (11.8) в точ­ках ag, bg и х8> мы можем утверждать, что корень лежит между 1.3203125 и 1.328125. ■

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| п |  | Ьп | хп | f(Xn) |
| 1 | 0.0- | 2.0+ | 1.0 | -1.0 |
| 2 | 1.0- | 2.0+ | 1.5 | 0.875 |
| 3 | 1.0- | 1.5+ | 1.25 | -0.296875 |
| 4 | 1.25- | 1.5+ | 1.375 | 0.224609 |
| 5 | 1.25- | 1.375+ | 1.3125 | -0.051514 |
| 6 | 1.3125- | 1.375+ | 1.34375 | 0.082611 |
| 7 | 1.3125- | 1.34375+ | 1.328125 | 0.014576 |
| 8 | 1.3125- | 1.328125+ | 1.3203125 | -0.018711 |

Рис. 11.15. Пошаговое выполнение алгоритма деления пополам для решения уравнения (11.8). Знаки после чисел во втором и третьем столбцах указывают знак f (х) = х3 — х — 1 в соответствующих конечных точках отрезка

Основной недостаток метода деления пополам в качестве алгоритма общего назначения для решения уравнений заключается в его низкой скорости сходи­мости по сравнению с другими известными методами. Именно по этой причине описанный метод редко используется на практике. Кроме того, его нельзя рас­пространить на уравнения более общего вида и на системы уравнений. Однако у этого метода имеются и сильные стороны. Он всегда сходится к корню, каким бы ни был начальный отрезок (свойства которого легко проверяются). Этот ме­тод не использует производную функции / (х), как это делают некоторые более быстрые методы.

Какой важный алгоритм напоминает вам метод деления пополам? Если вы сочтете, что он очень похож на бинарный поиск, то окажетесь правы. Оба эти алгоритма решают варианты задачи поиска, и оба являются алгоритмами умень­шения размера задачи в 2 раза. Основное отличие между ними — область при­менения: дискретная у алгоритма бинарного поиска и непрерывная у алгоритма деления пополам. Заметим также, что в то время как алгоритм бинарного поиска требует, чтобы массив был отсортирован, алгоритм деления пополам не наклады­вает на функцию условия невозрастания или неубывания. Наконец, в то время как бинарный поиск очень быстр, метод деления пополам достаточно медленный.

**Метод секущих**

Метод секущих[[71]](#footnote-71) (method of false position), известный также по его названию на латыни — regula falsi, соотносится с интерполяционным поиском так же, как метод половинного деления — с бинарным поиском. Как и алгоритм деления по­полам, на каждой итерации он получает некоторый отрезок [ап,Ьп], содержащий корень непрерывной функции / (х), которая имеет значения противоположных знаков в точках ап и Ьп. В отличие от метода деления пополам очередное при­ближение вычисляется не как средина отрезка [an,bn], а как точка пересечения оси абсцисс с прямой линией, проведенной через точки (ап, / (ап)) и (Ьп, / (Ъп)) (рис. 11.16).

Убедитесь самостоятельно, что формула для точки пересечения выглядит сле­дующим образом:

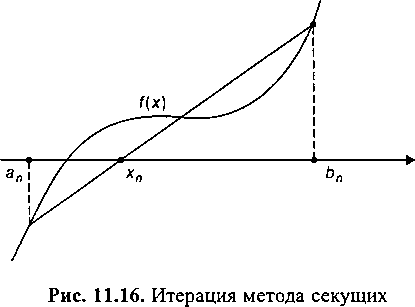
[\_ **anf (bn) ~ bnf (ап)** /1, Q4](#bookmark333)

3\*71 п /т \ Г / \ V\*

/ (Ьп) “ / \ап)

Пример 2. Таблица на рис. 11.17 содержит результаты первых восьми итераций этого метода при решении уравнения (11.8).

Хотя для данного примера метод секущих не работает так же хорошо, как ме­тод деления пополам, для многих экземпляров он дает более быстро сходящуюся последовательность. ■



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| п | ап | Ьп | хп | fixn) |
| 1 | 0.0- | 2.0+ | 0.333333 | -1.296296 |
| 2 | 0.333333- | 2.0+ | 0.676471 | -1.366909 |
| 3 | 0.676471- | 2.0+ | 0.960619 | -1.074171 |
| 4 | 0.960619- | 2.0+ | 1.144425 | -0.645561 |
| 5 | 1.144425- | 2.0+ | 1.242259 | -0.325196 |
| 6 | 1.242259- | 2.0+ | 1.288532 | -0.149163 |
| 7 | 1.288532- | 2.0+ | 1.309142 | -0.065464 |
| 8 | 1.309142- | 2.0+ | 1.318071 | -0.028173 |

Рис. 11.17. Пошаговое выполнение метода секущих для решения уравнения (11.8). Знаки после чисел во втором и третьем столбцах указывают знак / (х) =

= х3 — х — 1 в соответствующих конечных точках отрезка

**Метод Ньютона**

Метод Ньютона (Newton’s method), именуемый также методом Ньютона- Рафсона (Newton-Raphson), представляет собой один из наиболее важных алго­ритмов общего назначения для решения уравнений. Его применение к решению уравнения (11.3) с одним неизвестным проиллюстрировано на рис. 11.18: очеред­ной элемент xn+i последовательности приближений получается как пересечение касательной к графику функции / (х) в точке хп с осью абсцисс.

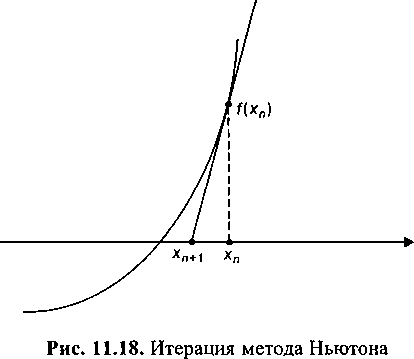
Аналитическая формула для элементов последовательности приближений ме­тода Ньютона выглядит следующим образом:

(11.10)

f(xn) п п

хп+1 = ХП- —Г для та = 0,1,.

Г (Хп)



В большинстве случаев метод Ньютона гарантирует сходимость последовательно­сти (11.10), если начальное приближение xq выбрано “достаточно близко” к корню (точное определение требований к выбору .го можно найти в учебниках по чис­ленным методам). Последовательность может сходиться и при далеком от корня начальном приближении, но это выполняется не всегда.

Пример 3. Вычисление у/а можно выполнить путем поиска неотрицательного корня уравнения х2 — а = 0. Если мы воспользуемся формулой (11.10) для / (ж) = = х2 — а и /' (х) = 2х, то получим формулу

\_ / ***(хп) \_ xl~a\_xl + a\_l{ а***

\*„+1-х„ ГЫ~Х" 2х„ - 2х„ ~ 2 ( х„

совпадающую с формулой приближенного вычисления квадратного корня из раз­дела 10.4. ■

Пример 4. Давайте применим метод Ньютона к уравнению (11.8), которое мы уже решали методом деления пополам и методом секущих. Формула (11.10) в этом случае превращается в

\_ ***Xzn - хп*** - 1

\*n+1 “ Хп 3x2-1 '

В качестве начального приближения выберем, скажем, точку хо = 2. На рис. 11.19 показаны результаты первых пяти итераций метода Ньютона. ■

Обратите внимание, насколько последовательность приближений, полученная методом Ньютона, быстрее сходится к точному решению по сравнению с мето­дом половинного деления и методом секущих. Такая очень быстрая сходимость типична для метода Ньютона, если начальное приближение оказывается близким

|  |  |  |  |
| --- | --- | --- | --- |
| п | Хп | Хп+1 | f(x n+l) |
| 0 | 2.0 | 1.545455 | 1.145755 |
| 1 | 1.545455 | 1.359615 | 0.153705 |
| 2 | 1.359615 | 1.325801 | 0.004625 |
| 3 | 1.325801 | 1.324719 | О  1  OS |
| 4 | 1.324719 | 1.324718 | 5.0-ИГ12 |

Рис. 11.19. Итерации метода Ньютона при решении уравнения (11.8)

к корню уравнения. Заметим, однако, что на каждой итерации этого метода мы должны вычислять новое значение функции и ее производной, в то время как предыдущие методы требовали вычисления только самой функции. Кроме того, метод Ньютона не указывает точно границы, в которых находится корень уравне­ния, как это делают рассмотренные ранее методы. Фактически при произвольно выбранных функции и начальном приближении последовательность приближе­ний может оказаться расходящейся. И, поскольку в формуле (11.10) в знаменателе находится производная функции, метод может оказаться неработоспособным, ес­ли она окажется равной нулю. Метод Ньютона наиболее эффективен, когда /' (х) ограничена ненулевым значением вблизи корня х\*. В частности, если

|/'(ж)| ^ mi > 0

на отрезке между хп и х\*, мы можем оценить расстояние от хп до х\* при помощи теоремы о среднем значении следующим образом:

/ (zn) - / (я\*) = /' (с) (хп - х\*),

где с — некоторая точка между хп и х\*. Поскольку / (х\*) = 0, а |/' (с)| ^ mi, мы получим

1~ **м\*\ ^ \f** (х™)1 /11 114

хп **^ • (11.11)**

***mi***

Формулу (11.11) можно использовать в качестве критерия для завершения работы метода Ньютона, когда ее правая часть станет меньше заранее выбранного уровня точности е. Другими возможными критериями могут быть

***\хп — х\*\ < е***

и

|/ ***(хп) < е***\,

где е — малое положительное число. Поскольку два последних критерия не обя­зательно означают хп к корню х\*, они должны рассматриваться как худшие по сравнению с (11.11).

Недостатки метода Ньютона не должны скрывать его главного преимущества: быстрая сходимость для произвольного начального приближения и применимость для большого количества типов уравнений и систем уравнений.

Упражнения 11.4

1. а) Найдите в Internet или в вашей библиотеке процедуру для поиска

действительного корня кубического уравнения общего вида ах3 + + Ъх2 + сх + d = 0 с действительными коэффициентами, б) На каком методе проектирования алгоритмов основана эта про­цедура?

1. Сколько корней имеют следующие уравнения:

а) **хех** **— 1 = 0 б) х** **— In х** **= 0 в)** х **sin х —** **1 = 0**

1. а) Докажите, что если р (х) — полином нечетной степени, то он должен

иметь как минимум один действительный корень.

б) Докажите, что если хо — корень полинома р (х) п-ой степени, то этот полином может быть разложен на множители следующим образом:

р **(х) =** (х **- х0)** q **(х),**

где q (ж) — полином степени п — 1. Поясните важность этой теоремы для поиска корней полиномов.

в) Докажите, что если хо — корень полинома р (х) п-ой степени, то

р' (®о) = q (яо),

где q (х) — частное от деления р (х) на х — хо-

1. Докажите неравенство (11.6).
2. Примените метод деления пополам для поиска корня уравнения

**х3 + х — 1 = 0**

с абсолютной ошибкой меньшей 10-2.

1. Выведите формулу (11.9), лежащую в основе метода секущих.
2. Примените метод секущих для поиска корня уравнения

**х3 + х — 1 = 0**

с абсолютной ошибкой меньшей 10-2.

1. Выведите формулу (11.10), лежащую в основе метода Ньютона.
2. Примените метод Ньютона для поиска корня уравнения

х3 + х — 1 = О

с абсолютной ошибкой меньшей 10-2.

1. Приведите пример, показывающий, что последовательность приближе­ний метода Ньютона может быть расходящейся.

Резюме

* Поиск с возвратом и метод ветвей и границ представляют собой два метода разработки алгоритмов для решения задач, в которых количе­ство вариантов выбора растет как минимум экспоненциально с ростом размера экземпляра задачи. В обоих методах решение строится по од­ному компоненту, с попытками завершить построение как только ста­новится понятным, что невозможно получить решение с уже выбран­ными компонентами. Такой подход делает возможным решение многих больших экземпляров iVP-сложных задач за приемлемое время.
* Как поиск с возвратом, так и метод ветвей и границ используют в каче­стве основного механизма дерево пространства состояний — корневое дерево, узлы которого представляют частично сконструированные ре­шения рассматриваемой задачи. Оба метода завершают работу с узлом, как только становится возможным гарантировать, что решение невоз­можно получить при рассмотрении решений, являющихся потомками данного узла.
* Поиск с возвратом строит дерево пространства состояний в глубину в большинстве случаев применения данного алгоритма. Если последо­вательность выбора, представленная текущим узлом дерева простран­ства состояний, может продолжаться без нарушения ограничений за­дачи, рассмотрение продолжается с первым разрешенным вариантом очередного компонента. В противном случае метод возвращается к по­следнему компоненту частично построенного решения и заменяет его очередным вариантом.
* Метод ветвей и границ представляет собой метод проектирования ал­горитмов, который усовершенствует генерацию дерева пространства состояний при помощи оценки наилучшего значения, которое можно получить из данного узла дерева. Если такая оценка не превосходит наилучшее решение, найденное к данному моменту, узел удаляется из дальнейшего рассмотрения.
* Зачастую для поиска приближенных решений задач комбинаторной оптимизации используются приближенные алгоритмы. Коэффициент производительности представляет собой основную меру точности та­ких приближенных алгоритмов.
* Алгоритм ближайшего соседа является простым жадным алгоритмом для решения задачи коммивояжера. Коэффициент производительности этого алгоритма не ограничен сверху, даже для важного подмножества евклидовых графов.
* Алгоритм двойного обхода дерева является приближенным алгоритмом для решения задачи коммивояжера с коэффициентом производительно­сти, равным 2 для евклидовых графов. Алгоритм основан на внесении изменений в путь обхода минимального остовного дерева.
* Практичный жадный алгоритм для решения задачи о рюкзаке основан на работе с предметами в порядке убывания их удельных стоимостей. Для непрерывной версии задачи алгоритм дает точное оптимальное решение.
* Полиномиальные схемы приближений для задачи о рюкзаке представ­ляют собой параметрические алгоритмы с полиномиальным временем работы и произвольным предопределенным уровнем точности.
* Решение нелинейных уравнений является одной из важных областей численного анализа. При отсутствии формул для корней нелинейных уравнений они, за небольшим исключением, могут быть решены при­ближенно рядом алгоритмов.
* Метод деления пополам и метод секущих представляют собой непре­рывный аналог бинарного и интерполяционного поисков, соответствен­но. Их главное преимущество в том, что на каждой итерации алгоритма они указывают отрезок, в котором находится корень уравнения.
* Метод Ньютона генерирует последовательность приближений кор­ня, представляющих собой точки пересечения касательных к графику функции с осью абсцисс. При хорошем начальном приближении этот метод обычно требует только нескольких итераций для получения при­ближенного значения корня с высокой точностью.

**Эпилог**

Наука — не что иное, как обученный и организованный здравый смысл.

— Томас Хаксли (Thomas Н. Huxley) (1825-1895), английский биолог и педагог

И

так, мы добрались до конца. Это была долгая дорога. Конечно, не такая дол­гая, как дорога человечества от алгоритма Евклида до последних разработок в области алгоритмов, но все равно достаточно долгая. Давайте оглянемся и бро­сим взгляд на все, что вы узнали во время нашего путешествия.

Мы начали с того, что понятие алгоритма является краеугольным камнем ин­форматики, а поскольку компьютерные программы всего лишь реализуют те или иные алгоритмы, последние являются также и основой практического програм­мирования.

Как и любая другая наука, информатика классифицирует объекты своего изу­чения. Хотя алгоритмы можно классифицировать множеством разных способов, особенно важны два из них. Можно классифицировать алгоритмы по лежащему в их основе методу проектирования и по их эффективности.

В этой книге мы рассмотрели девять основных методов проектирования:

грубая сила жадные методы

декомпозиция динамическое программирование

уменьшение размера задачи поиск с возвратом

преобразование метод ветвей и границ

пространственно-временной компромисс

Мы показали, как эти методы применяются к различным задачам, таким как сортировка, поиск, работа со строками, графами, некоторые геометрические и чис­ленные задачи. Хотя все эти методы неприменимы к каждой из задач, взятые вме­сте, они образуют отличный инструментарий для разработки новых алгоритмов и классификации старых. Кроме того, эти методы могут рассматриваться и как обобщенные способы решения задач, не ограниченных только компьютерной те­матикой. Головоломки, приведенные в книге, ясно доказывают это.

Анализ времени работы алгоритмов классифицирует их по порядку роста вре­мени работы как функции от размера входных данных. Это делается путем вы­яснения количества выполнений базовой операции алгоритма. Основным инстру­ментарием здесь являются формулы суммирования и рекуррентные соотношения, соответственно, для нерекурсивных и рекурсивных алгоритмов. Мы видели, что удивительно большое количество алгоритмов попадает в один из нескольких клас­сов из следующего списка:

|  |  |  |
| --- | --- | --- |
| Класс | Обозначение | Важные примеры |
| Постоянное время | ©а) | Хеширование (в среднем) |
| Логарифмическое | 0(logn) | Бинарный поиск (в среднем и наихудшем случаях) |
| Линейное | 0(п) | Последовательный поиск (в среднем и наихуд­шем случаях) |
| nlogn | 0 (n log п) | Эффективные алгоритмы сортировки |
| Квадратичное | 0(п2) | Элементарные алгоритмы сортировки |
| Кубическое | 0(п3) | Метод исключения Гаусса |
| Экспоненциальное | Q (2П) | Комбинаторные задачи |

Для некоторых алгоритмов мы должны различать эффективности в наихуд­шем, наилучшем и среднем случае. Средний случай наиболее сложен для анализа, так что мы рассматривали только, как это можно сделать эмпирически.

Мы коснулись и ограничений алгоритмов. Мы видели, что имеется две основ­ных причины для ограничений: внутренняя сложность задачи и необходимость работы с округленными числами в большинстве численных задач. Мы также рас­смотрели способы преодоления этих ограничений.

Естественно, вас не должно удивлять наличие областей алгоритмики, никак не затронутых в нашей книге. Наиболее важные из них — рандомизированные и па­раллельные алгоритмы. Рандомизированный алгоритм — это алгоритм, который в процессе работы делает случайные выборы. Например, мы можем случайным образом выбирать опорный элемент в алгоритме быстрой сортировки. В отличие от детерминистических алгоритмов рандомизированные алгоритмы ведут себя по- разному при разных запусках даже для одних и тех же входных данных и даже могут приводить к разным результатам. Для многих приложений рандомизирован­ные алгоритмы оказываются быстрее и/или проще, чем их детерминистические двойники.

Вероятно, наиболее впечатляющими рандомизированными алгоритмами, от­крытыми до сегодняшнего дня, являются рандомизированные алгоритмы провер­ки простоты чисел, такие как алгоритм Миллера-Рабина (см., например, [54]).

Этот рандомизированный алгоритм за приемлемое время решает задачу проверки простоты тысячезначных чисел с вероятностью ошибки, не превышающей веро­ятности ошибки аппаратного обеспечения. В настоящее время эффективные де­терминистические алгоритмы для решения этой задачи неизвестны, что является ключевым моментом современной криптологии. Если вы хотите побольше узнать о рандомизированных алгоритмах, можете обратиться к монографии Р. Мотвани (R. Motwani) и П. Рагавана (P. Raghavan) [81] и превосходному обзору Р Карпа (R. М. Karp) [60].

Подавляющее большинство современных компьютеров очень схожи с маши­ной, описанной более полувека назад фон Нейманом (John von Neumann). Ос­новным положением в этой архитектуре является последовательное выполнение команд. Соответственно, алгоритмы, разработанные для выполнения на таких машинах, называются последовательными. Именно эти алгоритмы и рассматри­ваются в данной книге. Однако это центральное положение фон Неймана не вы­полняется для тех современных компьютеров, которые могут выполнять команды одновременно, т.е. параллельно. Алгоритмы, использующие такие возможности компьютеров, называются параллельными.

Рассмотрим, например, задачу вычисления суммы п чисел, хранящихся, ска­жем, в массиве А[0..п — 1]. Можно доказать, что любой последовательный ал­горитм, который использует только умножение, сложение и вычитание, требует как минимум п — 1 шагов для решения этой задачи. Однако если разбить числа попарно и найти суммы элементов А [0] и А [1], А [2] и А [3] и т.д. параллельно, размер задачи уменьшится вдвое. Повторяя эту операцию до тех пор, пока не будет вычислена вся сумма, мы получим алгоритм, требующий только [log2 п] шагов.

Имеется богатый выбор книг, посвященных параллельным алгоритмам. Мно­гие учебники по алгоритмам включают отдельные главы, посвященные парал­лельным алгоритмам (например, [54]) или рассматривают их вместе с последова­тельными алгоритмами (например, [17, 78]).

Безудержное развитие технологий дает многообещающие результаты, такие, например, как квантовые вычислители, которые могут привести к значительным изменениям вычислительных возможностей и алгоритмов в будущем. В кванто­вых компьютерах (см. [83]) планируется использовать возможность пребывания атома в двух разных квантовых состояниях, т.е. теоретически система из п та­ких атомов (называемая кюбитом (qubit)) может хранить 2п битов информации. В 1994 году Питер Шор (Peter Shor) из AT&T Research Labs представил алгоритм разложения целых чисел, который использует эту теоретическую возможность

1. . Более того, исследователи из IBM смогли построить семикюбитовый ком­пьютер, реализующий алгоритм Шора и успешно разложивший число 15 на мно­жители 3 и 5. Хотя технологические проблемы масштабирования этой и подобных задач и перехода к задачам реальных размеров могут оказаться непреодолимыми, квантовые компьютеры потенциально в состоянии изменить наши представления о трудных задачах.

Если квантовые компьютеры пытаются воспользоваться для решения сложных вычислительных задач мощью квантовой физики, то ДНК-компьютеры пытаются сделать то же с использованием механизма генного отбора. Наиболее известный пример такого подхода относится к тому же 1994 году, когда кибернетик из США Лен Адлеман (Len Adleman) [2, 8], известный участием в разработке алгорит­ма шифрования RSA, показал, каким образом задача поиска гамильтонова цикла в ориентированном графе может быть принципиально решена путем генерации цепочек ДНК, представляющих пути в графе, и отбрасыванием тех из них, которые не удовлетворяют определению такого пути. Задача о существовании гамильто­нова пути — iVP-полная, и подход Адлемана подобен исчерпывающему перебору. Однако одновременное выполнение большого количества параллельных биохи­мических процессов оставляет надежду получить решение за приемлемое время. Адлеман решил задачу о гамильтоновом пути для небольшого графа из семи вершин, хотя был вынужден повторять части процедуры по несколько раз. Мас­штабирование подхода Адлемана для больших графов требует экспоненциально растущего количества нуклеотидов для выполнения данной процедуры. Таким образом, потенциал ДНК-вычислений остается пока что неясным, хотя нельзя отрицать его интеллектуальную привлекательность.

Итак, какое бы направление вы не выбрали для будущих путешествий по стране алгоритмов в учебе и профессиональной деятельности, предстоящая до­рога столь же волнующа и привлекательна, как и уже пройденная. Вы только в начале большого пути!

**Приложение**

Формулы, использующиеся при анализе алгоритмов

Э

то приложение содержит список полезных формул и правил, которые могут пригодиться при математическом анализе алгоритмов. Более полный мате­риал можно найти в [45, 46, 93] и [105].

Свойства логарифмов

В приведенных ниже формулах основание алгоритмов считается величиной, большей 1; его конкретная величина значения не имеет. Запись lg х в данной кни­ге означает логарифм по основанию 2, In х — логарифм по основанию е = 2.718281828...; х и у — произвольные положительные числа.

1. ***logxy — ylogx***
2. log ху = log х + log у-

X

1. log - = log X - log у

*У*

1. loga ***x =*** loga ***b logb x***
2. **alog6 x = zlogba**

Комбинаторика

1. Количество перестановок n-элементного множества: Р (п) = п!
2. Количество fc-комбинаций n-элементного множества (сочетаний из по к эле-

ментов из п): = — —

***п к\ (п — к)\***

Важные формулы суммирования

*и*

1. 1 = 1 + 1 Н + 1 = и — I + 1 (/, и — целые пределы суммирования;

^ Ч" ■ ' V ^

и—1+1 раз

п

I ^ и)\ 1 = п

г=1

v—>> 71 (П + 1) 1 о

1. / 7 — 1 + 2 + • • • + гг = ~ —тг

^ **2** **2**

г=1

1. ^г2 = 12 + 22 + --- + п2^П(га + 1)б(2п + 1)^1п[[72]](#footnote-72) г=1

— - гг2 (гг + I)2 1

г=1 гг

fc+1

**• »■**

fc + 1

1. ^ г3 = I3 + 23 н h n'J — ; — « -гг

1

1. V гк = 1к + 2fc н hnfc^ гг

' £• -4-

г=1

n \_ 1 ™

1. - 1 + а + • • • + ап = ———(а ^ 1); Х)2< = 2”+1 " 1

г=0 г=0

п

1. У г2\* = 1 • 2 + 2 • 22 + • • • + п • 2n = (п - 1) 2n+1 + 2

г=1

П 1 1 1

1. V - = 1 н 1 1— ^1пгг + 7, где 7 « 0.577216... (эта сумма называется

г 2 п

***г=1***

п-ым гармоническим числом и обозначается Нп\ константа 7 называется константой Эйлера)

п

1. lg г **«** пlg п

г=1

Правила работы с суммами

и и

1. **У COj = С У Qj**

i=l i=l

и и и

1. У (ai ± 6j) = У aj ± У **Ь{**

i=l i=l i=l

и и и

1. У (га\* + б») = сУ щ + У bt

i=l i=l i=l

4. *^2 щ* = *^2 аг* + *^2 аи* гд *Q I <и*

***i=l i=l i—m+l***

***и***

1. ^ ^ а%—i) — CL\i &1—\

***i=l***

Приближение суммы определенным интегралом

**и** U+1

' а

/ (х) dx для неубывающей функции / (х)

1. ***f(x)dx^^2f*** (г) ^

I-i i=l

и+1

**2**.

/ (х) dx ^ ^2 / (г) ^ / (х) dx для невозрастающей функции / (х)

Формулы для округлений снизу и сверху

Округление действительного числа х снизу (функция “пол”), обозначаемое как [х\, определяется как наибольшее целое, не превосходящее х (например, [3.8J = 3, [-3.8J = —4, [3J = 3). Округление действительного числа х сверху (функция “потолок”), обозначаемое как \х], определяется как наименьшее целое, не меньшее х (например, [3.8] = 4, |"—3.8] = —3, [3] = 3).

1. х — 1 ^ [х\ ^ X ^ \х] < X + 1
2. [х + ггJ = [х\ + п и \х + тг] = [ж] + п для действительного ж и целого п
3. fa/2j + fa/2] = гг
4. fig (гг Ч-1)1 = [lgnj + 1

Разное

/ /Т1\п 1 1 / /71\п

1. п! = у2ттп ( — ) eQn, где п ^ 1 и — <ап < ——, т.е. п!«\/27гп ( — )

\е/ 12n +1 12п \е/

при п —► оо (формула Стирлинга)

1. Арифметика по модулю (n, m — целые, р — положительное целое)

(п + тп) mod р = (n mod р + m mod р) mod р

(п ■ тп) mod р = ((п mod р) • (тп mod р)) mod р

**Приложение**

Краткое руководство по рекуррентным соотношениям

Последовательности и рекуррентные соотношения

Определение 1. (Числовая) последовательность (sequence) представляет собой

упорядоченный список чисел.

Примеры: 2, 4, 6, 8, 10, 12,... (положительные четные числа)

0, 1, 1, 2, 3, 5, 8, ... (числа Фибоначчи)

0, 1, 3, 6, 10, 15,... (количество сравнений ключей при сортировке выбо­

ром)

Последовательность обычно обозначается буквой (например, х или а) с под­строчным индексом (к примеру, п или г), записываемой в фигурных скобках, например {хп}. Мы воспользуемся альтернативным обозначением х(п), которое подчеркивает тот факт, что последовательность является функцией: ее аргумент п указывает позицию числа в списке, а значение функции х (п) — само число, х (п) называется обобщенным членом (generic term) последовательности.

Имеется два основных способа определения последовательности:

* явной формулой, выражающей обобщенный член как функцию от п, например х (п) = 2п для п > 0;
* уравнением, связывающим обобщенный член с одним или нескольки­ми другими членами последовательности, в комбинации с одним или несколькими явными значениями первых членов, например

(Б.1)

(Б.2)

х (п) = х (п — 1) + п для п > 0 х (0) = 0

Второй метод особенно важен для анализа рекурсивных алгоритмов (см. раз­дел 2.4).

Уравнение наподобие (Б.1) называется рекуррентным уравнением (recurrence equation) или рекуррентным соотношением (recurrence relation) (или просто ре­куррентностью (recurrence)), а (Б.2) — начальным условием (initial condition). Начальное значение может быть указано для значения п, отличного от 0, напри­мер, для п — 1, а для некоторых рекуррентных соотношений (например, для рекуррентного соотношения F (п) = F (п — 1) + F [п — 2), определяющего числа Фибоначчи — см. раздел 2.5) начальные условия должны определять несколько значений.

Решить рекуррентное соотношение при заданных начальных условиях озна­чает найти явную формулу обобщенного члена последовательности, которая удо­влетворяет как рекуррентному соотношению, так и начальным условиям, или доказать, что такая последовательность не существует. Например, решение ре­куррентного соотношения (Б.1) при начальных условиях (Б.2) равно

72 (Ti -L-

х (п) = —9— яяя п^О. (Б.З)

Непосредственной подстановкой в формулу (Б.1) можно убедиться, что урав­нение выполняется при всех п > 0, т.е. что

72 (п + 1) (п — 1) (п — 1 + 1)

—2“ = 2 + "'

а подстановкой в (Б.2) — в выполнении начального условия х (0) = 0, т.е. что

0(0 + 1) о 2

Иногда удобно различать общее и частное решения рекуррентного соотноше­ния. Обычно рекуррентное уравнение имеет бесконечное количество последова­тельностей, удовлетворяющих ему. Общее решение (general solution) рекуррент­ного уравнения представляет собой формулу, определяющую все такие последо­вательности. Обычно общее решение включает одну или несколько произвольных констант. Например, общее решение рекуррентного уравнения (Б.1) дается фор­мулой

п{п + 1)

х(п) = с-\ , (Б.4)

где с — такая произвольная константа. Присваивая с различные значения, мы можем получить все решения (Б.1), и только их.

Частным решением (particular solution) рекуррентного уравнения является конкретная последовательность, удовлетворяющая данному рекуррентному урав­нению. Обычно нас интересует частное решение, удовлетворяющее данному на­чальному условию. Например, последовательность (Б.З) представляет собой част­ное решение рекуррентного уравнения (Б.1) при начальном условии (Б.2).

Методы решения рекуррентных соотношений

Не существует универсального метода решения, который позволил бы решать любое рекуррентное соотношение (это не удивительно, поскольку у нас нет та­кого метода даже для решения существенно более простого уравнения с одним неизвестным f (х) = 0 для произвольной функции /(х)). Однако имеется ряд методов, которые позволяют решать ряд рекуррентных соотношений.

Метод прямой подстановки

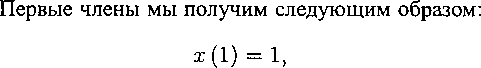
Начиная с начального члена (или членов) последовательности для данных начальных условий, мы можем использовать рекуррентное соотношение для ге­нерации нескольких первых членов его решения в надежде понять, как именно может выглядеть конечная формула. Если такая формула найдена, ее корректность должна быть проверена непосредственной подстановкой в рекуррентное уравне­ние и в начальные условия (как мы поступали для (Б.1) и (Б.2)) или доказана методом математической индукции.

Например, рассмотрим рекуррентное соотношение

(Б.5)

(Б.6)

х (п) = 2х (п — 1) + 1 для п > 1 х(1) = 1



х(2) = 2ж(1)+ 1 = 2-1 + 1 = 3,

ж{3) = 2ж (2) + 1 = 2 • 3 + 1 = 7,

ж (4) = 2ж (3) + 1 = 2 • 7 + 1 = 15.

Нетрудно заметить, что полученные числа на 1 меньше последовательных степе­ней 2:

ж (n) = 2n — 1 для п = 1... 4.

Можно доказать, что эта гипотеза дает обобщенный член решения (Б.5)-(Б.6) либо непосредственной подстановкой формулы в (Б.5) и (Б.6), либо при помощи математической индукции.

С практической точки зрения этот метод работает только для очень огра­ниченного количества случаев, поскольку обычно очень сложно распознать вид формулы по нескольким первым членам последовательности.

Метод обратной подстановки

Этот метод решения рекуррентных соотношений работает так, как указано в его названии: используя рассматриваемое рекуррентное соотношение, мы вы­ражаем х (п — 1) как функцию от х (п — 2) и подставляем результат в исходное

уравнение, чтобы получить х (п) как функцию от х (п — 2). Повторение этого шага для х (п — 2) дает выражение х (п) как функции от х (п — 3). Для многих рекур­рентных соотношений после этого мы оказываемся в состоянии выявить зависи­мость и выразить х (п) как функцию от х (п — г) для произвольного г = 1,2, — Выбирая г так, чтобы п — г достигало начального условия и используя одну их формул суммирования, зачастую удается пблучить явную формулу для решения рекуррентного соотношения.

В качестве примера применим этот метод к рекуррентному соотношению (Б.1)-(Б.2). Итак, у нас имеется рекуррентное соотношение

х (п) = х (п — 1) + п.

Заменяя в уравнении пнап-1, мы получим х (п — 1) = х (п — 2) + и — 1. После подстановки этого выражения для х (п — 1) в исходное уравнение мы получим

х (п) = (х (п — 2) + п — 1) + п = х (п — 2) + (п — 1) + п.

Замена п на п — 2 дает х (п — 2) = х (п — 3) + п — 2Т после подстановки этого выражения для х (п — 2) мы получим

х (п) = (х (п -■■■ 3) + п — 2) + (п — 1) + п = х (п — 3) + (п — 2) + (п — 1) + п.

Сравнивая три формулы для х (п), мы видим, что общий вид формулы после г подстановок имеет вид[[73]](#footnote-73)

х (п) = х (п — г) + (гг — г + 1) + (гг — г + 2) Н + п.

Поскольку начальное условие (Б.2) указано для п = 0, для его достижения требу­ется, чтобы выполнялось соотношение п — г = 0, т.е. г = п:

х (ti) = х (0) —I— 1 —2 —- \* \* —тт = 0 —1 —|— 2 —- - - —77. = тт (п + 1)/2.

Метод обратной подстановки на удивление хорошо работает для большого ко­личества простых рекуррентных уравнений. Вы можете найти немало примеров его успешного применения в данной книге (см., в частности, раздел 2.4 и упраж­нения к нему).

Линейные рекуррентные соотношения второго порядка с постоянными коэффициентами

Важный класс рекуррентных соотношений, который не может быть решен ни прямой, ни обратной подстановкой, — это рекуррентные соотношения вида

ах (п) + Ъх (п — 1) + сх (п — 2) = / (п), (Б.7)

где а, Ъ и с — действительные числа, а ф 0. Такое рекуррентное соотношение на­зывается линейным рекуррентным соотношением второго порядка с постоян­ными коэффициентами (second-order linear recurrence with constant coefficients). “Второго порядка”, поскольку в неизвестной последовательности элементы х (п) и х(п — 2) отстоят друг от друга на расстоянии двух позиций. Оно линейно, поскольку левая часть представляет собой линейную комбинацию неизвестных членов последовательности. Это уравнение с постоянными коэффициентами, так как а, b и с представляют собой фиксированные числа, не зависящие от п. Если / (гг) = 0 для всех п, рекуррентное соотношение называется однородным (homo­geneous); в противном случае оно неоднородно (inhomogeneous).

Начнем с рассмотрения однородного случая:

ах (гг) + Ъх (гг — 1) + сх (гг — 2) = 0. (Б.8)

Исключая вырожденный случай Ь — с — 0, уравнение (Б.8) имеет бесконечно много решений. Все эти решения, в совокупности образующие общее решение уравнения (Б.8), могут быть получены по одной из трех приведенных ниже фор­мул. Какая именно формула применима в каждом конкретном случае, зависит от корней квадратного уравнения с теми же коэффициентами, что и у рекуррентного уравнения (Б.8):

аг2 + Ьг + с = 0. (Б.9)

Квадратное уравнение (Б.9) называется характеристическим уравнением (char­acteristic equation) рекуррентного соотношения (Б.8).

Теорема 1. Пусть г\ и Г2 — два корня характеристического уравнения (Б.9) для рекуррентного соотношения (Б.8).

Случай 1. Если г\ и — действительны и различны, то общее решение рекур­рентного соотношения (Б.8) можно получить по формуле

х (гг) = ari + /Зг2, где а и /3 — две произвольные действительные константы.

Случай 2. Если г\ и Г2 равны, то общее решение рекуррентного соотношения (Б.8) можно получить по формуле

х (гг) = агп + /Зтггп,

где г = гх=Г2,иаи/3 — две произвольные действительные константы.

Случай 3. Если = и ± iv — два различных комплексных корня, то общее решение рекуррентного соотношения (Б.8) можно получить по формуле

х (гг) = 7n (a cos пв + /3 sin пв),

где 7 = у/и2 + v2, в = arctg (v/u), и а и /3 — две произвольные действительные константы. ■

Первый случай данной теоремы реализуется, в частности, при выводе явной формулы для п-го числа Фибоначчи (раздел 2.5). В качестве другого примера решим рекуррентное соотношение

х (п) — 6х (п — 1) + 9х (п — 2) = 0.

Его характеристическое уравнение

г2 — 6г + 9 = 0

имеет два одинаковых корня г\ — r<i — 3. Следовательно, согласно случаю 2 теоремы 1, его общее решение дает формула

х (п) = аЗп + /ЗпЗп.

Если вы хотите найти частное решение, для которого, скажем, х (0) = 0 и х (1) = = 3, подставьте п = 0ип = 1в последнее уравнение, чтобы получить систему двух линейных уравнений с двумя неизвестными. Для указанных начальных усло­вий а = 0 и (3 — 1, следовательно, искомое частное решение —

***х (п) = пЗп.***

Давайте теперь вернемся к случаю неоднородного линейного рекуррентного соотношения второго порядка с постоянными коэффициентами.

Теорема 2. Общее решение неоднородного уравнения (Б.7) может быть получе­но как сумма общего решения соответствующего однородного уравнения (Б.8) и частного решения неоднородного уравнения (Б.7). ■

Поскольку теорема 1 дает возможность получить общее решение однородного линейного рекуррентного уравнения второго порядка с постоянными коэффици­ентами, теорема 2 сводит задачу поиска всех решений уравнения (Б.7) к поис­ку одного частного решения. Для произвольной функции / (п) в правой части уравнения (Б.7) задача остается весьма сложной, не имеющей общих способов решения. Однако для некоторых классов функций может быть найдено частное решение. В частности, если / (п) — ненулевая константа, мы можем искать част­ное решение, также представляющее собой константу.

В качестве примера найдем общее решение неоднородного рекуррентного уравнения

х (п) — 6х (п — 1) + 9х (п — 2) = 4.

Если х (п) = с — частное решение этого рекуррентного уравнения, то величина с должна удовлетворять уравнению

с — бс + 9с — 4,

откуда с = 1. Поскольку мы уже нашли общее решение соответствующего одно­родного рекуррентного уравнения

х (га) — 6х (га — 1) + 9х (га — 2) = 0,

общее решение рекуррентного соотношения х (га) - 6х (га — 1) + 9а; (га — 2) = 4 можно записать в виде

х **(га) = аЗп + /?пЗп + 1.**

Перед тем как завершить рассмотрение этой темы, заметим, что результаты, аналогичные теоремам 1 и 2, справедливы и для обобщенного линейного рекур­рентного уравнения п-ой степени с постоянными коэффициентами (general linear fcth degree recurrence with constant coefficients)

akx (ra) + ak-ix (ra - 1) H h a0x (ra - fc) = / (га). (Б.10)

Однако практичность этого обобщения весьма ограничена, поскольку при этом требуется искать корни полинома к-ой степени:

+ afc\_ir^ 1 + • • • + ao = 0. (Б. 11)

Уравнение (Б. 11) является характеристическим уравнением для рекуррентного соотношения (Б.10).

Имеется также несколько других, более сложных методов решения рекуррент­ных соотношений. Особенно богатым материалом на эту тему, рассмотренным с точки зрения анализа алгоритмов, отличается книга Пурдома (Purdom) и Брауна (Brown) [93].

Распространенные типы рекуррентных соотношений в анализе алгоритмов

Имеется несколько видов рекуррентных соотношений, которые с завидной регулярностью появляются при анализе алгоритмов. Это связано с тем, что они отражают один из фундаментальных методов проектирования алгоритмов.

Уменьшение на единицу

Алгоритм уменьшения на единицу решает поставленную задачу, используя соотношение между данным экземпляром размером п и меньшим экземпляром размером п — 1. К конкретным примерам относится рекурсивное вычисление п! (раздел 2.4) и сортировка вставкой (раздел 5.1). Рекуррентное соотношение, возникающее при изучении временной эффективности таких алгоритмов, обычно имеет вид

Г(п)=Г(п-1) + /(п), (Б. 12)

где функция / (п) учитывает время, необходимое для приведения экземпляра зада­чи к меньшему и возврата от решения меньшего экземпляра к решению большего экземпляра. Применение обратной подстановки к (Б. 12) дает

Г(п)=Г(п-1) + /(п) =

= Т (п - 2) + / (п - 1) + / (п) =

= T(0) + **£/(j).**

3=1

Для конкретной функции / (га) сумма J2j=i f U) обычно либо вычисляется точно, либо выясняется ее порядок роста. Например, если / (га) = 1, то J2j=i /С?) = п’ если / (п) = logп, то 1 / (Л е 0 (nlogn); если / (п) = пк, то YTj=i f O') е € 0 (nk+1). Сумма Y^j=if (Л может быть также приближенно вычислена при помощи формул, включающих интегралы (см., в частности, соответствующие формулы в приложении А).

Уменьшение на постоянный множитель

Алгоритм уменьшения на постоянный множитель решает задачу путем приве­дения ее экземпляра размером п к экземпляру размером п/Ъ (для большинства, но не для всех алгоритмов Ъ = 2). Алгоритм рекурсивно решает меньшие экземпля­ры; затем при необходимости он распространяет решение меньшего экземпляра на решение исходного экземпляра задачи. Наиболее важным примером такого алго­ритма является бинарный поиск; другие примеры включают возведение в степень посредством вычисления квадратов (см. введение к главе 5), умножение по-рус­ски и задачу поиска фальшивой монеты (раздел 5.5). Рекуррентное соотношение, возникающее при изучении временной эффективности таких алгоритмов, обычно имеет вид

Т(п) =Т (п/Ь) + / (п), (Б. 13)

где b > 1 и функция / (п) учитывает время, необходимое для приведения экзем­пляра задачи к меньшему и возврата от решения меньшего экземпляра к реше­нию большего экземпляра. Строго говоря, уравнение (Б. 13) корректно только для п = Ьк, к = 0,1, Для значений тг, не являющихся степенями Ь, обычно при­меняется округление при помощи функций для округления сверху и снизу (“пол” и “потолок”). Стандартный подход к таким уравнениям заключается в первона­чальном решении для тг = Ьк. После этого решение либо немного корректируется, чтобы быть верным для всех тг (см., например, раздел 4.3 и упражнение 4.3.3), ли­бо на основании правила гладкости (smoothness rule), сформулированного в виде теоремы 4 в данном приложении, определяется порядок роста решения для всех п.

Рассматривая п — bk, к = 0,1,... и применяя обратную подстановку к (Б. 13), получаем:

**Т** **(**V**) = Т** (b**k~х) + /** (V) **=**

= г (ьк~2^ + / (ьк-+ / (V) =

***к***

= Г(1)+ £/(&\*).

J=1

Для конкретной функции / (п) сумма / (^) обычно либо может быть вы­числена точно, либо получен ее порядок роста. Например, если / (n) = 1,

***к***

***’^2f(bj)=k =*** log6 ***п. з***=1

Если / (п) = п, мы получаем другой пример точного вычисления суммы:

***3 = 1 3=1***

Кроме того, рекуррентное соотношение (Б. 13) является частным случаем рекур­рентного соотношения (Б. 14), для которого выполняется Основная теорема (Mas­ter theorem) (теорема 5 из данного приложения). Согласно этой теореме, в част­ности, если / (n) G fi (nd), где d > 0, то и Т (п) G П (nd).

Декомпозиция

Алгоритм декомпозиции решает задачу путем разделения данного экземпляра на несколько меньших, рекурсивного решения каждого из них и при необходимо­сти комбинации решений меньших экземпляров в решение данного экземпляра. В предположении, что все меньшие экземпляры имеют один и тот же размер п/Ъ и реально решаются а из них, мы получим следующее рекуррентное соотношение, справедливое для п = Ьк, к = 1,2,...:

***Т (п) = аТ {п/Ъ)*** + / ***(п)***, (Б. 14)

где а ^ 1, b ^ 2, а / (га) — функция, учитывающая время, затрачиваемое на разде­ление задачи на меньшие и комбинацию их решений. Рекуррентное соотношение (Б. 14) называется общим рекуррентным соотношением декомпозиции (general divide-and-conquer recurrence)[[74]](#footnote-74).

Применение обратной подстановки к (Б. 14) приводит к следующему резуль­тату:

Г (V) = аТ (V-1) + / (б\*) =

= а (Т (V-2) + / (V"1)) + / (V) =

= а2Т (V-2) + а/ (Vе-1) + / (V) =

= а2 (аТ (V~[[75]](#footnote-75)) + / (V-2)) + а/ (b\*"1) + / (V) =

= а3Т (V~3) + а2/ (Vе-2) + а/ (bk~1^ + / =

- акТ (1) + ak~lf (b1) + ak~2f (Ъ2) + • • • + а0/ (V) =

= а" + •

Поскольку ак = alogi>n = nlog(>a, получаем следующую формулу для решения рекуррентного соотношения (Б. 14) при п = Ьк:

(

logb \

Т(Х) + J2 f(V)/aj • (Б. 15)

Очевидно, что порядок роста решения Т (п) зависит от значений констант а и b и порядка роста функции / (п). При определенных предположениях о свойствах функции / (п), рассматриваемых в этом разделе, упростив формулу (Б. 15), можно получить точный результат для порядка роста Т (п).

Правило гладкости и Основная теорема

Ранее мы упоминали, что временная эффективность алгоритмов уменьшения на постоянный множитель и декомпозиции обычно сначала рассматривается для значений п, являющихся степенями b (чаще всего b = 2, как в случае бинарного поиска или сортировки слиянием; иногда Ъ = 3, как в случае лучшего алгорит­ма решения головоломки о взвешивании монет из раздела 5.5, но вообще говоря

Ъ может принимать любое значение, не меньшее 2). Вопрос, который мы рас­смотрим, заключается в распространении результата поиска порядка роста для п, являющихся степенями Ь, на все значения п.

Определение 1. Пусть / (га) — неотрицательная функция, определенная на мно­жестве натуральных чисел. / (га) называется в конечном счете неубывающей (eventually nondecreasing), если существует некоторое неотрицательное целое чис­ло по такое, что / (га) является неубывающей функцией на интервале [гао,оо), т.е. / (rai) ^ / (гаг) для любых гаг > п\ ^ по- ■

Например, функция (п — 100)2 является в конечном счете неубывающей, хотя она и убывает на отрезке [1,100]; функция же sin2 ттп/2 представляет собой функ­цию, не являющуюся в конечном счете неубывающей. Подавляющее большин­ство функций, с которыми мы сталкивались при анализе алгоритмов, являются в конечном счете неубывающими (большинство из них на самом деле являются неубывающими во всей их области определения).

Определение 2. Пусть / (п) — неотрицательная функция, определенная на мно­жестве натуральных чисел. / (п) называется гладкой (smooth), если она в конеч­ном счете неубывающая и / (2га) Е0(/ (п)). ■

Легко убедиться, что функции, растущие не слишком быстро, включая log п, п, nlogn и па, где а ^ 0, являются гладкими. Например, / (n) = nlogn — гладкая, поскольку

/ (2п) = 2пlog 2п = 2ra(log 2 + logn) = (2 log 2) n + 2nlogn G 0 (n log n).

Быстро растущие функции, такие как ап, где а > 1 и п!, не являются гладкими. Например, функция / (п) = 2п не является гладкой, так как

/ (2n) = 22п = 4n g 0 (2П).

Теорема 3. Пусть / (п) — гладкая функция. Тогда для любого целого 6^2

/ (Ьп) € 0 (/ (п)),

т.е. существуют положительные константы сь и db и неотрицательное целое число по такие, что

dbf (п) ^ / (Ьп) ^ Q,/ (п) при п ^ п0.

(То же утверждение, с очевидными изменениями, справедливо и для ОиП обо­значений.)

Доказательство. Докажем теорему только для О-обозначения; доказательство для Q аналогичное. По индукции легко убедиться, что если / (2п) ^ c^f (п) для п ^ по, то

/ (2knj ^ ckf (п) при fc = 1,2,... и п ^ по-

Гипотеза индукции при к = 1 проверяется тривиально. В общем случае, полагая, что / (2к~г

п) ^ с2 V (п) ПРИ п ^ по> получим

/ (2fcn) = f (2- 2k~1nj ^ с2/ ^ c2ck~lf (n) = 4f (n) ■

(Тем самым теорема доказывается для Ъ = 2к.) Рассмотрим теперь произвольное целое Ъ ^ 2. Пусть к — положительное целое такое, что 2к~1 < 2к. Мы можем

оценить / (Ьп) сверху, без потери общности, считая, что / (п) — неубывающая функция при п ^ по\*.

/ (Ьп) ^ / (2\*°™) ^ с2/ (п) •

Следовательно, мы можем использовать с| в качестве константы для данного значения Ь, что и завершает доказательство. ■

Важность введенных понятий объясняется следующей теоремой.

Теорема 4 (Правило гладкости). Пусть Т (п) — в конечном счете неубывающая функция, а / (п) — гладкая функция. Если

Т (п) € 0 (/ (п)) для значений п, равных степени 6,

где b ^ 2, то

Т (п) € 0 (/ (п)).

(Аналогичные результаты выполняются для О иЙ обозначений.)

Доказательство. Мы докажем только О-часть; доказательство для S7 выполняет­ся аналогично. В соответствии с формулировкой теоремы, имеется положительная константа с и положительное целое по = такие, что

Т (bfc) ^ с/ для 6fc ^ По,

Т (п) — невозрастающая функция при п > по, а / (Ьп) ^ с&/ (п) при п ^ по в соответствии с теоремой 3. Рассмотрим произвольное значение п ^ по- Оно ограничено двумя последовательными степенями b: по ^ bk ^ п < bk+1. Таким образом,

Т(п)^Т (V+1) ^ с/ (V+1) = с/ (bbfc) ^ сс6/ (V) < ccbf (n).

Следовательно, мы можем использовать произведение ссь в качестве констан­ты, требующейся в определении 0(/(п)), что и завершает доказательство

■

теоремы.

Теорема 4 позволяет распространить информацию о порядке роста, найденную для Т{п) для удобного подмножества значений (степеней Ь), на всю область определения данной функции.

Теорема 5 (Основная теорема). Пусть Т (п) — в конечном счете неубывающая функция, удовлетворяющая рекуррентному соотношению

Т (п) = аТ (п/Ъ) + / (п) при п = Ьк, к = 1, 2,...; Т (1) = с,

где а ^ 1, b ^ 2, с > 0. Если / (п) е 0 (nd), где 0, то

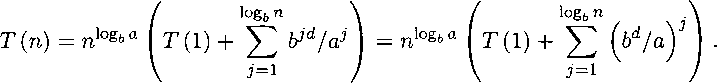
***' Q (nd)*** если ***а < bd,***

Т (п) € < 0 (nd log п) если а = bd,

ч0 (nlogba) если a >

(Аналогичные результаты справедливы для О иЙ обозначений.) Доказательство. Мы докажем теорему для важного частного случая / (n) = nd

(доказательство для общего случая представляет собой небольшое техническое изменение приведенного — см., например, [32]). Если / (n) = nd, уравнение (Б. 15) дает для п = Ьк, к = 0,1,...



Сумма в этой формуле представляет собой геометрическую прогрессию, так что

image256

logbn

((,■</«) log‘" -1

, , если Ь ф о,

и

logbn j

У (bd/a) = log^n, если 6d = й. j'=i

Если a < б6\*, то bd/a > 1 и, таким образом,

Следовательно,

**log6n**

Т (п) = nlog<-a |г(1) + J2 (bd/a) J enlogoa0 ((^/а)1°ёьП) =

= 0 (nlogba (бУй)10ёьП) = 0 ^alogbп (бУа)'°ёьП) =

= 0 (Vlogb"j = 0 (>gbn^ = © (nd^j .

Если а > bd, то bd/a < 1 и, таким образом,

1о£ьп • fud / \loSьп л

***3 = 1***

Следовательно,

**bg6n**

Т(n) = nlogba Т(1) + е 0 (nlog<-a) .

i=i

Если a = то bd/а = 1 и, следовательно,

**log6n**

Т (n) = nlogba ^T(l) + (6<i/a) J =

= nl0gb a (T (1) + logb 77.) £ 0 ^nl0gb a logfc nj =

= 0 (п1оёь **bd** log6 **n**j = 0 **(nd** logb **nj .**

Поскольку f (n) = nd — гладкая функция при любом d, обращение к теореме 4 завершает доказательство. ■

Теорема 5 предоставляет очень удобный инструмент для быстрого анализа эффективности алгоритмов декомпозиции и уменьшения размера на постоянный множитель. В книге имеется много примеров его применения.

Список литературы

1. Adelson-Velsky, G.M. and Landis, E.M. An algorithm for organization of infor­mation. Soviet Mathematics Doklaay, vol. 3, 1962, 1259-1263.
2. Adleman, L.M. Molecular computation of solutions to combinatorial problems. Science, vol. 266, 1994, 1021-1024.
3. Agrawal, М., Kayal, N., and Saxena, N. PRIMES Is in P. Preprint, Department of Computer Science and Engineering, Indian Institute of Technology Kanpur, Kanpur-208016, India, August 6, 2002.
4. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, MA, 1974.
5. Aho, A.V, Hopcroft, J.E., and Ullman, J.D. Data Structures and Algorithms. Addison-Wesley, Reading, MA, 1983. (Альфред Axo, Джон Хопкрофт, Джеф­фри Ульман, Структуры данных и алгоритмы. М., ИД “Вильямс”, 2000.)
6. Albert, R., Hawoong Jeong, and Barb6si, A.-L. Diameter of the World-Wide Web. Nature, vol. 40 Г, 1999, 130-131.
7. Atallah, M.J., editor. Algorithms and Theoiy of Computation Handbook. CRC Press, Boca Raton, FL, 1998.
8. Baase, S., and Van Gelder, A. Computer Algorithms: Introduction to Design and Analysis, 3rd ed. Addison-Wesley, Reading, MA, 2000.
9. Baecker, R. (with assistance of D. Sherman). Sorting Out Sorting. 30-minute color sound film. Dynamic Graphics Project, University of Toronto, 1981. (Dis­tributed by Morgan Kaufmann, Publishers.)
10. Baecker, R. Sorting out sorting: a case study of software visualization for teaching computer science. In Software Visualization: Programming as a Multimedia Experience, edited by J. Stasko, J. Domingue, M.C. Brown, and B.A. Price. MIT Press, Cambridge, MA, 1998, 369-381.
11. Baeza-Yates, R.A. Teaching Algorithms. ACM SIGACT News, vol. 26, no. 4, Dec. 1995, 51-59.
12. Bayer, R., and McGreight, E.M. Organization and maintenance of large ordered indices. Acta Informatica, vol. 1, no. 3, 1972, 173-189.
13. Bellman, R.E. Dynamic Programming. Princeton University Press, Princeton, NJ, 1957.
14. Bellman, R.E., and Dreyfus, S.E. Applied Dynamic Programming, Princeton University Press, Princeton, NJ, 1962.
15. Bentley, J. Programming Pearls, 2nd ed. Addison-Wesley, Reading, MA, 2000 (Бентли Дж. Жемчужины программирования. 2-е изд. — СПб. Питер — 2002.).
16. Berlinski, D. The Advent of the Algorithm. Harcourt, Inc., New York, 2000.
17. Berman, K.A., and Paul, J.L. Fundamentals of Sequential and Parallel Algo­rithms. PWS Publishing, Boston, 1996.
18. Berstekas, D.P. Dynamic Programming and Optimal Control, 2nd Edition (2 vols.), Athena Scientific, 2001.
19. Bloom, М., Floyd, R.W., Pratt, V, Rivest, R.L., and Tarjan, R.E. Time bounds for selection. Journal of Computer and System Sciences, vol. 7, no. 4, 1973, 448-461.
20. Bogomolny, A. Interactive Mathematics Miscellany and Puzzles, http: / / [www.cut-the-knot.org](http://www.cut-the-knot.org), 1996.
21. Boyer, R.S., and Moore, J.S. A fast string searching algorithm. Communications of the ACM, vol. 21, no. 10,1977, 762-772.
22. Brassard, G., and Bratley, P. Fundamentals of Algorithmics. Prentice Hall, En­glewood Cliffs, NJ, 1996.
23. Brown, M.R. ZEUS: A system for algorithm animation and multi-view editing. In Proceedings of the 1991 IEEE Workshop on Visual Languages, Kobe, Japan, October 1991, 4-9.
24. Brown, М., and Sedgewick, R. A system for algorithm animation. In Proceedings of ACM SIGGRAPH '84, Minneapolis, July 1984, 177-186.
25. Brualdi, R. Introductory Combinatorics, 3rd ed. Prentice Hall, Upper Saddle River, NJ, 1999.
26. Chabert, Jean-Luc, editor. A History of Algorithms: From the Pebble to the Microchip. Translated by Chris Weeks. Springer-Verlag, Berlin, 1998.
27. Chandler, J.P. Patent protection of computer programs. Minnesota Intellectual Property Review, vol. 1, no. 1, 2000, 33-46.
28. Comer, D. The ubiquitous В-tree. ACM Computing Surveys, vol. 11, no. 2, 1979, 121-137.
29. Computing Curricula 2001. The Joint Task Force on Computing Curricula. IEEE Computer Society, Association for Computing Machinery, 2001.
30. Cook, S.A. The complexity of theorem-proving procedures. In Proceedings of the Third Annual ACM Symposium on the Theory of Computing, 1971, 151-158.
31. Coopersmith, D., and Winograd, S. Matrix multiplication via arithmetic progres­sions. In Proceedings of Nineteenth Annual ACM Symposium on the Theory of Computing, 1987, 1-6.
32. Cormen, Т.Н., Leiserson, C.E., Rivest, R.L., and C. Stein. Introduction to Algo­rithms, 2nd ed. MIT Press, Cambridge, MA, 2001. (Кормен Томас, Лейзерсон Чарльз, Ривест Рональд, Стайн Клиффорд, Введение в алгоритмы. Второе издание. М., ИД “Вильямс”, 2005.)
33. Dantzig, G.B. Linear Programming and Extensions. Princeton University Press, Princeton, NJ, 1963.
34. Dewdney, A.K. The (New) Turing Omnibus. Computer Science Press, New York,

1993.

1. Dijkstra, E.W. A note on two problems in connection with graphs. Numerische Mathematik, vol. 1, 1959, 269-271.
2. Floyd, R.W. Algorithm 97: shortest path. Communications of the ACM, vol. 5, no. 6, 1962, 345.
3. Forsythe, G.E. What to do till the computer scientist comes. American Mathe­matical Monthly, vol. 75, 1968, 454-462.
4. Forsythe, G.E. Solving a quadratic equation on a computer. In The Mathematical Sciences, edited by COSRIMS and George Boehm, MIT Press, Cambridge, MA, 1969, 138-152.
5. Gardner, M. My Best Mathematical and Logic Puzzles. Dover Publications, New York, 1994.
6. Garey, M.R., and Johnson, D.S. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, New York, 1979.
7. Gerald, C.F., and Wheatley, P.O. Applied Numerical Analysis, 6th ed. Addison- Wesley, Reading, MA, 1999.
8. Gloor, P.A. User interface issues for algorithm animation. In Software Visual­ization: Programming as a Multimedia Experience, edited by John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. MIT Press, Cambridge, MA, 1998, 145-152.
9. Gonnet, G.H., and Baeza-Yates, R. Handbook of Algorithms and Data Structures in Pascal and C, 2nd ed. Addison-Wesley, Reading, MA, 1991.
10. Goodrich, M.T., and Tamassia, R. Algorithm Design: Foundations, Analysis, and Internet Examples. John Wiley & Sons, New York, 2002.
11. Graham, R.L., Knuth, D.E., and Patashnik, О. Concrete Mathematics: A Foun­dation for Computer Science, 2nd ed. Addison-Wesley, Reading, MA, 1994. (P. Грэхем, Д. Кнут, О. Паташник. Конкретная математика. Основание информатики. М., “Мир”, 1998.)
12. Green, D.H., and Knuth, D.E. Mathematics for Analysis of Algorithms, 2nd ed. Birkhzmser, Boston, 1982.
13. Gries, D. The Science of Programming. Springer-Verlag, New York, 1981.
14. Harel, D. Algorithmics: The Spirit of Computing, 2nd ed. Addison-Wesley, Read­ing, MA, 1992.
15. Hartmanis, J., and Steams, R.E. On the computational complexity of algorithms. Transactions of the American Mathematical Society, vol. 117, 1965, 285-306.
16. Heap, B.R. Permutations by interchanges. Computer Journal, vol. 6, 1963, 293-294.
17. Hoare, C.A.R. Quicksort. In Great Papers in Computer Science, Phillip Laplante, ed. West Publishing Company, Minneapolis/St. Paul, 1996, pp. 31-39.
18. Hochbaum, D.S., editor. Approximation Algorithms for NP-Hard Problems. PWS Publishing, Boston, 1997.
19. Hopcroft, J.E. Computer science: The emergence of a discipline. Communica­tions of the ACM, vol. 30, no. 3, 1987, pp. 198-202.
20. Horowitz, E., Sahni, S., and Rajasekaran, S. Computer Algorithms. Computer Science Press, New York, 1998.
21. Horspool, R.N. Practical fast searching in strings. Software—Practice and Expe­rience, vol. 10, 1980, 501-506.
22. Huffman, D.A. A method for the construction of minimum redundancy codes. Proceedings of the IRE, vol. 40, 1952, 1098-1101.
23. Karmarkar, N. A new polynomial-time algorithm for linear programming. Com- binatorica, vol. 4, 1984, 373-395.
24. Karp, R.M. Reducibility among combinatorial problems. In Complexity of Com­puter Communications, edited by R.E. Miller and J.W. Thatcher. Plenum Press, New York, 1972, 85-103.
25. Karp, R.M. Combinatorics, complexity, and randomness. Communications of the ACM, vol. 29, no. 2, 1986, 89-109.
26. Karp, R.M. An introduction to randomized algorithms. Discrete Applied Mathe­matics, vol. 34, 1991, 165-201.
27. Kemighan, B.W, and Pike. R. The Practice of Programming. Addison-Wesley, Reading, MA, 1999. (Керниган Б.В., Пайк P. Практика программирования. С.-Пб., “Невский Диалект”, 2001.)
28. Knuth, D.E. Estimating the efficiency of backtrack programs. Mathematics of Computation, vol. 29, 1975, 121-136.
29. Knuth, D.E. Big omicron and big omega and big theta. ACM SIGACT News, vol. 8, no. 2, 1976, 18-23.
30. Knuth, D.E. Selected Papers on Computer Science. CSLI Publications and Cam­bridge University Press, New York, 1996.
31. Knuth, D.E. ***The Art of Computer Programming,***, Volume 1: ***Fundamental Al­gorithms***, 3rd ed. Addison-Wesley, Reading, MA, 1997. (Дональд Э. Кнут. ***Искусство программирования, том 1. Основные алгоритмы, 3-е изд.*** М., ИД “Вильямс”, 2000.)
32. Knuth, D.E. ***The Art of Computer Programming***, Volume 2: ***Seminumerical Al­gorithms***, 3rd ed. Addison-Wesley, Reading, MA, 1998. (Дональд Э. Кнут. ***Искусство программирования, том 2. Получисленные алгоритмы, 3-е изд.*** М., ИД “Вильямс”, 2000.)
33. Knuth, D.E. ***The Art of Computer Programming***, Volume 3: ***Sorting and Search­ing***, 2nd ed. Addison-Wesley, Reading, MA, 1998. (Дональд Э. Кнут. ***Ис­кусство программирования, том 3. Сортировка и поиск, 2-е изд.*** М., ИД “Вильямс”, 2000.)
34. Knuth, D.E., Morris, Jr., J.H., and Pratt, V.R. Fast pattern matching in strings. SIAM Journal on Computing, vol. 5, no. 2, 1977, 323-350.
35. Kolman, B., and Beck, R.E. Elementary Linear Programming with Applications, 2nd ed. Academic Press, San Diego, 1995.
36. Kordemsky, B.A. The Moscow Puzzles. Charles Scribners’s Sons, New York, 1972.
37. Kruskal, J.B. On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society, vol. 7, 1956, 48-50.
38. Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B., editors. The Traveling Salesman Problem. John Wiley & Sons, New York, 1985.
39. Levin, L.A. Universal sorting problems. Problemy Peredachi Informatsii, vol. 9, no. 3, 1973, 115-116 (in Russian). English translation in Problems of Informa­tion Transmission, vol. 9, 265-266. (Левин Л.А., Универсальные проблемы сортировки. Проблемы передачи информации, том 9, 1 3, 1973, с. 115-116)
40. Levitin, A. Do we teach the right algorithm design techniques? In Proceedings SIGCSE ’99, New Orleans, 1999, 179-183.
41. Levitin, A., and Papalaskari, M-A. Using puzzles in teaching algorithms. SIGCSE ’02, Cincinnati, 2002.
42. Manber, U. Introduction to Algorithms: A Creative Approach. Addison-Wesley, Reading, MA, 1989.
43. Martello, S., and Toth, P. Knapsack Problems: Algorithms and Computer Imple­mentations. John Wiley & Sons, Chichester, England, 1990.
44. Miller, R., and Boxer, L. A Unified Approach to Sequential and Parallel Algo­rithms. Prentice Hall, Upper Saddle River, NJ, 2000.
45. Moret, B.M.E., and Shapiro, H.D. Algorithms from P to NP, Volume I: Design and Efficiency. Benjamin Cummings, Redwood City, CA, 1991.
46. Moscovich, I. ***1000 Play Thinks: Puzzles, Paradoxes, Illusions, and Games.*** Workman, New York, 2001.
47. Motwani, R., and Raghavan, P. Randomized Algorithms. Cambridge University Press, New York, 1995.
48. Neapolitan, R., and Naimipour, K. Foundations of Algorithms: With C++ Pseu­docode, 2nd ed. Jones and Bartlett, Sudbury, MA, 1998.
49. Nielsen, M.A., and Chuang, I.L. Quantum Computation and Quantum Informa­tion. Cambridge University Press, Cambridge, UK, 2000.
50. O’Connor, J.J., and Robertson, E.F. The MacTutor Histoiy of Mathematics Archive, June 1998, <http://www-history.mcs.st-andrews.ac.uk/> history/Mathematicians/Abel. html.
51. Pan, V.Y. Strassen’s algorithm is not optimal. In ***Proceedings ofNinteenth Annual IEEE Symposium on the Foundations of Computer Science.*** 1978, 166-176.
52. Papadimitriou, C.H. Computational Complexity. Addison-Wesley, Reading, MA,

1994.

1. Papadimitriou, C.H., and Steiglitz, K. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Englewood Cliffs, NJ, 1982.
2. Parberry, I. Problems on Algorithms. Prentice-Hall, Englewood Cliffs, NJ, 1995.
3. Polya, G. How to Solve It. Doubleday, Garden City, NY, 1957.
4. Preparata, EP, and Shamos, M.I. Computational Geometiy: An Introduction. Springer-Verlag, New York, 1985.
5. Price B., Baecker, R., and Small, I. A principled taxonomy of software visualiza­tion. Journal of Visual Languages and Computing, vol. 4, no. 3, 1993, 211-266.
6. Prim, R.C. Shortest connection networks and some generalizations. Bell System Technical Journal, vol. 36, no. 1, 1957, 1389-1401.
7. Purdom, P.W., Jr., and Brown, C. The Analysis of Algorithms. Holt, Rinehart and Winston, New York, 1985.
8. Rawlins, G.J.E. ***Compared to What? An Introduction to the Analysis of Algo­rithms.*** Computer Science Press, New York, 1991.
9. Reingold, E.M., Nievergelt, J., and Deo, N. Combinatorial Algorithms: Theoty and Practice. Prentice-Hall, Englewood Cliffs, NJ, 1977.
10. Rivest, R.L., Shamir, A., and Adleman, L.M. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, vol. 21, no. 2, 1978, 120-126.
11. Rosen, K.H. Discrete Mathematics and Its Applications, 4th ed. McGraw-Hill, New York, 1999.
12. Rosenkrantz, D.J., Steams, R.E., and Lewis, P.M. An analysis of several heuristics for the traveling salesman problem. SIAM Journal of Computing, vol. 6, 1977, 563-581.
13. Sahni, S. Approximation algorithms for the 0/1 knapsack problem. Journal of the ACM, vol. 22, 1975, 115-124.
14. Sahni, S., and Gonzalez, T. P-complete approximation problems. Journal of the ACM, vol. 23, 1976, 555-565.
15. Sayood, K. Introduction to Data Compression, 2nd ed. Morgan Kaufmann Pub­lishers, San Francisco, CA, 2000.
16. Sedgewick, R. Algorithms, 2nd ed. Addison-Wesley, Reading, MA, 1988.
17. Sedgewick, R. Algorithms in C, 3rd ed. Parts 1-4: Fundamentals, Data Structures, Sorting, Searching. Addison-Wesley, Reading, MA, 1998.
18. Sedgewick, R. Algorithms in C, 3rd ed. Part 5: Graph Algorithms, Addison- Wesley, Boston, 2002.
19. Sedgewick, R., and Flajolet, P. An Introduction to the Analysis of Algorithms. Addison-Wesley, Reading, MA, 1996.
20. Shasha, D., and Lazere, C. ***Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists.*** Copernicus, New York, 1998.
21. Shell, D.L. A high-speed sorting procedure. Communications of the ACM, vol. 2, no. 7, July 1959, 30-32.
22. Shen, A. Algorithms and Programming: Problems and Solutions. Birkbauser, Boston, 1997.
23. Shor, P.W. Algorithms for quantum computation: Discrete algorithms and fac­toring. Proceedings of the 35th Annual Symposium on Foundations of Com­puter Science (Shafi Goldwasser, editor). IEEE Computer Society Press, 1994, 124-134.
24. Sipser, M. Introduction to the Theoty of Computation. PSW Publishing, Boston, 1997.
25. Skiena, S.S. Algorithm Design Manual. Springer-Verlag, New York, 1998.
26. Stasko, J. TANGO: A framework and system for algorithm animation. Computer, vol. 23, no. 9, 1990, 27-39.
27. Strassen, V. Gaussian elimination is not optimal. Numerische Mathematik, vol. 13, 354-356.
28. Tarjan, R.E. Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, 1983.
29. Tarjan, R.E., and van Leeuwen, J. Worst-case analysis of set union algorithms. Journal of the ACM, vol. 31, no. 2, 1984, 245-281.
30. Tarjan, R.E. Amortized computational complexity. SIAM Journal on Algebraic and Discrete Methods, vol. 6, no. 2, 1985, 306-318.
31. Taijan, R.E. Algorithm design. Communications of the ACM, vol. 30, no. 3, 1987, 204-212.
32. Tucker, A. Applied Combinatorics. John Wiley & Sons, New York, 1980.
33. Warshall, S. A theorem on boolean matrices. Journal of the ACM, vol. 9, no. 1, 1962, 11-12.
34. Weide, B. A survey of analysis techniques for discrete algorithms. Computing Surveys, vol. 9, no. 4, 1977, 291-313.
35. Weiss, M.A. Data Structures and Algorithm Analysis, Benjamin/Cummings, 1991.
36. Williams, J.W.J. Algorithm 232 (heapsort). Communications of the ACM, vol. 7, no. 6, 1964, 347-348.
37. Wirth, N. Algorithms + Data Structures = Programs. Prentice-Hall, Englewood Cliffs, NJ, 1976. (Вирт H. Алгоритмы + структуры данных = программы. — М.: Мир, 1985.)
38. Yao, R Speed-up in dynamic programming. SIAM Journal on Algebraic and Discrete Methods, vol. 3, no. 4, 1982, 532-540.

Указания к упражнениям

Глава 1 **Упражнения 1.1**

1. Вероятно, быстрее ъсего выполнить поиск в Web, но может помочь и ваша библиотека.
2. Легко найти аргументы как “за”, так и “против”. Есть один основополага­ющий принцип — научные факты или математические выражения не могут быть запатентованы (как вы думаете, применим ли он в данном случае?). Но должен ли этот принцип препятствовать патентованию всех алгоритмов?
3. Вы можете считать, что составляете алгоритм для человека, а не для машины. Тем не менее убедитесь, что описание не содержит явных неоднозначностей. У Кнута в ([65], с. 6) имеется интересное сравнение алгоритмов с кулинар­ными рецептами.
4. а) Просто следуйте описанному в тексте алгоритму Евклида.

б) Сравните количество делений, выполненных обоими алгоритмами.

1. Докажите, что если d является делителем как т, так и п (т.е. т = sd и п = = td для некоторых натуральных s и £), то оно является и делителем как п, так и г — т mod п, и наоборот. Воспользуйтесь формулой т — qn + г (О ^ г < п) и тем фактом, что если d является делителем и и г», то оно является делителем и + v и и — v (почему?).
2. Выполните одну итерацию алгоритма для двух произвольно выбранных чи­сел т < п.
3. Ответ на часть а) можно получить непосредственно; ответ на часть б) можно получить, исследуя производительность алгоритма для всех пар 1 < т < < п ^ 10.
4. а) Воспользуйтесь тождеством gcd (m, п) = gcd (т — п, п) для т ^ п > 0. б) Вопрос заключается в том, чтобы вычислить количество различных чисел,

которые могут быть написаны на доске, начиная с пары т > п ^ 1. Вы

должны воспользоваться связью этого вопроса с вопросом из части а). Рассмотрите несколько небольших примеров, в частности, для п = 1 и п = 2 — это должно вам помочь.

1. Имеется достаточно простой алгоритм, основанный на определении [\/nJ.

**Упражнения 1.2**

1. Крестьянину придется совершить несколько поездок через реку, начиная с единственно возможной.
2. В отличие от древней народной головоломки, первый ход в данном случае неочевиден.
3. Главным вопросом в данном случае является возможная неоднозначность.
4. Ваш алгоритм должен корректно работать для всех возможных значений коэффициентов, включая ноли.
5. Вы уже должны были встречаться с этим алгоритмом в одном из учебных курсов по программированию. Если вам удалось не встретиться с этим алго­ритмом ранее, попробуйте либо самостоятельно разработать его, либо найти его описание в литературе.
6. Возможно, вам стоит прогуляться к ближайшему банкомату, чтобы освежить свою память.
7. Вопрос а) достаточно сложен, хотя ответ на него, найденный в 1760-х го­дах немецким математиком Иоганном Ламбертом (Johann Lambert), хорошо известен. Вопрос б) существенно проще.
8. Думаю, вам известно не менее двух алгоритмов сортировки массива чисел.
9. Можно уменьшить количество итераций внутреннего цикла, ускорить работу цикла (как минимум для некоторых входных данных) или, что существенно важнее, разработать более быстрый алгоритм с нуля.

**Упражнения 1.3**

1. Отследите работу алгоритма для предоставленных входных данных. Вос­пользуйтесь при ответе на вопросы определениями устойчивости и обмен­ного алгоритма, приведенными в тексте раздела.
2. Если вы не можете вспомнить ни одного алгоритма поиска, вам придет­ся самостоятельно разработать его. (Не поддавайтесь искушению заглянуть в конец книги.)
3. Такой алгоритм будет приведен далее в книге, но его разработка не должна вызвать у вас проблем.
4. Если вы не сталкивались с этой задачей ранее, можете поискать ответ в Web или в книге по дискретным структурам. Ответ удивительно прост.
5. Не существует эффективного алгоритма решения подобной задачи на про­извольном графе. Однако найти цепь Гамильтона для данного конкретно­го графа несложно (вам надо найти только одну из множества возможных цепей).
6. а) Представьте себя на месте пассажира и спросите сами себя, что для вас

означает “наилучший маршрут”. Затем подумайте о других пассажирах, с иными требованиями к маршруту, б) Представить задачу в виде графа очень просто. Подумать надо только над станциями пересадок.

1. а) Что собой представляет маршрут коммивояжера с точки зрения комбина­

торики?

б) Представляется вполне естественным рассматривать вершины одного цве­та как элементы одного и того же подмножества.

1. Создайте граф, вершины которого соответствуют областям карты. Чему со­ответствуют ребра графа, — решите сами.
2. Предположите, что данная окружность существует, и начните с поиска ее центра. Не забудьте о частном случае п ^ 2.
3. Будьте внимательны и не забудьте о каком-нибудь из частных случаев данной задачи.

**Упражнения 1.4**

1. а) Воспользуйтесь тем, что массив не отсортирован.

б) Мы использовали этот трюк при реализации одного из алгоритмов в раз­деле 1.1.

1. а) В случае отсортированного массива имеется алгоритм, о котором вы, без

сомнения, уже слышали, б) Неуспешный поиск может быть сделан более быстрым.

1. a) push(x) помещает х на вершину стека, a pop снимает элемент с вершины

стека.

б) enqueue(x) добавляет х в хвост очереди, a dequeue удаляет элемент из ее головы.

1. Воспользуйтесь определениями соответствующих свойств графа и исполь­зуемой структуры данных.
2. Есть два хорошо известных алгоритма, способных решить поставленную задачу Первый использует стек, а второй — очередь. Хотя эти алгоритмы рассматриваются в книге немного позже, не упустите шанс разработать их самостоятельно!
3. Неравенство h ^ п — 1 следует непосредственно из определения высоты дерева. Нижняя граница неравенства следует из неравенства 2h+l — 1 ^ тг, которое можно доказать, рассматривая наибольшее количество вершин, ко­торое может иметь бинарное дерево высотой h.
4. Вы должны указать, как реализовать каждую из трех операций очереди с при­оритетами.
5. Из-за вставок и удалений использование (отсортированного или несортиро­ванного) массива элементов словаря — не лучшее из возможных решений.
6. Для ответа на один из поставленных вопросов вам требуется знать о пост­фиксной записи (если вы с ней незнакомы — прогугяяйтесь в Internet).
7. Имеется несколько алгоритмов, решающих поставленную задачу. При реше­нии не забывайте о том, что в слове может содержаться несколько одинако­вых букв.

Глава 2

**Упражнения 2.1**

1. Вопросы и в самом деле такие простые, какими кажутся, хотя некоторые из них могут иметь различные ответы. Кроме того, не забывайте о том, как измеряется размер целых чисел.
2. а) Сумма двух матриц определяется как матрица, элементы которой пред­

ставляют собой суммы соответствующих элементов в матрицах-слагае­мых.

б) Умножение матриц требует применения двух операций — умножения и сложения. Какая из них должна рассматриваться как основная и по­чему?

1. Будет ли изменяться эффективность алгоритма для разных входных данных одного и того же размера?
2. а) Перчатки — не носки: они могут быть левыми и правыми.

б) У вас есть только две качественно разные возможности. Подсчитайте количество способов получить каждую из них.

1. а) Сначала докажите, что если положительное десятичное целое число п

имеет b цифр в двоичном представлении, то

26-1 ***^п<2ь.***

Затем возьмите логарифм по основанию 2 от всех частей данного нера­венства.

б) Формула будет точно такой же, с небольшим уточнением для учета дру­гого основания системы счисления.

в) Каким образом перейти от логарифма по одному основанию к логарифму по другому?

1. Добавьте проверку, не отсортирован ли уже исходный массив.
2. Аналогичный вопрос был разобран в тексте раздела.
3. Воспользуйтесь разностью либо отношением / (4п) и / (п), что более под­ходит для получения компактного ответа. (Если возможно, попытайтесь по­лучить ответ, не зависящий от п.)
4. При необходимости упростите функцию, сведя ее к одному члену, определя­ющему порядок роста с точностью до постоянного множителя. (Формальные методы ответа на такие вопросы мы рассмотрим в следующем разделе; од­нако на данный вопрос можно ответить и без знания таких методов.)
5. Воспользуйтесь формулой Yl?=o 21 — 2n+1 — 1.

**Упражнения 2.2**

1. Воспользуйтесь соответствующими количествами основных операций (см. раздел 2.1) и определением обозначений О, 0 и П.
2. Сначала определите порядок роста п(п+ 1)/2, а затем воспользуйтесь не­строгим определением обозначений О, 0 и Q (подобный пример можно найти в тексте раздела).
3. Упростите данные функции до одного члена, определяющего порядок роста.
4. а) Внимательно просмотрите соответствующие определения.

б) Вычислите пределы для каждой пары соседних функций в списке.

1. Сначала упростите некоторые из заданных функций. Затем воспользуйтесь списком функций в табл. 2.2 в качестве “привязки” для каждой из функций. Докажите корректность размещения функций, вычислив соответствующие пределы.
2. а) Вы можете доказать данное утверждение, либо вычисляя соответствую­

щий предел, либо применяя математическую индукцию,

б) Вычислите lim а^/аЦ.

п—юс

1. Докажите корректность а), б) и в) при помощи соответствующих опреде­лений. Для г) приведите контрпример (например, путем построения двух функций, которые по-разному ведут себя для четных и нечетных значений аргументов.
2. Доказательство а) аналогично приведенному в разделе. Само собой, для огра­ничения функции снизу должны использоваться другие неравенства.
3. Следуйте плану анализа, использовавшемуся в тексте раздела, когда был впервые упомянут данный алгоритм.
4. Вы должны поочередно отходить от исходного положения то влево, то впра­во, пока не обнаружите дверь.

Упражнения 2.3

1. Воспользуйтесь формулами и правилами, приведенными в приложении А. Возможно, вам придется выполнить простые алгебраические операции перед их применением.
2. Найдите среди сумм в приложении А суммы, выглядящие так же, как и сум­мы из этого упражнения, и попытайтесь привести суммы из упражнения к виду сумм из приложения А. Заметим, что для выяснения порядка роста суммы не требуется вычислять ее точное значение.
3. Просто следуйте приведенным в задании формулам.
4. а) Отследите выполнение алгоритма для нескольких небольших значений п

(например, п = 1,2,3) — это должно помочь справиться с задачей.

б) С этим вопросом вы уже встречались в примерах в тексте раздела.

в) Следуйте плану, приведенному в тексте раздела.

г) Ответ должен немедленно следовать из ответа на пункт в). Кроме того, вы можете захотеть дать ответ как функцию числа битов в двоичном представлении числа п (почему?).

д) Вам не приходилось встречаться с этой суммой где-нибудь еще, кроме этой книги?

1. а) Отследите выполнение алгоритма для нескольких небольших значений п

(например, п = 1,2,3) — это должно помочь справиться с задачей.

б) С этим вопросом вы уже встречались в примерах в тексте раздела.

в) Вы можете следовать плану, приведенному в тексте раздела, либо ответить на вопрос сразу (попробуйте оба метода).

г) Ответ должен немедленно следовать из ответа на пункт в).

д) Должен ли алгоритм всегда делать два сравнения на каждой итерации? Эта мысль может получить дальнейшее развитие и привести к значитель­ному усовершенствованию алгоритма. Примените ее к двухэлементному массиву, а затем обобщите. Но можем ли мы надеяться получить алгоритм с эффективностью, превышающей линейную?

1. а) Элементы А [г, j] и A [j, г] симметричны по отношению к главной диаго­

нали матрицы.

б) Имеется только один кандидат.

в) Вы можете исследовать только наихудший случай.

г) Ответ должен немедленно следовать из ответа на пункт в).

д) Сравните решаемую алгоритмом задачу с тем, каким образом он это де­лает.

1. Вычисление суммы п чисел может быть выполнено при помощи п — 1 сложе­ний. Сколько сложений выполняет алгоритм при вычислении каждого эле­мента матрицы-произведения?
2. Воспользуйтесь формулой, которая связывает п и количество битов в его двоичном представлении.
3. При доказательстве по индукции воспользуйтесь формулой

П 71—1

У^г = ^г + п.

г=1 г=1

Юный Гаусс заметил, что сумму 1 + 2 Н Ь 99 + 100 можно вычислить как

сумму 50 пар, имеющих одинаковое значение суммы.

1. а) Запись сумм не должна вызывать сложностей. Однако использование стандартных формул и правил суммирования потребует больших усилий, чем в предыдущих примерах,

б) Оптимизируйте внутренний цикл алгоритма.

Упражнения 2.4

1. Каждое из приведенных уравнений можно решить с помощью метода обрат­ной подстановки.
2. Интересующее нас рекуррентное уравнение практически идентично рекур­рентному соотношению для количества умножений, которое было решено в данном разделе.
3. а) Поставленный вопрос идентичен вопросу об эффективности рекурсивно­

го алгоритма вычисления факториала,

б) Напишите псевдокод нерекурсивного алгоритма и определите его эффек­тивность.

1. а) Обратите внимание, что в упражнении спрашивается о рекуррентном

уравнении для значений вычисляемой функции, а не для количества вы­полняемых операций. Для его составления следуйте представленному

в условии псевдокоду. Проще всего найти решение полученного урав­нения при помощи метода обратных подстановок (см. приложение Б).

б) Этот вопрос очень похож на уже рассмотренный нами.

в) Вы можете захотеть включить операции вычитания, необходимые для уменьшения значения п.

1. а) Воспользуйтесь формулой для общего количества перемещений дисков,

которая была приведена в тексте раздела.

б) Решите задачу для трех дисков, чтобы выяснить, сколько перемещений совершает каждый диск. Затем обобщите свои наблюдения и докажите их корректность для общего случая п дисков.

в) Если вы не справитесь с этим заданием, не расстраивайтесь: хотя ал­горитм достаточно прост, его не так-то легко разработать. В качестве утешения — поищите решение в Web.

1. а) Рассмотрите отдельно случаи четных и нечетных значений п и покажите,

что для обоих случаев [log2 п\ удовлетворяет рекуррентному уравнению и начальному условию, б) Просто следуйте псевдокоду алгоритма.

1. а) Воспользуйтесь формулой 2п = 2п~1 + 2n\_1, не упрощая ее. Не забудьте

об условии прекращения рекурсивных вызовов.

б) Подобный алгоритм рассматривался в разделе 2.4.

в) Подобный вопрос рассматривался в разделе 2.4.

г) Плохой класс эффективности алгоритма еще не означает, что плох сам алгоритм. Например, классический алгоритм для задачи о Ханойских башнях оптимален, несмотря на его экспоненциальную эффективность. Таким образом, утверждение о том, что конкретный алгоритм неэффек­тивен, должно содержать указание на более эффективный алгоритм.

1. а) Вам должно помочь отслеживание работы алгоритма при п = 1 и п = 2. б) Очень похожий вопрос обсуждался в одном из примеров в данном разделе.
2. Вам должно помочь отслеживание работы алгоритма при п = 1 и п = 4.
3. а) Воспользуйтесь формулой из определения детерминанта, чтобы получить рекуррентное уравнение для количества умножений, выполняемых алго­ритмом.

б) Исследуйте правую часть рекуррентного соотношения. Вычисление не­скольких первых значений М (п) также может помочь вам в решении задачи.

Упражнения 2.5

1. Будет проще подставить в рекуррентное соотношение фп и фп по отдельно­сти. Почему этого достаточно?
2. Воспользуйтесь приближенной формулой для F (п), чтобы найти минималь­ные значения п, при которых F (п) превышает заданные числа.
3. Имеется несколько способов решения этой задачи. Наиболее элегантный из них позволяет поместить эту задачу в данный раздел.
4. Постройте рекуррентные соотношения для С (п) и Z (п), само собой, с со­ответствующими начальными условиями.
5. Вся информация, необходимая для очередной итерации алгоритма, — это значения двух последних последовательных чисел Фибоначчи. Измените ал­горитм так, чтобы воспользоваться данным фактом.
6. При доказательстве воспользуйтесь методом математической индукции.
7. Рассмотрите небольшой пример; скажем, вычисление gcd (13,8).
8. а) Проведите доказательство методом математической индукции.

б) Знаменитые авторы Конкретной математики — Р. Грэхем, Д. Кнут и О. Паташник — отмечают в своей широко известной книге по дис­кретной математике [45], что это была одна из любимых головоломок Льюиса Кэролла (Lewis Carroll). К сожалению, они также пишут: “Этот парадокс объясняется тем, что... — впрочем, волшебство объяснять не принято”. Я не могу идти против таких выдающихся ученых, и должен воздержаться от подсказки.

1. Последние к цифр числа N можно получить путем вычисления N mod 10fc. Выполнение всех операций вашего алгоритма по модулю 10^ (см. приложе­ние А) позволит справиться с экспоненциальным ростом чисел Фибоначчи. Попутно заметим, что раздел 2.6 посвящен эмпирическому анализу алго­ритмов.

Упражнения 2.6

1. Будет ли получено корректное количество сравнений для всех массивов раз­мером 2?
2. Начните с отладки счетчика сравнений и генератора массива входных дан­ных.
3. В случае сравнительно быстрой машины вы можете получить нулевое время работы, по крайней мере для малых размеров входных данных. В разделе 2.6 рассказано, как справиться с этой неприятностью.
4. Проверьте, насколько быстро растет количество операций при удвоении раз­мера данных.
5. Подобный вопрос рассматривался в тексте раздела.
6. Сравните значения функций lglgn и lgn при п = 2к.
7. Вставьте счетчик числа делений в программу, реализующую алгоритм, и вы­полните программу для входных пар чисел из указанного диапазона.
8. Получите эмпирические данные для случайных значений п в диапазоне, скажем, от 102 до 104 или 105 и постройте график по полученным данным (возможно, вы захотите использовать различные масштабы по разным осям системы координат).

Глава 3 Упражнения 3.1

1. а) Подумайте об алгоритмах, которые поразили вас своей эффективностью

и/или интеллектуальностью. Ни одна из этих характеристик не является признаком алгоритма, основанного на методе грубой силы, б) Интересно, что на этот вопрос ответить не так-то легко. Источником тре­бующихся примеров задач могут послужить математические задачи, с ко­торыми вы сталкивались на лекциях по высшей математике.

1. а) Ответ на первый вопрос, по сути, был дан в тексте раздела. Ответ легко

выразить как функцию от числа битов, если воспользоваться формулой, связывающей две эти метрики,

б) Как можно вычислить (ab) mod ml

1. Для начала надо выполнить указанные задания.
2. а) Наиболее простой алгоритм, основанный на непосредственной подста­

новке значения хо в формулу для р (ж), имеет квадратичное время работы.

б) Анализ излишних вычислений в квадратичном алгоритме должен приве­сти вас к разработке линейного алгоритма вычисления полинома.

в) Сколько коэффициентов имеет полином степени п! Можно ли вычислить значение полинома в произвольной точке, не обращаясь ко всем им?

1. Просто пошагово пройдите алгоритм для приведенных входных данных (в начале раздела это было сделано для других входных данных).
2. Хотя большинство элементарных алгоритмов сортировки устойчивы, не спе­шите с ответом.
3. Вообще говоря, реализация алгоритма для связанных списков усложняется, если алгоритм требует обращения к элементам списка не в последовательном порядке.
4. Просто пошагово пройдите алгоритм для приведенных входных данных (в начале раздела это было сделано для других входных данных).
5. а) Список является отсортированным тогда и только тогда, когда все его

соседние элементы находятся в надлежащем порядке. Почему?

б) Добавьте логический флаг для регистрации наличия или отсутствия об­менов.

в) Начните с выяснения, какие входные данные приводят к наихудшему случаю.

1. Может ли пузырьковая сортировка изменить порядок двух одинаковых эле­ментов во входных данных?

Упражнения 3.2

1. Модифицируйте анализ версии алгоритма в разделе 2.1.
2. Какой вид имеет функция Cavg (р)?
3. См. дельные советы в разделе 2.6.
4. Цитата Махатмы Ганди стоит того, чтобы подумать не только над упражне­нием, но и над ней.
5. Для каждого из входных данных одна итерация алгоритма дает всю необхо­димую для ответа информацию.
6. Будет достаточно, если вы ограничитесь бинарными текстами и шаблонами.
7. Измените псевдокод алгоритма поиска подстрок, приведенный в данном раз­деле. На сколько символов следует выполнить сдвиг после того, как искомая подстрока найдена?
8. Для нас представляется естественным выполнять проверку символов слева направо только потому, что это способ чтения и записи, принятый в евро­пейских и ряде других языков.
9. При работе методом грубой силы вы можете блеснуть тем, что не будете проверять слова в тех направлениях, в которых букв недостаточно для раз­мещения искомого слова.
10. Алгоритм с использованием (очень) грубой силы может просто обстреливать все клетки подряд, начав, например, с одного из углов поля. Не могли бы вы предложить лучшую стратегию? (Вы бы могли исследовать относитель­ные эффективности разных стратегий, создав две программы, реализующие их, и заставив их играть друг с другом.) Оказалась ли предложенная вами стратегия лучше стратегии случайного обстрела клеток поля противника?

Упражнения 3.3

1. Сортировка п действительных чисел требует О (п log п) времени.
2. а) Убедитесь в выполнении требований 2) и 3) при помощи основных свойств

абсолютных значений.

б) При использовании манхэттенского расстояния интересующие нас точки задаются уравнением \х — 0| + |у — 0| = 1. Вы можете изобразить со­ответствующие точки в первом квадранте, в котором все точки имеют положительные координаты (х, у ^ 0), а затем воспользоваться симмет­рией.

в) Предположение ложно. Возьмите две точки, Pi (0,0) и P<i (1,0), и найдите точку Рз, которая завершает контрпример.

1. Ваш ответ должен быть функцией двух параметров: пик. Частный случай к = 2 был рассмотрен в тексте раздела.
2. Обратитесь к приведенным в разделе примерам.
3. Может ли количество угловых точек выпуклой оболочки множества из п различных точек быть равным п? А нулю?
4. Найти некоторые из угловых точек выпуклой оболочки проще, чем другие.
5. Если на прямой линии, проведенной между точками Pi и Pj, располагаются и другие точки, то какие из них должны быть сохранены для последующей обработки?
6. Ваша программа должна работать с произвольным множеством из п различ­ных точек, включая множества, в которых на одной прямой может находиться более двух точек.
7. а) Множество точек, удовлетворяющих неравенству ах + by ^ с, представ­

ляет собой полуплоскость, располагающуюся с одной стороны от прямой линии ах + by = с, включая точки на самой линии. Изобразите такие полуплоскости для каждого их неравенств и найдите их пересечение.

б) Угловые точки представляют собой вершины многоугольника, получен­ные в части а) данного упражнения.

в) Вычислите и сравните значения целевой функции в угловых точках.

Упражнения 3.4

1. а) Определите основную операцию алгоритма и подсчитайте количество ее

выполнений.

б) Для каждого из приведенных значений времени определите наибольшее п, для которого этот предел не будет перейден.

1. Чем задача коммивояжера отличается от задачи поиска гамильтонова цикла?
2. Ваш алгоритм должен проверять хорошо известное условие, необходимое и достаточное для существования эйлерова цикла в связанном графе.
3. Сгенерируйте оставшиеся 4! — 6 = 18 возможных назначений, вычислите их стоимости и найдите среди полученных значений минимальное.
4. Приведите контрпример как можно меньшего размера.
5. Венгерский метод обычно рассматривается в книгах, посвященных иссле­дованию операций. Узнайте у преподавателя, следует ли вам реализовать данный алгоритм в виде компьютерной программы как часть вашего рефе­рата.
6. Измените условие задачи так, чтобы надо было искать сумму элементов только в одном подмножестве, а не в двух.
7. Следуйте определению клики и алгоритма на основании исчерпывающего перебора.
8. Рассмотрите все возможные перестановки данных элементов.
9. а) Просуммируйте все элементы магического квадрата двумя различными способами.

б) Какие комбинаторные элементы должны быть сгенерированы при ответе на этот вопрос?

Глава 4 Упражнения 4.1

1. В нескольких аспектах этот вопрос сходен с вопросом о вычислении суммы методом декомпозиции. Подобный алгоритм рассматривался и в упражнени­ях 2.4.
2. В отличие от задачи 1 декомпозиционный алгоритм может быть более эффек­тивным, чем алгоритм на основе грубой силы, за счет меньшего постоянного множителя.
3. Каким образом вы вычислите а8 путем решения задачи о возведении в сте­пень размера 4? А а9?
4. Взгляните на обозначения, используемые в формулировке теоремы.
5. Примените Основную теорему.
6. Пошагово выполните алгоритм, как это было сделано для другого примера в тексте раздела.
7. Как сортировка слиянием может изменить относительный порядок двух эле­ментов?
8. а) Как обычно, воспользуйтесь обратной подстановкой.

б) Какие входные данные минимизируют количество сравнений ключей, вы­полняемых сортировкой слиянием? Сколько сравнений выполнит сорти­ровка слиянием для этих данных на этапе слияния?

в) Не забудьте включить перемещения ключей как перед разбиением, так и в процессе слияния.

1. Метод декомпозиции требует, чтобы исходный экземпляр задачи был сведен к нескольким меньшим экземплярам той же задачи.

Упражнения 4.2

1. Мы уже отслеживали в этом разделе работу алгоритма для других входных данных.
2. Еще раз обратитесь к описанию процесса разбиения.

а) Воспользуйтесь условиями прекращения сканирования.

б) Воспользуйтесь условиями прекращения сканирования.

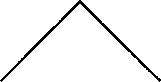
в) Рассмотрите массив, состоящий из одинаковых элементов.

1. Определение устойчивости алгоритма сортировки было приведено в раз­деле 1.3. В общем случае алгоритмы, которые используют обмен далеко отстоящих друг от друга элементов, устойчивыми не являются.
2. Пошагово проследите за работой алгоритма, чтобы увидеть, для каких вход­ных данных индекс г выходит за пределы границы.
3. Рассмотрите, как именно ведет себя быстрая сортировка с указанными мас­сивами. Естественно, ваш ответ должен основываться на количестве выпол­няемых сравнений.
4. В каком месте будет выполняться разделение для описанных в упражнении входных данных?
5. Для решения этого рекуррентного соотношения следует воспользоваться стандартными приемами решения рекуррентных уравнений. Это решение можно найти в большинстве книг, посвященных разработке и анализу алго­ритмов.
6. Воспользуйтесь идеей, лежащей в основе разбиения.
7. Просто выполните указанное в упражнении задание, и не забудьте, что про­верку следует выполнять на входных данных в широком диапазоне их раз­меров.
8. Воспользуйтесь идеей, лежащей в основе разбиения.

Упражнения 4.3

1. а) Воспользуйтесь формулой, непосредственно дающей требуемый ответ, б-г) Воспользуйтесь бинарным деревом, отображающим операции алгорит­ма при поиске произвольного ключа. Первые три узла данного дерева для приведенного в условии экземпляра задачи будут иметь следующий вид:

6(55)



2(27) 9(81)

(Первое число в узле показывает индекс т элемента массива, с которым выполняется сравнение, а число в скобках указывает значение сравнива­емого элемента, т.е. А [га]).

1. Если вам требуется освежить память, — обратитесь к разделу 2.4, где уже решались подобные уравнения, а также к приложению Б.
2. а) Воспользуйтесь тем фактом, что п ограничено снизу и сверху степенями

двойки, т.е. 2к ^ п < 2/с+1. б) Случай четного п(п = 2г) рассмотрен в тексте раздела. Для случая нечет­ного п (п = 2г + 1) выполните подстановку в обе части уравнения и пока­жите их тождественность. При решении вам может пригодиться формула из части а). Не забудьте также проверить выполнение начальных условий.

1. Оцените отношение среднего количества сравнения ключей при успешном последовательном поиске и при успешном бинарном поиске.
2. Как достичь среднего элемента связанного списка?
3. Найдите отдельно элементы, не меньшие L, и элементы, не превышающие U. Не забывайте, что ни L, ни U не обязаны быть элементами массива.
4. Вам может помочь диаграмма бинарного поиска, приведенная в тексте раз­дела.
5. Воспользуйтесь сравнением К ^ А [га], где т <— [(/ + г)/2\_|, пока I не равно г. После этого проверьте, найден ли искомый элемент.
6. Анализ практически идентичен анализу представленного в тексте раздела алгоритма бинарного поиска.
7. Перенумеруйте рисунки и используйте их номера в ваших вопросах.[[76]](#footnote-76)

Упражнения 4.4

1. Эта задача практически идентична рассмотренной в тексте раздела.
2. Это стандартное упражнение, которое вы, должно быть, не раз выполня­ли в процессе изучения структур данных. Определения вариантов обхода, приведенные в конце раздела, должны помочь вам пошагово выполнить со­ответствующие алгоритмы, даже если вы никогда не делали этого ранее.
3. Псевдокод просто отражает описание процесса обхода.
4. Сколько раз узел вносится в стек обхода? Могут ли все узлы одновременно оказаться в стеке?
5. Сравните алгоритм и определения обходов.
6. Если вы не знаете ответа на этот важный вопрос, можете просто посмотреть на результаты различных обходов небольшого бинарного дерева поиска. Для доказательства ответьте на вопрос: что можно сказать о двух узлах с ключами А\* и &2, если кг < /С2?
7. Модифицируйте алгоритм вычисления высоты бинарного дерева.
8. Выполните алгоритм пошагово для небольшого входного дерева.
9. Воспользуйтесь математической индукцией (как это было сделано при дока­зательстве равенства х = п + 1 в тексте раздела).

Упражнения 4.5

1. Попробуйте сначала ответить на вопрос при п = 2, а затем обобщить свой ответ для произвольного п.
2. Пошагово пройдите алгоритм для приведенных в условии входных данных. Конечно, в процессе вычислений вам придется применить этот же алгоритм для вычисления произведения двузначных чисел.
3. а) Прологарифмируйте обе части равенства.

б) Для чего используется точное решение рекуррентного уравнения?

1. а) Как выполняется умножение на степень 10?

б) Попробуйте повторить вывод для, например, произведения 98 • 76.

1. Подсчитайте количество сложений при умножении в столбик, например, двух четырехзначных чисел — это должно вам помочь.
2. Для проверки достаточно выполнить ряд простых алгебраических действий.
3. Пошагово пройдите алгоритм Штрассена для приведенных в условии вход­ных данных (количество работы могло бы оказаться существенно большим, если бы рекурсию требовалось остановить не при п — 2, а при п = 1). По завершении работы было бы неплохо сравнить результат с полученным при помощи алгоритма грубой силы.
4. Примените метод обратной подстановки для решения рекуррентного соот­ношения из текста раздела.
5. Рекуррентное уравнение для количества умножений в алгоритме Пана по­добно рекуррентному уравнению для алгоритма Штрассена. Для получения порядка роста его решения воспользуйтесь Основной теоремой.

Упражнения 4.6

1. а) Сколько точек требуется рассмотреть на стадии комбинации решения?

б) Разработайте более простой алгоритм, принадлежащий тому же классу эффективности.

1. Вспомните (см. раздел 4.1), что количество сравнений, выполняемых сорти­ровкой слиянием в наихудшем случае равно Cworst (п) = п bg2 п — п +1 при п = 2к. В требуемом от вас рекуррентном соотношении вы можете исполь­зовать только один член наивысшего порядка из приведенной формулы.
2. Ответ на вопрос а следует непосредственно из учебника по геометрии.
3. Воспользуйтесь формулой, связывающей определитель с площадью тре­угольника.
4. Вернитесь к определению множества Si в описании алгоритма.
5. Естественно, она должна быть П (тг). (Почему?)
6. Создайте последовательность п точек, для которых алгоритм при каждом рекурсивном вызове уменьшает размер задачи на единицу.

Глава 5 Упражнения 5.1

1. Решите задачу для п — 1.
2. Как найти наименьший элемент в массиве из п чисел, если вам известен наи­меньший среди его п— 1 первых элементов? Кроме того, заметим, что почти та же задача решалась одним из алгоритмов в упражнениях к разделу 2.4.
3. Воспользуйтесь тем фактом, что все подмножества n-элементного множе­ства S = {а,1,..., ап} могут быть разделены на две группы — те, которые содержат элемент ап, и те, которые его не содержат.
4. Пройдите алгоритм пошагово, как это было сделано в тексте для других входных данных (см. рис. 5.4).
5. а) Ограничитель должен предотвратить перемещение наименьшего элемента

за пределы массива, б) Примените анализ эффективности алгоритма, проведенный в тексте раз­дела, к версии с ограничителем.

1. Вспомните, что к элементам однократно связанного списка возможен только последовательный доступ.
2. Поскольку единственное отличие между представленными версиями алго­ритмов заключается в операциях во внутреннем цикле, вы должны оценить отличие времени выполнения одной итерации внутреннего цикла обоих ал­горитмов.
3. а) Ответ на вопрос для массива из трех элементов должен помочь ответить

на вопрос в общем случае, б) Положим для простоты, что все элементы различны и что вставка А [г] в каждую из г + 1 возможных позиций среди предшествующих элементов равновероятна. Вначале проанализируйте версию алгоритма с ограничи­телем.

1. Воспользуйтесь формулой для количества сравнений ключей алгоритмом би­нарного поиска в худшем случае (раздел 4.3) и соответствующей формулой суммирования из приложения А для определения порядка роста. Для рас­сматриваемого алгоритма класс эффективности определяет не данная опера­ция — а какая?
2. а) Заметим, что более удобно сортировать подфайлы параллельно, т.е. срав­нивать А [0] с A [hi], затем А [1] с А [1 + Ы] и т.д. б) Вспомним, что, вообще говоря, алгоритмы сортировки, в которых выпол­няется обмен далеких элементов, устойчивыми не являются.

Упражнения 5.2

1. а) Воспользуйтесь определениями матрицы смежности и связанных списков

смежности, приведенными в разделе 1.4. б) Выполните поиск в глубину точно так же, как это было сделано в тексте раздела (см. рис. 5.5).

1. Сравните классы эффективности двух версий поиска в глубину для разре­женного графа.
2. а) Чему равно количество деревьев в лесу?

б) Сначала ответьте на этот вопрос для связного графа.

1. Выполните поиск в ширину так же, как это было сделано в тексте раздела (см. рис. 5.6).
2. Вы можете воспользоваться тем фактом, что уровень вершины в дереве по­иска в ширину указывает количество ребер в кратчайшем пути от корня до этой вершины.
3. а) Какое свойство леса поиска в ширину указывает наличие циклов? (Ответ

аналогичен ответу на этот же вопрос для поиска в глубину.) б) Ответ отрицательный. Найдите два примера, подтверждающие это.

1. Исходя из того факта, что при любом из поисков новая вершина может быть достигнута тогда и только тогда, когда она смежна одной из ранее посе­щенных вершин, какие вершины будут посещены к моменту прекращения обхода (т.е. когда стек или очередь обхода оказываются пустыми)?
2. Воспользуйтесь для решения частей а и б упражнения лесом поиска, соот­ветственно, в глубину и в ширину.
3. Используйте в программе поиск в глубину или в ширину.
4. а) Строго следуйте инструкциям из условия упражнения.

б) Использование обоих способов должно быстро привести вас к выходу из лабиринта.

Упражнения 5.3

1. Пошагово выполните алгоритм так, как это было сделано в тексте раздела для другого ориентированного графа (см. рис. 5.10).
2. а) Вам надо доказать два утверждения: 1) если ориентированный граф име­

ет ориентированный цикл, то задача топологической сортировки такого ориентированного графа не имеет решения; и 2) если ориентированный граф не содержит ориентированных циклов, то задача топологической сортировки такого ориентированного графа разрешима,

б) Рассмотрите вырожденный случай ориентированного графа.

1. а) Как временная эффективность алгоритма топологической сортировки на

основе поиска в глубину связана с временной эффективностью поиска в глубину?

б) Знаете ли вы длину списка, который будет сгенерирован алгоритмом? Где вы должны разместить, например, первую снимаемую со стека обхода в глубину вершину, чтобы она находилась в своей окончательной пози­ции?[[77]](#footnote-77)

1. Попробуйте сделать это для одного-двух небольших примеров ориентиро­ванных графов.
2. Пошагово выполните алгоритм, так, как это было сделано в тексте раздела для другого ориентированного графа (см. рис. 5.11).
3. а) Воспользуйтесь доказательством от противного.

б) Если у вас возникли трудности при ответе на этот вопрос, рассмотрите пример ориентированного графа с вершиной, в которую не входит ни одно ребро, и запишите его матрицу смежности.

в) Ответ следует из определений источника и связанных списков смежности.

1. Для каждой вершины храните количество ребер, входящих в эту вершину в остающемся после удаления источника подграфе. Используйте очередь вершин-источников.
2. а) Пошагово выполните алгоритм для приведенных в условии входных дан­ных, строго следуя описанным шагам алгоритма,

б) Определите эффективность каждого из трех основных шагов алгоритма, а затем определите его общую временную эффективность. Разумеется, ответ зависит от способа представления графа — при помощи матрицы смежности или с использованием связанных списков смежности.

1. Можно воспользоваться известным алгоритмом обхода, хотя это и не самый эффективный способ решения задачи.

Упражнения 5.4

1. Воспользуйтесь стандартными формулами для количества упомянутых ком­бинаторных объектов. Для простоты можно считать, что генерация комбина­торного объекта занимает то же время, что и, например, присваивание.
2. Примеры работы этих алгоритмов (для множества меньшего размера) рас­сматривались в тексте раздела.
3. Набросок данного алгоритма приведен в тексте раздела.
4. а) Пошагово пройдите алгоритм при п — 2; воспользуйтесь полученными

результатами при проходе при п — 3, а затем — полученными при послед­нем проходе результатами при п — 4.

б) Покажите, что алгоритм генерирует п! перестановок и что все они раз­личны. Воспользуйтесь методом математической индукции.

в) Запишите рекуррентное соотношение для количества обменов, выполня­емых алгоритмом. Найдите его решение и порядок роста этого решения. Вам может потребоваться формула е ^ Y17=o (1/г!),

1. Примеры работы этих алгоритмов (для задач меньшего размера) рассматри­вались в тексте раздела.
2. После пояснений трюки становятся неинтересны...
3. Это несложное упражнение, поскольку способ получения битовых строк длиной п из битовых строк длиной п — 1 очевиден.
4. Вы можете имитировать бинарное сложение без явного его использования.
5. Код Грея для п = 3 приведен в конце раздела. Нетрудно увидеть, каким образом можно воспользоваться имеющимся кодом для генерации кода Грея при п = 4. Коды Грея имеют полезную геометрическую интерпретацию, ос­нованную на отображении их битовых строк на вершины n-мерного куба. Найдите такое отображение для п = 1, 2 и 3. Эта геометрическая интерпре­тация может помочь вам при разработке алгоритма для генерации кода Грея произвольного порядка п.
6. Для решения поставленной задачи имеется несколько алгоритмов, использу­ющих метод уменьшения размера задачи. Они несколько тоньше, чем можно было бы ожидать. Генерация комбинаций в предопределенном порядке (воз­растающем, убывающем, лексикографическом) может помочь как при раз­работке алгоритма, так и при доказательстве корректности. Очень полезно также следующее простое свойство. Предположим (без потери общности), что исходное множество — {1,2,..., п}. Тогда имеется C^Z} к-подмножеств, наименьший элемент которых — г (г = 1,..., п — к + 1).

Упражнения 5.5

1. Если экземпляр задачи размером п состоит в вычислении [log2 nj, то что из себя представляет экземпляр размером п/2? Каким соотношением связаны решения этих двух экземпляров?
2. Алгоритм, конечно, очень похож на бинарный поиск. Сколько сравнений ключей выполняется в одной итерации в наихудшем случае и какая часть массива остается для дальнейшего поиска?
3. Совершенно очевидно, как следует поступать при п mod 3 = 0 или п mod 3 = 1; но что делать при п mod 3 = 2?
4. Пошагово выполните алгоритм так же, как это было сделано в тексте раздела для других входных данных (см. рис. 5.136).
5. Сколько итераций выполняет данный алгоритм?
6. Вы можете реализовать алгоритм как рекурсивно, так и нерекурсивно.
7. Скорейший путь получения решения для п = 40 — воспользоваться форму­лой, использующей бинарное представление числа п (см. конец раздела 5.5).
8. Используйте бинарное представление п.
9. а) Воспользуйтесь прямой подстановкой (см. приложение Б) в рекуррентное

соотношение, приведенное в тексте раздела.

б) Выявив закономерность среди 15 значений, полученных в части а) упраж­нения, запишите ее аналитически и затем докажите по методу математи­ческой индукции.

в) Начните с бинарного представления п, и преобразуйте в бинарное пред­ставление формулу для J (п), полученную в части б) упражнения.

Упражнения 5.6

1. а) Ответ следует непосредственно из формулы, лежащей в основе алгоритма

Евклида.

б) Пусть г — т mod п. Исследуйте два возможных случая величины г по отношению к величине п.

1. У вас не должно возникнуть никаких трудностей при решении этого упраж­нения.
2. Поскольку указанный алгоритм основан на той же идее разбиения, что и быстрый поиск, естественно ожидать, что эффективность в наихудшем случае у обоих алгоритмов окажется одинаковой.
3. Запишите уравнение прямой, проходящей через точки (1,А[1\) и (г, Л [г]), и найдите координату х точки на прямой, координата у которой равна v.
4. Постройте массив, для которого интерполяция уменьшает остающийся мас­сив на один элемент при каждой итерации.
5. а) Решите неравенство log2 log2 п + 1 > 6.

б) Вычислите lim lo^gn. Обратите внимание, что с точностью до ПОСТО-

71—► ОС Юь **п**

янного множителя можно считать логарифм натуральным.

1. а) Определение бинарного дерева поиска непосредственно приводит к иско­

мому алгоритму.

б) Каким будет наихудший случай у вашего алгоритма? Сколько сравнений ключей придется сделать для входных данных такого вида?

1. а) Рассмотрите по отдельности три случая: 1) узел представляет собой лист;

2) узел имеет один дочерний узел; 3) узел имеет два дочерних узла, б) Считайте, что вам известно расположение удаляемого ключа.

1. Поиск выигрышной стратегии (если таковая существует) для п = 1,2,..., 10 — один из путей получения общего решения данной задачи.
2. Не ждите подсказки — придумайте алгоритм сами. Он не обязательно должен быть оптимальным, но все же должен быть достаточно эффективным.

Глава 6 Упражнения 6.1

1. Алгоритм уже описан в условии упражнения. Его анализ аналогичен анализу алгоритмов в тексте раздела.
2. Задача подобна одной из рассматривавшихся в тексте раздела.
3. а) Сравните каждый элемент одного множества со всеми элементами вто­

рого.

б) В действительности предварительную сортировку можно использовать тремя разными путями: отсортировать элементы только одного масси­ва, элементы обоих массивов в отдельности и элементы двух массивов вместе.

1. а) Как найти наименьшее и наибольшее значения в отсортированном списке? б) И алгоритм грубой силы, и алгоритм декомпозиции линейны.
2. Воспользуйтесь уже известными результатами для эффективности указан­ных в условии алгоритмов в среднем случае.
3. Считайте, что сортировка требует около п log2 п сравнений. Воспользуй­тесь известными результатами о количестве сравнений при успешном поиске в среднем случае для алгоритма бинарного поиска и для последовательного поиска.
4. а) Поставленная задача подобна задаче из одного из упражнений к данному

разделу.

б) Как бы вы решали эту задачу, если бы информация о студентах была записана на бумажных карточках? Еще лучше: представьте, что получить нужные сведения поручили человеку со здравым смыслом, но никогда не изучавшему алгоритмы.

1. а) Многие задачи такого вида обладают исключением для одной конфигу­

рации точек. Что же касается единственности решения, то попробуйте рассмотреть несколько “случайных” экземпляров задачи, б) Постройте многоугольники для нескольких небольших “случайных” эк­земпляров задачи, а затем попытайтесь вывести общее правило построе­ния вами таких многоугольников.

1. Возможно, вам поможет представление чисел в виде упорядоченных точек на числовой оси. Еще один вариант, который может оказаться полезным: рассмотрите случай 5 = 0, причем элементы массива могут быть как поло­жительными, так и отрицательными.
2. Дважды воспользуйтесь идеей предварительной сортировки.

Упражнения 6.2

1. Пошагово выполните алгоритм так, как это было сделано в тексте раздела для другой системы линейных уравнений.
2. а) Воспользуйтесь результатами метода исключения Гаусса так, как поясня­

лось в тексте раздела,

б) Это одна из разновидностей метода “преобразуй и властвуй”. Какая именно?

1. Для поиска обратной матрицы вы можете либо решить систему линейных уравнений одновременно с тремя столбцами свободных членов, представля­ющих столбцы единичной матрицы размера 3x3, либо воспользоваться LU- разложением матрицы коэффициентов исходной системы линейных уравне­ний, найденным при решении упражнения 2.
2. Хотя конечный ответ корректен, вывод содержит ошибку, которую вам и надо найти.
3. Псевдокод этого алгоритма очень прост. Если у вас все же возникли трудно­сти, вернитесь к пошаговому выполнению этого алгоритма в тексте раздела. Класс эффективности алгоритма можно оценить, следуя стандартному плану анализа нерекурсивных алгоритмов.
4. Оцените отношение времени работы алгоритмов с использованием прибли­женных формул для количества делений и умножений в обоих алгоритмах.
5. а) Это “нормальный” случай — одно из двух уравнений не пропорционально

второму.

б) Коэффициенты в одном уравнении должны быть такими же или про­порциональными коэффициентам второго уравнения, но для свободных членов это не должно выполняться.

в) Два уравнения должны быть одинаковы или пропорциональны друг другу (включая свободные члены).

1. а) Работайте со строками матрицы над опорной строкой точно так же, как

и со строками под ней.

б) Основаны ли методы Гаусса-Джордана и Гаусса на разных способах раз­работки алгоритмов или на одном и том же?

в) Выведите формулу для количества умножений в методе Гаусса-Джордана так же, как это было сделано для метода исключения Гаусса в тексте раздела.

1. Сколько времени потребуется для вычисления определителя по сравнению со временем, необходимым для решения системы линейных уравнений методом исключения Гаусса?
2. а) Просто примените правило Крамера к данной системе линейных уравне­ний.

б) Сколько различных определителей в формулах правила Крамера?

Упражнения 6.3

1. Воспользуйтесь определением AVL-деревьев. Не забывайте, что AVL-дере- вья — частный случай бинарных деревьев поиска.
2. Будет проще строить деревья в порядке возрастания п.
3. Одиночный L-поворот и двойной ДД-поворот представляют собой зеркаль­ное отражение одиночного Д-поворота и двойного ДД-поворота, диаграммы которых имеются в тексте раздела.
4. Вставьте ключи в указанном порядке, выполняя необходимые повороты так, как это делалось в примере, приведенном в тексте раздела.
5. а) Эффективный алгоритм вытекает непосредственно из определения бинар­

ного дерева поиска, частным случаем которого является AVL-дерево. б) Правильный ответ противоположен тому, который первым приходит в го­лову.

1. а) Пошагово выполните алгоритм для указанных входных данных (см. при­

мер на рис. 6.8).

б) Не забывайте, что количество сравнений ключей, выполняемое при поиске в 2-3-дереве, зависит не только от глубины узла, но и от того, является ли ключ первым или вторым в узле.

1. Ложно; найдите простой контрпример.
2. Где располагаются наименьший и наибольший ключи?

Упражнения 6.4

1. а) Пошагово выполните описанный в тексте раздела алгоритм для приведен­

ных входных данных.

б) Пошагово выполните описанный в тексте раздела алгоритм для приведен­ных входных данных.

в) Математический факт не может быть установлен путем проверки кор­ректности единственного примера.

1. Для пирамиды, представленной в виде массива, требуется только проверка доминирования родительских узлов.
2. а) Какую структуру имеет заполненное дерево высотой h с максимальным

количеством узлов? А с минимальным количеством узлов? б) Воспользуйтесь результатами выполнения части а) данного упражнения.

1. Сначала выразите правую часть формулы как функцию от h. Затем докажите полученное равенство либо с использованием формулы для суммы г2г из приложения А, либо с использованием математической индукции по h.
2. а) Где следует искать наименьший элемент пирамиды?

б) Удалить произвольный элемент пирамиды можно при помощи обобщения алгоритма для удаления корня.

1. Пошагово выполните алгоритм для приведенных входных данных (см. при­мер на рис. 6.14).
2. Как правило, алгоритм сортировки, который может обменивать далеко от­стоящие друг от друга элементы, устойчивым не является.
3. Ответы могут оказаться различными для разных представлений пирамиды.
4. Соберите спагетти в пучок и поставьте их на столе вертикально.

Упражнения 6.5

1. Запишите соответствующую сумму и упростите ее с использованием стан­дартных формул и правил для работы с суммами. Не забудьте включить в нее умножения вне внутреннего цикла.
2. Воспользуйтесь тем фактом, что значение хг легко вычислить на основе ранее вычисленного значения хг~1.
3. а) Используйте формулы для количества умножений (и сложений) для обоих

алгоритмов.

б) Требуется ли дополнительная память для работы схемы Горнера?

1. Примените схему Горнера для данного экземпляра так же, как это было сделано для другого экземпляра в тексте раздела.
2. Если вы эффективно реализуете алгоритм для длинного деления на х — с, ответ может вас удивить.
3. а) Пошагово выполните алгоритм бинарного возведения в степень слева на­

право для данного экземпляра задачи так же, как это было сделано для другого экземпляра в тексте раздела, б) Ответ — да. Алгоритм может быть расширен для работы с нулевой степе­нью. Как?

1. Пошагово выполните алгоритм бинарного возведения в степень справа нале­во для данного экземпляра задачи так же, как это было сделано для другого экземпляра в тексте раздела.
2. Вычисляйте и используйте биты п “на лету”.
3. Используйте формулу для суммы членов этого полинома специального вида.
4. Сравните количество операций, необходимое для реализации задач, перечис­ленных в условии.

Упражнения б.б

1. а) Воспользуйтесь правилом для вычисления lcm (га, п) и gcd (га, n) из раз­

ложений на простые множители тип. б) Ответ следует непосредственно из формулы для вычисления lcm (га, п).

1. Воспользуйтесь соотношением между задачами минимизации и максимиза­ции.
2. Докажите утверждение путем индукции по к.
3. а) Постройте ваш алгоритм на следующем наблюдении: граф содержит цикл

длиной 3 тогда и только тогда, когда он имеет две соседние вершины г и j, соединенные также путем длиной 2.

б) Не торопитесь с выводом, отвечая на данный вопрос.

1. Простейшее решение — привести задачу к другой, для которой известен алгоритм решения. Поскольку мы рассматривали не так уж много геометри­ческих алгоритмов, выбрать подходящий должно быть несложно.
2. Выразите данную задачу как задачу максимизации функции одной перемен­ной.
3. Введите двухиндексную переменную указывающую назначение j-го за­дания г-му работнику.
4. Воспользуйтесь особенностью данной задачи, чтобы уменьшить количество неизвестных на одно.
5. Создайте новый граф.
6. а, б) Постройте граф пространства состояний для данной задачи, как это было сделано для задачи о переправе волка, козы и капусты,

в) Обратите внимание на состояние после шести пересечений реки в реше­нии части б).

Глава 7 Упражнения 7.1

1. Да, это возможно. Как?
2. Обратитесь к псевдокоду алгоритма и посмотрите, что он делает в случае одинаковых значений.
3. Пошагово пройдите алгоритм для данных значений (см. рис. 7.2 в качестве примера).
4. Проверьте, может ли алгоритм обратить относительный порядок равных эле­ментов.
5. Где именно в отсортированном массиве находится элемент А [г]?
6. Попробуйте для начала решить задачу об украинском флаге, т.е. эффективно преобразовать массив символов Ж и С (с подобной задачей мы уже встреча­лись в упражнении 4.2.8).
7. Воспользуйтесь стандартными обходами таких деревьев.
8. а) Следуйте определению массивов В и С в описании метода, б) Найдите, скажем, В [С [3]] для примера из части а).
9. а) Воспользуйтесь для хранения ненулевых элементов матрицы связанными

списками.

б) Представьте каждый из данных полиномов связанным списком с узлами, содержащими степень г и коэффициент а\* для каждого ненулевого члена

***ciiX1*.**

1. Вы можете воспользоваться симметрией доски для уменьшения позиций, которые следует хранить.

Упражнения 7.2

1. Пошагово выполните алгоритм так же, как это было сделано в тексте раздела для другого экземпляра задачи поиска подстроки.
2. Несмотря на специальный алфавит, это приложение алгоритма ничем не отличается от приложения к строкам на естественном языке.
3. Для каждого образца заполните таблицу сдвигов и определите количество сравнений символов (как удачных, так и неуспешных) при каждой попытке, а также количество выполненных попыток.
4. Найдите пример бинарной строки длиной т и бинарной строки длиной п (п ^ га) таких, что алгоритм Хорспула выполняет

а) наибольшее возможное количество сравнений символов перед выполне­нием минимального возможного сдвига;

б) наименьшее возможное количество сравнений символов.

1. Логично рассмотреть наихудший случай для алгоритма Хорспула.
2. Может ли алгоритм сдвинуть образец более чем на одну позицию без воз­можного пропуска другого вхождения искомой подстроки?
3. Для каждого образца заполните две таблицы сдвигов и определите коли­чество сравнений символов (как удачных, так и неуспешных) при каждой попытке, а также количество выполненных попыток.
4. Обратитесь к описанию алгоритма Бойера-Мура.
5. Обратитесь к описаниям алгоритмов.

Упражнения 7.3

1. Примените схему открытого хеширования (с раздельными цепочками) для указанных входных данных, как это было сделано в главе для других входных данных (см. рис. 7.5). Затем вычислите наибольшее и среднее количества сравнений для успешного поиска в построенной таблице.
2. Примените схему закрытого хеширования (с открытой адресацией) для ука­занных входных данных, как это было сделано в главе для других входных данных (см. рис. 7.6). Затем вычислите наибольшее и среднее количества сравнений для успешного поиска в построенной таблице.
3. Сколько различных адресов может выдать такая хеш-функция? Будет ли рас­пределение ключей равномерным?
4. Вопрос подобен вопросу о вычислении вероятности получения одного и того же результата при п бросках кости в честной игре.
5. Найдите вероятность того, что п человек имеют различные дни рождения. Что касается хеширования, то какое явление в нем связано с совпадениями?
6. а) Нет необходимости вставлять новый ключ в конец связанного списка,

в который он хеширован, б) Какая операция выполняется быстрее с отсортированным связанным спис­ком и почему? В случае сортировки должны ли мы копировать все эле­менты из непустых списков в массив и затем применять один из уни-

версальных алгоритмов сортировки, или имеется способ воспользоваться отсортированностью ключей в непустых связанных списках?

1. После, того как вы ответите на поставленные в этом упражнении вопросы, сравните эффективность вашего алгоритма с алгоритмом на основе грубой силы (раздел 2.3) и алгоритма с использованием предварительной сортиров­ки (раздел 6.1).
2. Рассматривайте этот вопрос как небольшое повторение пройденного мате­риала: ответы для последних двух столбцов содержатся в разделе 7.3, для остальных столбцов вы найдете ответы в соответствующих разделах книги (само собой, для ответов вы должны использовать наилучшие из имеющихся алгоритмов).
3. Если вам надо обновить память, просто загляните в оглавление книги.

Упражнения 7.4

1. Размышления о поиске информации должны натолкнуть вас на множество примеров.
2. а) Воспользуйтесь стандартными правилами работы с суммами, в частности

формулой суМмы геометрической прогрессии, б) Вам надо применить логарифмирование по основанию [т/2].

1. Найдите искомое значение из неравенства для верхней границы высоты В- дерева, приведенного в тексте раздела.
2. Следуйте алгоритму вставки, описанному в разделе.
3. Алгоритм следует из определения В-дерева.
4. а) Просто следуйте описанию алгоритма, данному в условии задачи. Обра­

тите внимание, что ключ всегда вставляется в лист и что полные узлы всегда разбиваются по пути вниз, несмотря на то, что лист для нового ключа может иметь достаточно места для его размещения,

б) Может ли разделение полного узла вызвать каскадное разделение по це­почке предков? Можем ли мы получить более высокое дерево, чем это необходимо?

Глава 8 Упражнения 8.1

1. Сравните определения двух методов.
2. а) Пошагово выполните алгоритм Binomial для п=6кк=3и заполните

таблицу, аналогичную таблице на рис. 8.1. б) Обратитесь к формуле алгоритма еще раз и посмотрите, какие значения требуются для вычисления С%.

1. Покажите, что существуют положительные константы с\ и С2 и натуральное число по такие, что для всех пар целых чисел п и /с (п ^ по и 0 ^ /с ^ п)

с^пк ^ --- --------- + к (п — к) ^ С\пк.

1. а) Вычислить пространственную эффективность алгоритма можно тем же

способом, как и временную эффективность в разделе 8.1. б) С определенными предосторожностями новую строку таблицы можно за­писать поверх предыдущей.

1. Воспользуйтесь явной формулой для С% и при необходимости формулой Стирлинга.
2. Запишите рекуррентное соотношение для количества сложений А (п, к) и ре­шите его подстановкой А (п. к) = В (п, к) — 1.
3. Найдите и сравните классы эффективности указанных алгоритмов.
4. Формулу можно доказать либо с использованием явной формулы для С%, либо с применением комбинаторной интерпретации этой величины.
5. а) В ситуации, когда командам А и В требуется для победы в серии г и j

игр, соответственно, рассмотрите результат победы в игре команды А и результат ее поражения.

б) Заполните таблицу из 5 строк (0 ^ г ^ 4) и 5 столбцов (0 ^ j ^ 4) с использованием рекуррентного соотношения из части а) упражнения.

в) Псевдокод должен руководствоваться рекуррентным соотношением из ча­сти а) упражнения. Ответ об эффективности должен следовать непосред­ственно из размера таблицы и времени, затраченного на вычисление каж­дого из ее элементов.

Упражнения 8.2

1. Примените алгоритм к матрице смежности так, как это делалось в тексте раздела для другой матрицы.
2. а) Ответ может быть получен либо путем рассмотрения количества вычис­

ляемых алгоритмом значений, либо при помощи стандартного плана для анализа нерекурсивных алгоритмов (т.е. путем записи суммы для количе­ства выполнений базовых операций).

б) К какому классу эффективности принадлежит алгоритм на основе обхода для разреженных графов, представленных при помощи связанных списков смежности?

1. Покажите, что можно просто перезаписывать элементы Tt^-i) элементами RW без каких-либо изменений в алгоритме.
2. Что будет, если R(k~^ [г, к] = О?
3. Сначала покажите, что формула (8.7) (из которой можно удалить верхние индексы в соответствии с упражнением 3)

***nj*** = ***rij or*** *rik*and *rkj*

эквивалентна формуле

if rik rij <- or rkj.

1. а) Какое свойство транзитивного замыкания указывает наличие ориентиро­

ванного цикла? Имеется ли лучший алгоритм для проверки этого факта?

б) Какие элементы транзитивного замыкания неориентированного графа рав­ны 1? Можно ли найти такие элементы при помощи более быстрого ал­горитма?

1. См. пример применения алгоритма к другому экземпляру задачи в тексте раздела.
2. От каких элементов матрицы D^k~^ зависит элемент на пересечении г-ой строки и j-ro столбца матрицы Могут ли эти значения быть из­менены при перезаписывании?
3. Ваш контрпример должен содержать цикл отрицательной длины.
4. Достаточно хранить в отдельной матрице Р индексы промежуточных вер­шин, использованных при обновлении матрицы расстояний. Такую матрицу можно инициализировать, скажем, значениями —1.

Упражнения 8.3

1. Продолжите применение формулы (8.11), как предписывает алгоритм.
2. а) Временную эффективность алгоритма можно найти в соответствии со

стандартным планом анализа временной эффективности нерекурсивного алгоритма.

б) Сколько памяти требуется для двух таблиц, генерируемых алгоритмом?

1. к = R [1, п\ указывает, что корень оптимального дерева представляет собой к-ый ключ в списке упорядоченных ключей ai, <22,..., ап. Корни его левого и правого поддеревьев определяются как R [1, к — 1] и R [к + 1, п], соответ­ственно.
2. Воспользуйтесь методом пространственно-временного компромисса.
3. Если предложенное утверждение истинно, то разве это не означает нали­чия более простого алгоритма построения оптимального бинарного дерева поиска?
4. Структура дерева должна просто минимизировать среднюю глубину его уз­лов. Не забудьте указать способ распределения ключей по узлам дерева.
5. а) Поскольку имеется однозначное соответствие между бинарными деревья­

ми поиска для данного множества из п упорядочиваемых ключей и бинар­ными деревьями с п узлами (почему?), вы можете пересчитать последние. Рассмотрите все возможные разделения узлов между левым и правым поддеревьями.

б) Вычислите интересующие вас значения с использованием двух формул.

в) Воспользуйтесь формулой для n-го числа Каталана и формулой Стирлин­га для п\.

1. Измените границы внутреннего цикла из алгоритма OptimalBST, восполь­зовавшись монотонностью таблицы корней, о которой упоминается в конце раздела 8.3.
2. Предположим, что ai,..., an — различные ключи, упорядоченные от наи­меньшего к наибольшему; pi,... ,рп — вероятности их поиска, а go, gi,..., qn — вероятности неудачных поисков ключей в интервалах (—00, ai), (ai, a2),

..., (an, 00), соответственно; (pi H + pn) + (go H H gn) = 1. Запишите

рекуррентное соотношение, аналогичное соотношению (8.11), для ожидае­мого количества сравнений ключей с учетом как успешных, так и неудачных поисков.

1. а) Проще всего найти общую формулу для количества умножений, необ­ходимых для вычислений (А\ • А2) • As и А\ • (A<i • As) для матриц А\ размером do х А2 размером d\ х d<i и As размером d<i х ds, а затем выбрать конкретные размеры так, чтобы одно из значений как минимум в 1000 раз превышало другое.

б) Ответ можно получить, следуя подходу, использовавшемуся для подсчета количества бинарных деревьев.

в) Рекуррентное отношение для оптимального количества умножений при вычислении Ai • ... • Aj очень похоже на рекуррентное соотношение для оптимального количества сравнений в бинарном дереве поиска, состав­ленном из ключей а\*,..., aj.

Упражнения 8.4

1. а) Воспользуйтесь формулами (8.12) и (8.13) для заполнения соответствую­

щей таблицы, как это было сделано для другого экземпляра задачи в тексте раздела.

б) Что означает равенство членов в формуле

max{V[\* - 1 ,j),Vi + V[i - 1, j - го\*]}?

1. а) Напишите псевдокод для заполнения таблицы на рис. 8.12 (скажем, строка

за строкой) с использованием формул (8.12) и (8.13). б) Алгоритм для определения состава оптимального подмножества описан в разделе на конкретном примере.

1. Сколько значений должен вычислить алгоритм? Сколько времени занимает вычисление одного значения? Сколько ячеек таблицы следует обойти для того, чтобы определить состав оптимального подмножества?
2. Воспользуйтесь определением V [г, j] для проверки истинности

а) V [i,j - 1] < V [г, j] для 1 ^ j ^ W;

б) V [г — 1, j] < V [г, j] для 1 < i ^ п.

1. Пройдите пошагово вызов функции MFKnapsack (г, j) для указанного экзем­пляра задачи (применение этой функции к другому экземпляру задачи можно найти в тексте раздела).
2. Алгоритм использует формулу (8.12) для заполнения некоторых ячеек таб­лицы. Почему мы можем утверждать, что его эффективность остается при­надлежащей классу Q{nW)l
3. Модифицируйте восходящий алгоритм из раздела 8.3 так же, как был моди­фицирован восходящий алгоритм в разделе 8.4.
4. Одна из причин связана со временной эффективностью, а вторая — с эффек­тивностью пространственной.
5. Запишите рекуррентное соотношение для C[z,j], наименьшего количества монет в экземпляре задачи при использовании монет первых г достоинств (1 ^ г ^ т) и сумме j (0 ^ j ^ п). Для экземпляра задачи без решения используйте в качестве значения С [г, j] величину +оо.

Глава 9

Упражнения 9.1

1. В качестве контрпримера можете использовать набор номиналов, упомяну­тый в тексте: d\ — 7, d<i — 5, dz — 1.
2. В своем алгоритме вы можете использовать деление нацело.
3. Вам может помочь рассмотрение случая двух заданий. Конечно, сформули­ровав гипотезу, вы должны либо доказать оптимальность выхода для произ­вольных входных данных, либо найти контрпример, показывающий, что это не так.
4. Вы можете применить жадный подход либо ко всей матрице стоимости, либо к каждой из ее строк (или столбцов).
5. В обоих версиях задачи несложно выдвинуть гипотезу о виде решения после рассмотрения случаев п = 1, 2 и 3. Главным в задаче является доказательство корректности ее решения.
6. а) Пошагово выполните алгоритм для данного в условии графа. Пример

выполнения можно найти в тексте раздела, б) После того как очередная пограничная вершина добавляется в дерево, в очередь с приоритетами следует добавить все незамеченные вершины, смежные с данной.

1. Применение алгоритма Прима ко взвешенному несвязному графу должно помочь вам ответить на поставленный вопрос.
2. а) Поскольку алгоритм Прима требует наличия весов у ребер графа, им

следует назначить некоторые веса, б) Знаете ли вы другой алгоритм, способный справиться с этой задачей?

1. Строго говоря, формулировка задачи требует доказать две вещи: что суще­ствует по крайней мере одно минимальное остовное дерево для любого взве­шенного связного графа и что минимальное остовное дерево единственно, если все веса различны. Доказательство первого утверждения вытекает из очевидного наблюдения о конечности количества остовных деревьев у взве­шенного связанного графа. Доказательство второго утверждения может быть получено путем повторения доказательства корректности алгоритма Прима с небольшим изменением в конце.
2. Рассмотрите два случая: уменьшение значения ключа (этот случай реализу­ется в алгоритме Прима) и увеличение значения ключа.

Упражнения 9.2

1. Пошагово выполните алгоритм для данных графов так же, как это было сделано для другого графа в тексте раздела.
2. Два из четырех утверждений истинны, два — ложны.
3. Примените алгоритм Крускала к несвязному графу; это должно помочь вам ответить на поставленный вопрос.
4. Ответ один и тот же для обоих алгоритмов. Если вы считаете, что алгоритмы будут корректно работать для графов с отрицательными весами, докажите это; если нет, приведите контрпримеры для каждого алгоритма.
5. Применим ли здесь метод преобразования минимизации в задачу максими­зации из раздела 6.6?
6. Подставьте три операции над абстрактным типом данных непересекающихся подмножеств — makeset (ж), find(x) и union (ж, у) — в соответствующие места псевдокода, приведенного в тексте раздела.
7. Следуйте плану, использованному в разделе 9.1 для доказательства коррект­ности алгоритма Прима.
8. Доказательство очень похоже на доказательство, выполненное в тексте для версии union-by-size быстрого поиска.
9. Вы можете воспользоваться списком желательных характеристик визуализа­ций алгоритмов, приведенном в разделе 2.7.

Упражнения 9.3

1. Один из вопросов не требует внесения изменений в алгоритм или граф; остальные требуют простых модификаций.
2. Пошагово выполните алгоритм для данных графов так же, как это было сделано для другого графа в тексте раздела.
3. В таком графе ближайшая вершина не обязательно должна быть смежной с исходной.
4. Верно только одно из утверждений. Найдите небольшой контрпример для второго.
5. Упростите приведенный в тексте раздела псевдокод, реализуя очередь с при­оритетами в виде неупорядоченного массива и игнорируя метки вершин, указывающие родительские узлы.
6. Докажите корректность по индукции по количеству вершин, включенных в дерево, строящееся алгоритмом.
7. Сначала топологически отсортируйте вершины ориентированного ацикличе­ского графа.
8. Сначала топологически отсортируйте вершины ориентированного ацикличе­ского графа.
9. Воспользуйтесь преимуществами геометрического и физического мышле­ния.
10. Перед тем как приступать к реализации алгоритма поиска кратчайших пу­тей, решите, какой критерий определяет “наилучший маршрут”. Желательно, чтобы ваша программа спрашивала пользователя, какой из нескольких кри­териев оптимальности следует применить.

Упражнения 9.4

1. См. пример, приведенный в тексте раздела.
2. После объединения двух узлов с наименьшими вероятностями разрешить возникающую неоднозначность можно двумя путями. Для каждого из двух полученных кодов Хаффмана вычислите среднее и дисперсию длин кодов.
3. Вы можете основываться при ответе на методе работы алгоритма Хаффмана или на том факте, что коды Хаффмана являются оптимальными префиксны­ми кодами.
4. Максимальная длина кода очевидным образом связана с высотой дерева Хаффмана. Попытайтесь найти множество п конкретных частот для алфави­та из п символов, для которых дерево имеет вид, приводящий к максимально возможной длине кода.
5. а) Какая структура данных наиболее подходит для реализации алгоритма,

основной операцией которого является поиск двух минимальных элемен­тов данного множества и их замена их суммой?

б) Определите базовую операцию алгоритма, количество ее выполнений и эффективность для выбранной структуры данных.

1. Поддерживайте две очереди: одну для заданных частот, вторую — для весов новых деревьев.
2. Естественно было бы использовать один из алгоритмов обхода дерева.
3. Генерируйте коды справа налево.
4. Аналогичный пример рассматривался в конце раздела 9.4. Постройте дерево Хаффмана, а затем подумайте над вопросами, которые должно давать такое дерево (учтите, что можно, например, спросить: “Это единица или семерка, или восьмерка?”).

Глава 10 Упражнения 10.1

1. Поскольку вы знаете, что количество перемещений дисков, выполняемых классическим алгоритмом, равно 2П — 1, то можете доказать (например, при помощи математической индукции), что для любого алгоритма, решающего эту задачу, количество перемещений дисков М (п), выполняемое этим алго­ритмом, не менее 2п — 1. В качестве альтернативного метода можете показать, что если М\* (п) — минимальное количество перемещений дисков, то М\* (тг) удовлетворяет рекуррентному соотношению

М\* (тг) = 2М\* (тг) + 1 при тг > 1 и М\* (1) = 1,

решение которого — 2п — 1.

1. Все эти вопросы имеют достаточно простые ответы. Если тривиальная ниж­няя граница плотная, не забудьте указать конкретный алгоритм, который доказывает ее плотность.
2. Вернитесь к разделу 5.5, где вы впервые встретились с задачей о фальшивой монете; это должно помочь в решении поставленной задачи.
3. Подумайте о числах, проигрывающих при сравнении.
4. Разделите множество вершин графа на два непересекающихся подмножества U и W, имеющих по [п/2J и \п/2] вершин, соответственно, и покажите, что любой алгоритм будет проверять наличие ребер между всеми парами вершин (it,w), где ueU и ги Е W перед тем как установит связность графа.
5. Вопрос и ответ очень похожи на случай двух n-элементных отсортирован­ных списков, рассмотренный в тексте раздела. Как и доказательство нижней границы.
6. Просто следуйте формулам преобразования, предложенным в тексте раздела.
7. а) Проверьте, выполняются ли формулы для двух произвольных квадратных

матриц.

б) Воспользуйтесь формулой, аналогичной формуле в тексте раздела, и по­кажите, что умножение произвольных квадратных матриц может быть приведено к возведению в квадрат большей матрицы.

1. Какая задача с известной нижней границей наиболее подобна поставлен­ной в упражнении? После того, как вы найдете подходящее приведение, не забудьте указать алгоритм, который делает нижнюю границу плотной.
2. Вам может помочь рассмотрение частного случая п = 1.

Упражнения 10.2

1. а) Сначала докажите, что 2h ^ I при помощи индукции по h. б) Сначала докажите, что Zh ^ I при помощи индукции по h.
2. а) Сколько всего выходов имеет данная задача?

б) Естественно, имеется много способов решения этой простой задачи.

в) Вам должно помочь представление а, b и с как точек на числовой оси.

1. Это очень простой вопрос. Вы можете считать, что все три сортируемых элемента различны. (Если вам нужна помощь, обратитесь к деревьям приня­тия решения для трехэлементной сортировки выбором и вставкой в тексте раздела.)
2. Вычислите нетривиальную нижнюю границу для сортировки четырехэле­ментного массива, а затем определите, у какого алгоритма количество срав­нений в наихудшем случае соответствует этой нижней границе.
3. Это не такая уж простая задача. Ни один из стандартных алгоритмов сорти­ровки не способен на это. Попытайтесь разработать специальный алгоритм, который получает из каждого сравнения как можно больше информации.
4. Это очень простое упражнение. Воспользуйтесь тем очевидным фактом, что последовательный поиск в отсортированном массиве может прекращаться по достижении элемента, большего, чем искомый ключ.
5. а) Начните с приведения логарифмов к одному основанию, б) Простейший способ состоит в доказательстве того, что

flOg; (П + I)] nioo flogg (2П + 1)1

Чтобы избавиться от функции “потолок”, можно воспользоваться тем, что

/(”)-! , Г/ 0)1 / (n) + 1

**д** **(n) + 1** \д **(п)]** д (п) — **1 ’**

где / (п) = log2 (п + 1) и д (п) = log3 (2п + 1), и показать, что

/ (п) — 1 / (тг) + 1

lim —у— = lim —— г > 1.

п-+ оо д (тг) + 1 п ►оо д (тг) — 1

1. а) Сколько выходов имеет данная задача?

б) Изобразите тернарное дерево принятия решения для данной задачи.

1. а) Покажите, что каждый из двух случаев — взвешивания двух монет (по

одной на каждой чашке) или четырех монет (по две на каждой чашке) — приводит как минимум к одной ситуации с более чем тремя возможными выходами. Получающаяся ситуация не может быть однозначно разрешена при помощи единственного взвешивания.[[78]](#footnote-78)

б) Сначала решите, следует ли начинать со взвешивания двух монет. Не забудьте, что у вас есть подлинная дополнительная монета.

в) Это знаменитая головоломка. Главное для ее решения — разобраться с ча­стью б) упражнения.

1. а) Подумайте о проигравших.

б) Рассмотрите высоту турнирного дерева (или количество шагов, необходи­мых для приведения п-элементного множества к одноэлементному путем деления пополам).

в) После того как определен победитель, какой игрок оказывается на втором месте?

Упражнения 10.3

1. Обратитесь к определению разрешимой задачи принятия решения.
2. Сначала выясните, является ли nlog271 полиномиальной функцией. Затем вни­мательно прочтите определения легких и трудных задач.
3. Все четыре комбинации возможны, и для каждой имеются примеры неболь­шого размера.
4. Просто воспользуйтесь определением хроматического числа. Решение упраж­нения 5 может помочь вам (хотя и не обязательно).
5. Эта задача должна быть вам уже знакома.
6. Что именно является мерой размера входных данных для рассматриваемой задачи?
7. Обратитесь к формулировке версии принятия решения задачи о раскраске графа и алгоритму верификации для задачи о гамильтоновом цикле.
8. Можно начать с выражения задачи о разделении в виде линейного уравнения с переменными г = 1,..., п, которые могут принимать только значения 0 и 1.
9. Если вы не знакомы с понятиями клики, вершинного покрытия и незави­симого множества, неплохо начать с поиска клики максимального разме­ра, вершинного покрытия минимального размера и независимого множества максимального размера для некоторых простых графов, наподобие рассмат­риваемых в упражнении 4. Затем попытайтесь найти связь между этими тремя понятиями. Вам может помочь рассмотрение дополнения вашего гра­фа, которое представляет собой граф с теми же вершинами и ребрами между теми вершинами, которые не являются смежными в исходном графе.
10. Современным представлениям о классах сложности не противоречат две из приведенных диаграмм.
11. Необходимая вам задача явно упоминается в тексте раздела 10.3.

Упражнения 10.4

1. Вычислите относительные ошибки приближений и воспользуйтесь опреде­лением числа значащих цифр в приближении. Один из ответов не согласуется с нашим интуитивным представлением о данном понятии.
2. Воспользуйтесь определением абсолютной и относительной ошибок и свой­ствами модуля.
3. Вычислите значение Х^=о0-5г/^ и величину разности между полученным значением и у/ё = 1.648721....
4. Примените формулу площади трапеции к каждой из трапеций и вычислите сумму полученных площадей.
5. Примените формулы (10.7) и (10.9) к данным интегралам.
6. Найдите верхнюю границу второй производной esinx и воспользуйтесь фор­мулой (10.9) для поиска значения п, гарантирующего ошибку усечения мень­ше указанного предела.
7. Аналогичная задача была разобрана в тексте раздела.
8. Рассмотрите все возможные значения параметров а, b и с. Не забывайте о том, что решение уравнения состоит в поиске его корней или доказательстве их отсутствия.
9. а) Докажите, что каждый элемент хп последовательности 1) положителен,

2) больше y/D (вычисляя xn+i — y/D) и 3) меньше предыдущего (вычисляя xn+i — хп). Затем найдите предел каждой из частей равенства (10.15) при п, стремящемся к- бесконечности.

б) Воспользуйтесь уравнением xn+i — y/D = ^Хп2д^ .

1. Это было сделано в тексте раздела для у/2.

Глава 11 Упражнения 11.1

1. а) Продолжите алгоритм путем возврата из листа, соответствующего перво­

му решению.

б) Как можно получить второе решение из первого, используя симметрию доски?

1. Подумайте о применении поиска с возвратом в обратном направлении.
2. а) Воспользуйтесь общим шаблоном алгоритма поиска с возвратом. Вы

должны найти способ определить, имеются ли при данной расстановке два ферзя, угрожающие друг другу.

Чтобы упростить сравнение с алгоритмом исчерпывающего поиска, вы можете рассмотреть версию, которая находит все решения задачи, не поль­зуясь симметрией доски. Заметьте также, что алгоритм исчерпывающего перебора может либо рассматривать все возможные размещения п фер­зей в п различных клетках на доске размером п х п, либо размещение ферзей только на различных горизонталях (или вертикалях), или только на различных вертикалях и различных горизонталях, б) Алгоритм для оценки размера дерева можно получить из алгоритма для решения задачи методом поиска с возвратом путем внесения небольших изменений. Хотя более поучительно посмотреть, насколько точна такая оценка для одного случайного пути, вы можете захотеть вычислить сред­нее значение для нескольких путей, чтобы получить более точную оценку размера дерева.

1. В тексте раздела был решен другой экземпляр этой задачи.
2. Заметим, что без потери общности можно считать, что вершина а окрашена цветом 1, и связать эту информацию с корнем дерева пространства состоя­ний.
3. Такое применение поиска с возвратом достаточно тривиально.
4. а) В тексте раздела был решен другой экземпляр этой задачи.

б) Некоторые из узлов будут рассматриваться как обещающие, в то время как они не являются таковыми.

1. Для выполнения задания требуется внести в шаблон минимальное измене­ние.
2. Не забудьте о симметрии доски при решении этой задачи.

Упражнения 11.2

1. Какие операции алгоритм ветвей и границ с выбором наилучшего варианта выполняет над живыми узлами в дереве пространства состояний?
2. Воспользуйтесь для вычисления нижних границ наименьшими числами, вы­бранными из столбцов матрицы стоимости. При применении такой функции для вычисления границ логичнее рассматривать четыре варианта назначения задания 1 в качестве узлов первого уровня дерева пространства состояний.
3. а) Ваш ответ должен быть матрицей размером пхп с простой структурой,

которая приводит к максимально быстрой работе алгоритма, б) Набросайте структуру дерева пространства состояний для вашего ответа к части а) упражнения.

1. Аналогичная задача была решена в тексте раздела.
2. Примите во внимание более одного предмета из тех, которые не включены в рассматриваемое подмножество.
3. Гамильтонов цикл для каждой вершины графа должен иметь ровно два ин­цидентных ребра.
4. Аналогичная задача была решена в тексте раздела.

Упражнения 11.3

1. а) Начните с маркировки первого столбца матрицы и поиска наименьшего

элемента в первой строке и немаркированном столбце,

б) Вы должны найти оптимальное решение путем исчерпывающего пере­бора или при помощи алгоритма ветвей и границ (или каким-то иным методом).

1. а) Простейший подход заключается в маркировке столбцов, соответствую­

щих посещенным городам. В качестве альтернативы можно использовать связанный список непосещенных городов,

б) Следование стандартному плану анализа эффективности алгоритма не должно вызвать сложностей (и привести к одному и тому же результату для обоих вариантов, упомянутых в указании к части а упражнения).

1. Двигайтесь по часовой стрелке.
2. Распространите неравенство треугольника на случай к ^ 1 промежуточных вершин и докажите его корректность методом математической индукции.
3. Сначала определите временную эффективность каждого из трех шагов алго­ритма.
4. Вы должны доказать два факта.
5. / (5\*) ^ 2/ (sa) для любого экземпляра задачи о рюкзаке, где / (sa) — зна­чение приближенного решения, полученное усовершенствованным жад­ным алгоритмом, а / (5\*) — оптимальное значение точного решения того же экземпляра задачи.
6. 2 — наименьшая константа, для которой справедливо это утверждение. Для доказательства 1) воспользуйтесь значением оптимального решения не­прерывной версии задачи и ее связью со значением приближенного решения. Для доказательства 2) найдите семейство трехпредметных экземпляров, ко­торое доказывает данное утверждение (два предмета могут иметь вес W/2, а третий может иметь вес немного больший, чем W/2).
7. а) Пошагово пройдите алгоритм для данного экземпляра, а затем ответьте

на вопрос, можно ли разместить те же предметы в меньшем количестве корзин.

б) Какова базовая операция данного алгоритма? Какие входные данные за­ставляют алгоритм дольше работать?

в) Сначала докажите неравенство

*п*

Врр < 2 ^ si для любого экземпляра с Bff > 1,

»=1

где Bff — количество корзин, полученных при применении алгоритма первого подходящего к экземпляру с размерами si, 52,..., sn. Чтобы до­казать это, воспользуйтесь тем фактом, что может быть не более одной корзины, заполненной наполовину или менее.

1. а) Пошагово пройдите алгоритм для данного экземпляра, а затем ответьте

на вопрос, можно ли разместить те же предметы в меньшем количестве корзин.

б) На вопрос можно ответить либо при помощи теоретического доказатель­ства, либо при помощи контрпримера.

в) Воспользуйтесь двумя следующими свойствами.

1. Все предметы, размещаемые алгоритмом первого подходящего с убы­ванием в дополнительные корзины, т.е. корзины после первых В\*9 имеют размер не более 1/3.
2. Общее количество предметов, размещенных в дополнительных корзи­нах, не превышает В\* — 1 (В\* — оптимальное количество корзин).

г) Эта задача имеет две версии с существенно разными уровнями сложности. Какие это версии?

1. а) Один такой алгоритм основан на идее, подобной использованной в алго­

ритме поиска транзитивного замыкания, с тем отличием, что он начинает работу с произвольного ребра графа,

б) Вспомните предупреждение о том, что эквивалентность сложности по­линомиально приводимых NP-сложных задач не распространяется на приближенные алгоритмы.

1. а) Раскрашивайте вершины, не используя новые цвета без крайней необхо­димости.

б) Найдите последовательность графов Gn, для которых отношение

*Ка (&п)*

***к\* (Gn)***

(где ка (Gn) и к\* (Gn) — соответственно, количество цветов, получен­ное жадным алгоритмом, и оптимальное количество цветов) может быть сделано произвольно большим.

Упражнения 11.4

1. В ваших поисках может помочь информация о том, что это решение было впервые опубликовано итальянским математиком эпохи Возрождения Джи- роламо Кардано (Girolamo Cardano).
2. На эти вопросы можно ответить без применения дифференциального ис­числения или сложных вычислений, просто представив уравнения в виде fi (ж) = /2 (я) и нарисовав графики f\ (ж) и /2 (ж).
3. а) Воспользуйтесь свойством, лежащим в основе метода деления пополам.

б) Воспользуйтесь определением деления полинома р (ж) на х — то, т.е. урав­нением р (ж) = q (ж) (ж — жо) + г, где жо — корень полинома р (ж), a q (ж) и г — соответственно, частное и остаток.

в) Продифференцируйте обе стороны уравнения из части б и подставьте жо в полученный результат.

1. Используйте тот факт, что \хп — ж\*| — расстояние между жп, срединой от­резка [ап, Ьп], и корнем ж\*.
2. Набросайте график для определения общего положения корня и выберите подходящий отрезок, в котором находится корень. Воспользуйтесь неравен­ством из раздела 11.4 для определения наименьшего количества необходи­мых итераций. ВьГполните итерации алгоритма, как это было сделано для примера в тексте раздела.
3. Напишите уравнение прямой, проходящей через точки (ап,/(ап)) и (Ьп, / (Ьп)), и найдите точку ее пересечения с осью абсцисс.
4. Обратитесь к примеру в тексте раздела. В качестве критерия останова можно использовать длину отрезка (ап, Ьп) либо неравенство (11.11).
5. Напишите уравнение касательной к графику функции в точке (жп,/(жп)) и найдите точку ее пересечения с осью абсцисс.
6. Обратитесь к примеру в тексте раздела. Конечно, вы можете выбрать другое начальное значение жо.
7. Рассмотрите, например, / (ж) = ^/ж.

Предметный указатель

**Символы**

2-3-4-дерево, 266 2-3-дерево, 266; 212

**А**

Abel, Niels, 476 Absolute error, 431 Abstract data type, 69 Acyclic graph, 63 Adjacency matrix, 59 Adleman, Len, 490 ADT, 69

Adversary method, 405 Agrawal, Manindra, 424 Algorithm design technique, 36 Algorithmics, 23

All-pairs shortest-paths problem, 349 Amortized efficiency, 84 Analysis, 73 Ancestors, 64 Ancestry problem, 311 Approximation algorithm, 35 Approximation schemes, 471 Array, 55

Articulation point, 215 Average-case efficiency, 82 AVL tree, 266; 267

В

B-tree, 266; 331 Back edge, 212; 221 Backward substitution, 109 Baecker, Ronald, 135 Basic operation, 77 Bayer, Rudolf, 331 Bellman, Richard, 339; 352 Berlinski, David, 23

Binary insertion sort, 207 Binary search tree, 65 Binary string, 55 Binary tree, 65 Binomial coefficient, 341 Bipartite graph, 219 Bisection method, 476 Bit string, 55 Bit vector, 68

Boyer, Robert, 48; 150; 316 Breadth-first search, 212; 215 Breadth-first search forest, 215 Brewer, E. Cobham, 203 Brown, C., 501 Burks, A., 34 Byrne, Robert, 141

***С***

С-approximation algorithm, 463

Cardano, Girolamo, 561

Carroll, Lewis, 525

Catalan number, 355

Change-making problem, 367; 369

Characteristic equation, 499

Child, 64

Class, 69

Clique, 427

Clique problem, 165

Closed hashing, 325

Cluster, 328

Collision, 324

Comparison counting, 307

Complete graph, 59

Complete tree, 276

Compression ratio, 394

Computational complexity, 417

Connected component, 62

Connected graph, 62 Cook, Stephen, 424 Cost matrix, 61 Cramer’s rule, 263 Cross edge, 215; 221 Cycle, 63

D

Dantzig, George, 296 Data structure, 54 Decision problem, 418 Decision tree, 396; 409 Dense graph, 59 Depth, 65

Depth-first search, 212 Depth-first search forest, 212 Dequeue, 57 Descartes, Rene, 292 Descendants, 65 Determinant, 262 Dickens, Charles, 339 Dictionary, 69 Digraph, 58; 220 Dijkstra algorithm, 387 Dijkstra, Edsger W., 387 Directed acyclic graph, 221 Directed graph, 58; 220 Distance matrix, 349 Distribution counting, 308 Double hashing, 328 Doubly linked list, 56 Douglas, Michael, 369 Dynamic Huffman encoding, 394 Dynamic programming, 306

**E**

Edison, Thomas, 441 Einstein, Albert, 73 Enqueue, 57

Essentially complete tree, 276 Euler, Leonard, 52; 255 Exact algorithm, 35 Exhaustive search, 159 Exponent, 431 Extensible hashing, 329 External path length, 189 Extrapolation, 133

Extreme point, 156

**F**

Fast Fourier transform, 284 Feasible region, 159 Feasible solution, 451 FFT, 284

Fibonacci heap, 390 Fibonacci, Leonardo, 119 FIFO, 57

First-in-first-out, 57 Fixed-length encoding, 392 Flavius, Josephus, 235 Flowchart, 38 Floyd’s algorithm, 350 Floyd, R., 350 Forest, 63

Forsythe, George, 14; 435 Forward edge, 221 Free tree, 63 Front, 57

G

Galois, Evariste, 476 Gauss elimination, 255 Gauss, Karl Friedrich, 107; 255; 476 Gauss-Jordan elimination, 265 General linear kth degree recurrence with constant coefficients, 501 Generality, 39 Generic term, 495 Gloor, Peter, 136

Goethe, Johann Wolfganh von, 305 Golden ratio, 121 Goldstine, H., 34 Graph

Acyclicity, 61 Connectivity, 61 Graph-coloring problem, 49 Gray code, 231

H

Halting problem, 419 Hamilton, William Rowan, 53; 160 Hamiltonian circuit, 53 Harel, David, 23; 154

Hash address, 324 Hash function, 323 Hash table, 323 Hashing, 323 Header, 56 Heap, 58; 275; 276

Bottom-up construction, 277 Top-down construction, 279 Heapsort, 281 Height-balanced tree, 272 Hopcroft, John, 271 Homer’s rule, 284 Homer, William George, 284 Horspool, R., 150; 312 Huffman code, 393 Huffman tree, 393 Huffman’s algorithm, 393 Huffman, David, 393 Huxley, Thomas H., 487

I

Icosian Game, 53 Ill-conditioned problem, 433 In place algorithm, 47 Independent set, 428 Information-theoretic lower bounds, 405 Inhomogeneous recurrence, 123 Initial condition, 108; 496 Inorder traversal, 187 Input enhancement, 305 Instability, 433

Integer linear programming, 296 Internal path length, 189 Interpolation, 133 Interpolation search, 240 Inversion, 210

J

Johnson-Trotter algorithm, 228 Josephus problem, 235

К

Kahan, William, 437 Karmarkar, Narenda, 296 Karp, Richard, 419; 489 Kayal, Neeraj, 424

Key, 46; 323

Knuth, Donald Ervin, 284; 328 Kruskal’s algorithm, 378 Kruskal, Joseph, 378

L

Lambert, Johann, 518 Last-in-first-out, 57 Leaf, 65

Least common multiple, 292 Left-to-right binary exponentiation, 287 Lexicographic order, 229 LIFO, 57

Linear congruential method, 130 Linear probing, 327 Linear programming, 295 List, 57

Load factor, 326 Loop of graph, 59 Lower bound, 403 Lower hull, 198 LU-разложение, 259 Lucas, Edouard, 117 Lytton, Edward, 141

M

M-coloring problem, 418 Mantissa, 431 Martello, S., 472 Matroid, 370

Maximization problem, 293 McGreight, Edwarg Meyers, 331 Median, 238 Memory functions, 364 Mergesort, 169 Method of false position, 480 Minimal-change requirement, 227 Minimization problem, 293 Minimum spanning tree, 371 Mode, 250

Moore, J., 48; 150; 316 Morse, Samuel, 392 Motwani, R., 489 Multiplication a la russe, 234 Multiset, 68

Multiway search tree, 66

**N**

Neumann, John von, 34; 489 Newton’s method, 481 Newton, Isaac, 255; 284 Node, 56

Nondeterministic algorithm, 421 NP-complete, 422; 423 NP-сложная задача, 161

О

Open addressing, 325 Open hashing, 325 Optimal solution, 451 Optimality, 42 Order of growth, 78 Order statistic, 238 Ordered tree, 65 Overflow, 432

P

Parallel algorithms, 35 Parent, 64 Parental vertex, 65 Particular solution, 496 Partition, 174 Partition problem, 165 Path, 61 Pattern, 148 Pivot, 174 Pointer, 56 Polya, George, 45 Postorder traversal, 187 Power set, 210; 229 Prefix code, 393 Preorder traversal, 187 Prestructuring, 306 Prim’s algorithm, 371 Prim, R.C., 371 Principle of optimality, 352 Priority, 275 Priority queue, 57; 275 Problem reduction, 291 Problem’s instance, 34 Proper ancestor, 64 Pseudocode, 37 Purdom, P.W., 501

**Q**

Queue, 57 Quickhull, 198 Quicksort, 174

R

Radix sort, 249 Raghavan, P., 489 RAM, 34

Random-access machine, 34 Rear, 57 Recurrence, 108 equation, 496 General solution, 496 relation, 108; 496 Red-black tree, 266 Regula falsi, 480 Relative error, 431 Representative, 381

Right-to-left binary exponentiation, 288

Root of tree, 63

Rooted tree, 64

Rotation, 266

Round-off error, 431

Ruffini, Paolo, 476

Russian peasant method, 234

s

Sahni, S., 471

Saint-Exupery, Antoine de, 40 Saxena, Nitin, 424 Search key, 47

Second-order linear recurrence with constant coefficients, 499 Selection problem, 238 Separate chaining, 325 Sequence, 495 Sequential algorithms, 35 Set, 67

Shell, D.L., 209 Shellsort, 209 Sherman, D., 135 Shor, Peter, 489 Sibling, 65 Significant digits, 431 Simplex method, 156; 296

Single-destination shortest-paths problem, 391

Single-pair shortest-path problem, 391

Single-source shortest-paths problem, 386

Singly linked list, 56

Singular matrix, 261

Smoothness rule, 115; 503

Space efficiency, 39; 75

Spanning tree, 371

Sparse graph, 59

Splay tree, 266

Squashed order, 230

Stable algorithm, 47

Stack, 57

State-space graph, 298 State-space tree, 64; 443 Straight insertion sort, 206 Strassen, V., 192 String, 48\ 55 Binary, 55 Bit, 55 Matching, 48 Strongly connected components, 225 Strongly connected graph, 225 Subgraph, 62 Subset-sum problem, 445 Subtractive cancellation, 432 Subtree, 65

Synthetic division, 286

T

Tarjan, Robert, 84 Ternary search, 237 Time efficiency, 39; 75 Top, 57

Topological sorting, 222 Toth, P., 473

Transitive closure, 226; 345 Traveling salesman problem, 35 Tree, 63

Tree edge, 212; 215; 220 Truncating error, 429 Turing, Alan, 419

u

Undecidable problem, 418 Underflow, 432

Universal set, 68 Upper hull, 198

**V**

Variable-length encoding, 393

Vertex cover, 427

Virtual initialization, 311; 364

w

WarshalTs algorithm, 346

Warshall, S., 346

Weight, 61

Weight matrix, 61

Weighted graph, 61

Williams, John William Joseph, 281

Worst-case efficiency, 81

**A**

Абель, Нильс, 476 Абстрактные типы данных, 69 Агравал, Маниндра, 424 Адельсон-Вельский, Г.М., 267 Адлеман, Лен, 490 Алгоритм, 25\31\ 71 с-приближенный, 463 RSA, 425 Анимация, 135 Базовые операции, 77 Бойера-Мура, 317 Быстрой оболочки, 198 Визуализация, 135 Воршалла, 346

Временная эффективность, 75 Дейкстры, 387 Деления пополам, 476 Джонсона-Троттера, 228 Евклида, 27; 243

Евклида усовершенствованный, 32 Информационно-теоретическая нижняя граница, 405 Исключения Гаусса, 86; 254

С выбором ведущего элемента, 258 Исключения Гаусса-Джордана, 265 Кармаркара, 296 Кодирование, 40 Корректность, 38

Крускала, 378 Ложного положения, 480 Метод проектирования, 36 Методы представления, 37 Недетерминистический, 421 Нерекурсивный, 98 Нестабильность, 433 Нижняя граница, 403 Ньютона, 481 Общность, 39 Оптимальность, 42 Основные операции, 77 Параллельный, 35; 489 Поиск НОД, 28 Последовательный, 35; 489 Приближенный, 35; 461 Прима, 371 Простота, 39

Пространственная эффективность, 75 Рандомизированный, 488 Рекурсивный, 107 Секущих, 480

Сортировки методом подсчета, 51 Сортировки обменный, 47 Сортировки устойчивый, 47 Точный, 35

Тривиальная нижняя граница, 403

Универсальность, 39

Флойда, 350

Хаффмана, 393

Хила, 231

Хорд, 480

Хорспула, 313

Числа Фибоначчи, 122

Штрассена, 192

Этапы проектирования и анализа, 33 Эффективность, 39 Алгоритмика, 23 Аль Хорезми, 31 Анализ, 73

Анимация алгоритма, 135 Антипереполнение, 432

Б

Базовые операции, 77

Байер, Рудольф, 331 Барке, А., 34 Беккер, Рональд, 135 Беллман, Ричард, 339; 352 Берлински, Дэвид, 23 Бинарная сортировка вставкой, 207 Бинарное возведете в степень слева направо, 287 Бинарное возведете в степень справа налево, 288 Бинарное дерево, 184

Внутренние и внешние узлы, 186 Обход в обратном порядке, 187 Обход в прямом порядке, 187 Обход в центрированном порядке, 187 Расширенное, 186 Симметричный обход, 187 Бинарное дерево поиска, 65; 242 Бинарный поиск, 180 Биномиальный коэффициент, 341 Бирн, Роберт, 141 Битовый вектор, 68 Блок-схема, 38 Бойер, Роберт, 48; 150; 316 Браун, К., 501 Брювер, Кобхэм, 203 Быстрое преобразование Фурье, 284

В

Ведущий элемент, 258 Вектор

Битовый, 68 Венгерский метод, 164 Верхняя оболочка, 198 Взаимно простые числа, 39 Взвешенный граф, 61 Визуализация алгоритма, 135 Вильямс, Джон Вильям Джозеф, 281 Виртуальная инициализация, 311; 364 Вороного диаграмма, 200 Вороного многоугольник, 200 Воршалл, С., 346 Выпуклая оболочка, 155 Выпуклое множество, 154 Вычислительная сложность, 417 Вычислительное устройство, 25

Г

Галуа, Эварист, 476

Гамильтон, Вильям Ровен, 53; 160

Гамильтонов цикл, 53; 160

Гаусс, Карл Фридрих, 107; 255; 476

Гаусса метод исключения, 254

Гаусса-Джордана метод исключения, 265

Гете, Иоганн Вольфганг фон, 305

Глур, Питер, 136

Голдстин, Г., 34

Головоломка

Анаграмма, 254

Волк, коза, капуста, 43

Игра Hi-Q, 450

Игра в крестики-нолики, 311

Игра Ним, 244

Икосаэдр, 53

Лабиринт, 220

Магический квадрат, 165

Морской бой, 151

Мосты Кенигсберга, 52

О гайках и болтах, 180

О гирях, 376

О лестнице, 126

О переправе отряда солдат, 209

О переправе через реку, 298

О плитке шоколада, 409

О ревнивых мужьях, 301

Об изобретателе шахмат, 87

Обед в Камелоте, 428

Парадокс дней рождений, 330

Переворачивающиеся блинчики, 244

Переход через мост, 43

Поиск фальшивой монеты, 237

Триомино, 173

Усложненная задача поиска фальшивой монеты, 416 Ханойские башни, 111 Горнер, Вильям Джордж, 284 Горнера схема, 284 Граф, 48

2-раскрашиваемый, 219 Ациклический, 61; 63 Вершина, 48; 58 Вершинное покрытие, 427 Взвешенный, 61

Двудольный, 219

Клика, 427

Маршрут, 61

Матрица смежности, 59

Минимальное вершинное покрытие,

474

Минимальное остовное дерево, 371

Независимое множество, 428

Определение, 58

Ориентированный, 58; 220

Ориентированный ациклический, 221

Остовное дерево, 371

Петля, 59

Плотный, 59

Поиск в глубину, 212

Поиск в ширину, 215

Полный, 59

Представление, 59

Пространства состояний, 298

Путь, 61

Ориентированный, 62 Разреженный, 59 Ребро, 48; 58

Связанные списки смежных вершин, 60 Связный, 61; 62 Связный компонент, 62 Сильно связные компоненты, 225 Сильно связный, 225 Точка сочленения, 215 Транзитивное замыкание, 226; 345 Цикл, 63 Грея код, 231

д

Данциг, Джордж, 296 Двойное хеширование, 328 Дейкстра, Эдсгер В., 387 Декарт, Рене, 292 Деление синтетическое, 286 Дерево, 63

2-3, 266; 272 2-3-4, 266 AVL, 266; 267 В-дерево, 266; 331 Бинарное, 65; 184 Бинарное поиска, 65; 242

Высота, 65

Глубина вершины, 65

Двоичное, 65

Длина внешнего пути, 189

Длина внутреннего пути, 189

Корень, 63

Корневое, 63

Косое, 266

Красно-черное, 266

Минимальное остовное, 371

Остовное, 371

Поворот, 266; 267

Поиска многоканальное, 66

Показатель сбалансированности, 267

Полное, 276

Практически полное, 276 Предок, 64

Принятия решения, 396; 409 Пространства состояний, 64; 443 Рекурсивных вызовов, 113 Сбалансированное по высоте, 272 Упорядоченное, 65 Хаффмана, 393 Детерминант, 118; 262 Джонсона-Троттера алгоритм, 228 Диаграмма Вороного, 200 Диккенс, Чарльз, 339

Динамическое кодирований Хаффмана, 394 Динамическое программирование, 306 Диофантово уравнение, 32 Длина внешнего пути дерева, 189 Длина внутреннего пути дерева, 189 Допустимая область, 159 Допустимое решение, 451 Дуглас, Майкл, 369

**Е**

Евклид, 26

Алгоритм, 27 Игра, 32 Евклидово расстояние, 152; 158 Единичная матрица, 261

3

Заголовок списка, 56 Задача

CNF-выполнимости, 424

т-раскраски, 418 NP-полная, 422; 423 NP-сложная, 161 Выбора, 238

Вычисления выпуклой оболочки, 154; 156

Вычисления моды, 250 Иосифа, 235 Комбинаторная, 50 Коммивояжера, 35; 49; 159; 463 Линейного программирования, 295 Линейного программирования целочисленная, 296 Максимизации, 293 Минимизации, 293 Неразрешимая, 418 О п ферзях, 443 О выпуклой оболочке, 198 О Гамильтоновом цикле, 444 О клике, 165

О минимальном остовном дереве, 371 О паре ближайших точек, 196 О Российском флаге, 310 О размене, 367; 369 О родословной, 311 О рюкзаке, 160; 361 О сумме подмножества, 445 Останова, 419

Перемножения цепочек матриц, 361 Плохо обусловленная, 433 Поиска, 47; 147

Поиска кратчайшего пути между парой вершин, 391 Поиска кратчайших путей в одну вершину, 391 Поиска кратчайших путей из одной вершины, 386 Поиска кратчайших путей между всеми парами вершин, 349 Поиска пары ближайших точек, 152 Построения выпуклой оболочки, 50 Приведение, 291 Принятия решения, 418 Проверки единственности элементов массива, 249 Разбиения, 165

Раскраски графа, 49 Сортировки, 45; 142 Закрытое хеширование, 325; 326 Замыкание транзитивное, 345 Значащие цифры, 431 Золотое сечение, 121

И

Инверсия, 210 Индекс массива, 55 Инициализация виртуальная, 311; 364 Интерполяционный поиск, 240 Интерполяция, 133 Истинный предок, 64 Исчерпывающий перебор, 159

***К***

Каган, Вильям, 437

Кардано, Джироламо, 561

Кармаркар, Наренда, 296

Карп, Ричард, 419; 489

Кассини тождество, 126

Каталана числа, 355; 361

Каял, Нирай, 424

Класс, 69

Ключ, 46; 47; 323

Кнут, Дональд Эрвин, 284; 328

Код

Грея, 231 Префиксный, 393 Хаффмана, 393 Кодирование

Переменной длины, 393 Фиксированной длины, 392 Хаффмана динамическое, 394 Коллизия, 324 Комбинаторные задачи, 50 Компьютер, 25 Корректность алгоритма, 38 Косое дерево, 266 Крамера правило, 263 Красно-черное дерево, 266 Крускал, Джозеф, 378 Куком, Стефен, 424 Кэролл, Льюис, 525

**Л**

Ламберт, Иоганн, 518

Ландис, Е.М., 267 Левин, Леонид, 424 Лексикографический порядок, 229 Лес, 63

Поиска в глубину, 212 Поиска в ширину, 215 Линейное исследование, 327 Линейное программирование, 156; 295 Линейные структуры данных, 55 Линейный конгруэнтный метод, 130 Лист, 65

Литтон, Эдуард, 141 Лопиталя правило, 93 Лукас, Эдуард, 117

м

Магический квадрат, 165 Мак-Грейт, Эдуард Мейерс, 331 Мантисса, 431

Манхэттенское расстояние, 157

Мартелло, С., 472

Массив

Ассоциативный, 55 Индекс, 55 Одномерный, 55 Математическое моделирование, 248 Матрица

Верхнетреугольная, 255 Весов, 61

Детерминант, 118; 262 Единичная, 261 Нижнетреугольная, 260 Обратная, 261 Определитель, 118; 262 Расстояний, 349 Сингулярная, 261 Стоимости, 61 Умножение, 101; 192 Матрица смежности, 59 Матроид, 370

Машина с произвольным доступом, 34

Медиана, 238

Метод

Ветвей и границ, 451

С выбором наилучшего варианта, 453

Грубой силы, 141 Декомпозиции, 168 Деления пополам, 476 Исчерпывающего перебора, 159 Ложного положения, 480 Ньютона, 481

Обратной подстановки, 109 Поиска с возвратом, 442 Преобразования, 247 Проектирования алгоритма, 36 Противника, 405 Русский крестьянский, 234 Секущих, 480

Уменьшения на постоянный множитель, 204 Уменьшения переменного размера, 206 Уменьшения размера на постоянную величину, 203 Хорд, 480 Минимальное остовное дерево, 371 Минимальных изменений требование, 227 Многоканальное дерево поиска, 66 Многоугольник Вороного, 200 Множество, 67

Показательное, 210; 229 Универсальное, 68 Мода, 250

Морзе, Сэмюэль, 392 Мотвани, Р., 489 Мультимножество, 47; 68 Мур, Дж., 48; 150; 316

н

Наилучший случай, 81 Наименьшее общее кратное, 292 Наихудший случай, 81 Начальное условие, 108; 496 Нейман, Джон фон, 34; 489 Непересекающиеся подмножества, 381 Представитель, 381 Нижнетреугольная матрица, 260 Нижняя оболочка, 198 Ньютон, Исаак, 255; 284

О

Обменный алгоритм сортировки, 47

Обобщенный член последовательности, 495

Обработка строк, 48 Обратная матрица, 261 Обратное ребро, 212; 221 Опорный элемент, 174 Определитель, 118; 262 Оптимальное решение, 451 Опустошение, 432 Ориентированный граф, 220 Ориентированный цикл, 221 Основные операции, 77 Остовное дерево, 371 Открытое хеширование, 325 Очередь, 57 Голова, 57 Постановка, 57 С приоритетом, 57 Удаление, 57 Хвост, 57 Очередь с приоритетами, 275 Ошибка

Абсолютная, 431 Округления, 431 Относительная, 431 Усечения, 429

п

Пакет, 68

Парадокс дней рождений, 330 Параллельные алгоритмы, 35 Паскаля треугольник, 341 Переполнение, 432 Перестановка, 227 Пирамида, 58; 275; 276

Восходящее построение, 277 Неубывающая, 375 Нисходящее построение, 279 Фибоначчи, 390 Пирамидальная сортировка, 281 Плотный порядок, 230 Подграф, 62 Поддерево, 65 Поиск, 47

Бинарный, 180 Интерполяционный, 240 Подстроки, 48; 148

Последовательный, 147 Поиск с возвратом, 442 Пойа, Джорж, 45

Показатель сбалансированности, 267 Показатель степени, 431 Показательное множество, 210; 229 Полином Тейлора, 429 Поперечное ребро, 215; 221 Поразрядная сортировка, 249 Порядковая статистика, 238 Порядок роста, 78 Последовательность, 495 Последовательные алгоритмы, 35 Потеря значащих разрядов, 432 Потомок, 64 Правило

Гладкости, 503 Крамера, 263 Сглаживания, 115 Предварительная структуризация, 306 Предок, 64

Истинный, 64 Собственный, 64 Префиксный код, 393 Приближенный алгоритм, 35 Приведение задачи, 291 Прим, Р., 371

Принцип оптимальности, 352 Приоритет, 275

Простая сортировка вставкой, 206 Прямое ребро, 221 Псевдокод, 37

Псевдослучайные числа, 130 Пурдом, П.В., 501

Р

Рагаван, П., 489 Расстояние

Евклидово, 152; 158 Манхэттенское, 157 Расширяемое хеширование, 329 Ребро дерева, 212; 215; 220 Рекуррентное соотношение, 496 Рекуррентное уравнение, 496 Декомпозиции общее, 504 Линейное второго порядка

с постоянными коэффициентами, 499

Обобщенное линейное n-ой степени с постоянными коэффициентами, 501

Общее решение, 496 Однородное, 499 Частное решение, 496 Рекуррентность, 108; 496 Рекуррентные уравнения, 108 Рекурсия

Неоднородная, 123 Решето Эратосфена, 29 Родитель, 64

Родительская вершина, 65 Родственные вершины, 64 Руффини, Паоло, 476

**С**

Саксена, Нитин, 424 Сахни, С., 471

Сент-Экзюпери, Антуан де, 40 Симплекс-метод, 156; 296 Сингулярная матрица, 261 Синтетическое деление, 286 Словарь, 69 Сортировка, 45 Быстрая, 174 Анализ, 176 Разбиение, 174 Вставкой, 206 Вставкой бинарная, 207 Выбором, 143 Пирамидальная, 281 Подсчетом распределения, 308 Подсчетом сравнений, 307 Поразрядная, 249 Пузырьковая, 144 Слиянием, 169 Анализ, 170 Топологическая, 222 Шелла, 209; 211 Список, 57

Заголовок, 56 Связанный, 56

Связанный двунаправленный, 56 Связанный однонаправленный, 56

Стек, 57

Вершина, 57 Степень сжатия, 394 Стирлинга формула, 93 Строка, 48; 55 Бинарная, 55 Двоичная, 55 Операции, 55 Текстовая, 48 Структура данных, 54 Линейная, 55 Схема Горнера, 284 Схемы приближений, 471

т

Таржан, Роберт, 84

Текст, 148

Теория графов, 48

Тернарный поиск, 237

Тождество Кассини, 126

Топологическая сортировка, 222

Тот, П., 473

Точный алгоритм, 35

Транзитивное замыкание, 345

Требование минимальных изменений, 227

Треугольник Паскаля, 341

Тьюринг, Алан, 419

**У**

Угловая точка, 156 Узел, 56

Внешний, 186 Внутренний, 186 Указатель, 56 Нулевой, 56

Фибоначчи пирамида, 390 Фибоначчи числа, 116; 119; 339 Фибоначчи, Леонардо, 119 Флавий, Иосиф, 235 Флойд, Р., 350 Формула трапеций, 429 Форсайт, Джордж, 14; 435 Функция с запоминанием, 364 Фурье быстрое преобразование, 284

X

Хаксли, Томас, 487

Характеристическое уравнение, 120; 499 Харел, Дэвид, 23; 154 Хаффман, Дэвид, 393 Хеш-адрес, 324 Хеш-таблица, 323 Хеш-функция, 323 Хеширование, 323 Двойное, 328 Закрытое, 325; 326 Кластеризация, 328 Коэффициент заполнения, 326 Линейное исследование, 327 Открытое, 325 Расширяемое, 329 С открытой адресацией, 325; 326 С раздельными цепочками, 325 Хопкрофт, Джон, 271 Хорспул, Р, 150; 312

ц

Цикл

Гамильтонов, 160

Улучшение входных данных, 305 Умножение матриц, 192 Умножение по-русски, 234 Универсальное множество, 58; 68 Универсум, 58; 68 Уравнение

Диофантово, 32 Характеристическое, 499 Устойчивый алгоритм сортировки, 47

**Ф**

Факториал, 107

Числа

Каталана, 355; 361 Фибоначчи, 116; 119; 339

ш

Шаблон, 148 Шелл, Д., 209 Шерман, Д., 135 Шор, Питер, 489 Штрассен, В., 192 Штрассена алгоритм, 192

**Э**

Эвристика, 462 Эдисон, Томас, 441 Эйлер, Леонард, 52; 255 Эйнштейн, Альберт, 73 Экземпляр задачи, 34

Экспонента, 431 Экстраполяция, 133 Эратосфен, 29 Эффективность

Асимптотические классы, 94

Ананий В. Левитин

Алгоритмы: введение в разработку и анализ

Литературный редактор Ж.Е. Прусакова Верстка А.Н. Полинчик Художественный редактор В.Г. Павлютин Корректоры Л.А. Гордиенко,

***Ж.Е. Прусакова***

Издательский дом “Вильямс”.

127055, Москва, ул. Лесная, д. 43, стр. 1.

Подписано в печать 10.03.2006. Формат 70x100/16.

Гарнитура Times. Печать офсетная.

Уел. печ. л. 46,4. Уч.-изд. л. 30,1.

Тираж 3000 экз. Заказ № 2966.

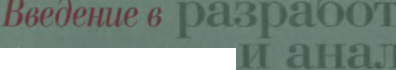
Отпечатано с диапозитивов в ОАО “Печатный двор” им. А. М. Горького.

197110, Санкт-Петербург, Чкаловский пр., 15.

**Ананий Левитин**



Алгоритмы



**Университет Вилла нова**

■



9785845909879

9 785845 909879

В этой книге используется новая разработанная автором таксономия методологии прое ктирования алгоритмов. Новая таксономия позволяет охватить в одной книге множество классических алгоритмов, которые иначе было бы невозможно собрать под одной обложкой при условии последовательного изложения материала.

Методы проектирования алгоритмов рассмотрены в книге необычайно широко — как инструментарий общего назначения для решения различных задач. В частности, возможности этих методов продемонстрированы путем применения для решения различных известных головоломок (наподобие той, что показана на первой странице обложки).

В представлении материала автор делает упор на понимание идей, а не на формальное пояснение работы алгоритмов, чем зачастую грешат другие книги, посвященные алгоритмам. Ясность изложения основана на многолетнем опыте автора в преподавании алгоритмов для студентов и аспирантов.

В книге содержится более 600 упражнений, включая такие, которые требуют обращения к ресурсам World Wide Web. Имеющиеся в книге указания помогут читателям в случае возникновения трудностей при решении упражнений.

**Издательским дом "Вильяме**

**\vww.williamspuMishin£.eoni**

Addison-Weslev

"Способ описания алгоритмов в этой книге — как с точки зрения их структуры, так и работы — превосходен”.

Джон Мейер, Мичиганский университет

"Упражнения представляют собой отличную комбинацию исследования работы алгоритмов, их проектирования, математического доказательства и программной реализации”.

Ричард Бори, университет в Алабаме

ОБ АВТОРЕ

Ананий Левитин — профессор информатики в университете Вилланова. Его знаменитая статья A New Road Map of Algorithm Design Techniques: Picking Up Where the Traditional Classification Leaves Отбыла опубликована в апреле 2000 года в Dr. Dobb's Journal. Кроме того, он автор докладов по методологии обучения дпгоритмам на конференциях SIGSCE. включая доклад 2002 года, посвященный использованию головоломок в лекциях по алгоритмам.

ISBN 5-8459-0987-2

06073

image261

1. Древняя народная головоломка. На берегу реки находятся крестьянин, волк, коза и кочан капусты. Крестьянин должен в своей лодке пере­везти их на другой берег. Однако в лодке есть только два места — для крестьянина и еще одного объекта (т.е. либо волка, либо козы, либо капусты). В отсутствие крестьянина волк может съесть козу, а ко­за — капусту. Помогите крестьянину решить эту задачу или докажите, что она не имеет решения. {Примечание. Для определенности будем считать, что крестьянин является вегетарианцем, но терпеть не может капусту, поэтому в лодке он не может съесть ни козу, ни капусту. Кро­ме того, крестьянин должен учитывать, что он столкнулся с редкой породой волка, которая занесена в Красную книгу.)

1. Для каждого из приведенных ниже алгоритмов определите: 1) есте­ственные единицы измерения размера входных данных; 2) основную операцию алгоритма; 3) будет ли изменяться количество основных опе­раций, выполняемых алгоритмом, в зависимости от используемого на­бора входных данных одинакового размера.

а) Вычисление суммы п чисел.

б) Вычисление п\.

в) Поиск наибольшего элемента в списке из п чисел.

г) Алгоритм Евклида.

д) Решето Эратосфена.

е) Алгоритм умножения в столбик двух n-разрядных десятичных це­лых чисел.

1. а) Рассмотрите алгоритм сложения двух матриц размером п х п, осно­

ванный на определении сложения матриц. Назовите его основную операцию. Определите зависимость количества основных операций как функцию от порядка матрицы п. Найдите зависимость количе­ства основных операций как функцию от числа элементов в матри­цах.

б) Выполните аналогичное упражнение для алгоритма умножения двух матриц размером п х п, соответствующего определению этой опе­рации.

1. Рассмотрите вариант алгоритма поиска методом последовательного пе­ребора, который бы возвращал количество элементов в списке, совпа­дающих с заданным ключом поиска. Будет ли его эффективность от­личаться от эффективности классического алгоритма поиска методом последовательного перебора?
2. а) В ящике хранится 22 перчатки: 5 пар красных, 4 пары желтых и 2 па­

ры зеленых. Предположим, что вы выбираете их в темноте наугад

1 / n 1 2 1 ^ 1 2 11 1 2

-п (п — 1) = -п п ^ -п — -п-п = -п .

У вас не должно сложиться ошибочное мнение по поводу того, что приве­денным выше планом анализа нерекурсивных алгоритмов можно пользоваться всегда. Бессистемное изменение значения переменной цикла, слишком сложная для анализа сумма и трудности, присущие анализу эффективности алгоритма для среднего случая, — вот только малая часть тех проблем, которые могут оказаться

1. а) Приведите пример алгоритма, который нельзя рассматривать как

применение метода грубой силы,

б) Приведите пример задачи, которую нельзя решить при помощи ал­горитма, основанного на грубой силе.

3. Использует ли каждый из алгоритмов задач 4-6 упражнений 2.3 метод грубой силы?

1. Чему равно наименьшее и наибольшее возможное количество цифр в произведении двух гг-значных чисел?

3. а) Докажите тождество alogbC = clogfaa, которое дважды было исполь­

зовано в разделе 4.5. б) Почему запись nlog2 3 лучше записи 3log2 72 в решении уравнения для М (п)?

4. а) Почему умножение на 10п не включается в общее количество умно­

жений М (п) в алгоритме перемножения больших целых чисел? б) Кроме предположения о том, что п является степенью 2, при написа­нии рекуррентного соотношения для М (п) мы делаем для простоты еще одно, более тонкое предположение, которое не всегда истинно (но это не влияет на конечный ответ). Что это за предположение?

5. Сколько сложений цифр делается при умножении двух n-значных чи­сел в столбик? (Переносами можно пренебречь.)

1. а) Разработайте на основе метода декомпозиции алгоритм решения од­

номерной версии задачи о паре ближайших точек, т.е. задачи поиска пары ближайших чисел во множестве из п действительных чисел. Определите класс эффективности предложенного вами алгоритма, б) Насколько хорош этот алгоритм для решения поставленной задачи?

3. Реализуйте описанный в данном разделе алгоритм декомпозиции для поиска пары ближайших точек на своем любимом языке программи­рования.

4. Найдите в Web визуализацию алгоритма решения задачи поиска па­ры ближайших точек. Какой алгоритм представляет найденная вами визуализация?

5. Многоугольник Вороного для точки Р из множества точек S на плос­кости определяется как множество всех точек плоскости, расстояние от которых до точки Р меньше, чем до любой другой точки множества

S. Объединение всех многоугольников Вороного для точек множества S образует диаграмму Вороного для множества S.

а) Как выглядит диаграмма Вороного для множества из трех точек?

б) Найдите в Web визуализацию алгоритма для генерации диаграм­мы Вороного и рассмотрите несколько примеров диаграмм. Може­те ли вы указать, каким образом обобщить решение предыдущего вопроса?

1. Отряд из п солдат должен переправиться через широкую и глубокую реку, через которую нет моста. Солдаты заметили лодку с двумя маль­чишками, но лодка так мала, что в ней могут поместиться либо два мальчишки, либо один солдат. Как солдатам переправиться через ре­ку и после переправы возвратить лодку мальчишкам? Сколько раз при этом лодка проплывет от берега к берегу?

1. а) Если измерять размер экземпляра задачи по вычислению наиболь­

шего общего делителя тип величиной второго параметра п, то насколько может уменьшиться размер экземпляра задачи при вы­числении gcd (тп, п) при помощи алгоритма Евклида? б) Докажите, что размер задачи поиска наибольшего общего делителя после двух последовательных итераций алгоритма Евклида умень­шается как минимум в два раза.

1. а) Напишите псевдокод нерекурсивной реализации алгоритма решения

задачи выбора путем разбиения, б) Напишите псевдокод рекурсивной реализации этого алгоритма.

1. Покажите, что эффективность наихудшего случая алгоритма для реше­ния задачи выбора путем разбиения квадратична.
2. Выведите формулу, лежащую в основе интерполяционного поиска.
3. Приведите пример входных данных для наихудшего случая интерполя­ционного поиска и покажите линейность данного алгоритма в наихуд­шем случае.
4. а) Найдите наименьшее значение п, для которого log2 log2 n 1 пре­

вышает 6.

б) Определите, какие из приведенных утверждений истинны:

1) log log n € о (log n) 2) log log n € © (log n)

1. а) Набросайте алгоритм для поиска наибольшего ключа в бинарном

дереве поиска. Можно ли классифицировать ваш алгоритм как ал­

горитм с переменным уменьшением размера задачи?

б) Какова временная эффективность вашего алгоритма в наихудшем случае?

1. а) Набросайте алгоритм для удаления ключа из бинарного дерева по­

иска. Можно ли классифицировать ваш алгоритм как алгоритм с пе­ременным уменьшением размера задачи? б) Какова временная эффективность вашего алгоритма в наихудшем случае?

1. Вспомним, что медианой множества из п чисел называется его [гг/2] в порядке возрастания элемент (т.е. медиана больше одной полови­ны элементов множества и меньше другой). Разработайте алгоритм для поиска медианы с использованием предварительной сортировки и определите класс его эффективности.
2. Рассмотрим задачу поиска расстояния между двумя ближайшими чис­лами в массиве из п чисел (расстояние между двумя числами х и у вычисляется как \х —у\).

а) Разработайте алгоритм с использованием предварительной сорти­ровки для решения данной задачи и определите его класс эффек­тивности.

б) Сравните эффективность разработанного вами алгоритма с эффек­тивностью алгоритма грубой силы (см. упражнение 1.2.9).

1. Пусть А = (ai, a2,..., ап} и В = (&i, &2, • • •, Ьт} — два множества чисел. Рассмотрим задачу поиска их пересечения, т.е. множества С всех чисел, которые входят как в А, так ив В.

а) Разработайте алгоритм грубой силы для решения данной задачи и определите класс его эффективности.

1. а) Постройте пирамиду для чисел 1, 8, 6, 5, 3, 7, 4 при помощи восхо­

дящего алгоритма.

б) Постройте пирамиду для чисел 1, 8, 6, 5, 3, 7, 4 при помощи нисхо­дящего алгоритма.

в) Всегда ли восходящий и нисходящий алгоритмы построения пира­миды дают одну и ту же пирамиду?

**который ныне известен как один из ведущих ученых в области кибернетики. Его многотомник Искусство программирования [65, 66, 67] остается одной из наиболее полных и важных книг об**

б) найдите наибольшее количество сравнений ключей при успешном поиске в полученной таблице;

в) найдите среднее количество сравнений ключей при успешном по­иске в полученной таблице.

1. Приведите пример использования индексов в реальной жизни, без ком­пьютеров.

2. а) Докажите равенство

/i-i

1 + ^2 r™/2li-1 {\т/2] - 1) + 2 |"m/2]ft-1 = 4 [m/2]/l-1 - 1, 1=1

использовавшееся в выводе верхней границы высоты В-дерева (7.7).

б) Завершите вывод неравенства (7.7), дающего верхнюю границу вы­соты В-дерева.

3. Найдите наименьшее значение порядка В-дерева т, которое гаранти­рует, что число обращений к диску при поиске в файле со 100 миллио­нами записей не превысит 3. При этом считаем, что корневая страница хранится в оперативной памяти.

4. Изобразите В-дбрево, полученное вставкой 30 и 31 в В-дерево на рис. 7.8, в предположении, что лист не может содержать более трех элементов.

5. Опишите алгоритм для поиска наибольшего ключа В-дерева.

6. а) Нисходящее 2-3-4-дерево (top-down 2-3-4-tree) представляет собой

В-дерево порядка 4 то следующей модификацией операции встав­ки. Когда в процессе поиска листа для нового ключа встречается заполненный узел (т.е. узел с тремя ключами), узел разбивается на два путем передачи среднего ключа в родительский узел (если же полным узлом оказывается корень, то создается новый корень для среднего ключа). Постройте нисходящее 2-3-4-дерево путем вставки в изначально пустое дерево ключей 10, 6, 15, 31, 20, 27, 50, 44, 18.

б) В чем заключается главное преимущество такой процедуры вставки по сравнению со вставкой в 2-3-дерево? В чем ее недостатки?

1. Приведите пример экземпляра задачи о размене, для которой жадный алгоритм не дает оптимального решения.

1. Напишите псевдокод жадного алгоритма для задачи о размене для сум­мы п и монет с номиналами d\ > cfo > • • • > dm в качестве входных данных. Чему равна временная эффективность вашего алгоритма как функция от п?
2. Рассмотрим задачу составления расписания выполнения п заданий с известными продолжительностями £i,...,£n на одном процессоре. Задания могут выполняться по одному в любом порядке. Необходимо найти расписание, которое минимизирует время, затраченное на все задания в системе (время, затрачиваемое на задание, представляет со­бой сумму времени, затраченного на выполнение задания, и времени ожидания выполнения задания).

а) Разработайте жадный алгоритм для решения этой задачи.

б) Всегда ли жадный алгоритм приводит к оптимальному решению?

1. а) Разработайте жадный алгоритм для задачи о назначениях (см. раз­

дел 3.4).

б) Всегда ли жадный алгоритм приводит к оптимальному решению?

1. Задача о гирях. Найдите оптимальное множество п весов гирь {tui, W2, ... ,tun}, позволяющее взвесить на рычажных весах любой целочис­ленный вес от 1 до наибольшего возможного значения W, в случае, если

а) гири могут размещаться только на одной чашке весов;

б) гири могут размещаться на обеих чашках весов.

1. Докажите, что классический рекурсивный алгоритм решения задачи о ханойской башне (раздел 2.4) делает минимальное количество пере­мещений дисков, необходимых для решения задачи.

3. Рассмотрим задачу определения более легкой фальшивой монеты сре­ди п одинаковых по виду монет при помощи рычажных весов. Можем ли мы использовать то же информационно-теоретическое доказатель­ство, что и приведенное в тексте для количества вопросов в игре на отгадывание, чтобы сделать вывод о том, что для определения фальши­вой монеты требуется как минимум [log2 п] взвешиваний в наихудшем случае?

1. Теория классификации и систематизации сложноорганизованных областей действительности, имеющих обычно иерархическое строение (органический мир, объекты географии, геологии, язы­кознания, этнографии). **Современный словарь иностранных слов**, 3-е изд., М.: Рус. яз., 2000. — **Прим. ред.** [↑](#footnote-ref-1)
2. От английского “greatest common divisor”. — **Прим. ред.** [↑](#footnote-ref-2)
3. Речь идет о структуре вычислительного устройства, описанного выдающимся американским математиком венгерского происхождения Джоном фон Нейманом (John von Neumann) (1903-1957) в сотрудничестве с А. Барксом (A. Burks) и Г. Голдстином (Н. Goldstine) в 1946 году. [↑](#footnote-ref-3)
4. Чаще всего английского, хотя встречаются случаи использования в псевдокоде русского языка. — **Прим. перев.** [↑](#footnote-ref-4)
5. Я обнаружил это высказывание по поводу простоты конструкции в сборнике рассказов Джона Бентли (Jon Bentley) [15]. Этот сборник посвящен особенностям проектирования и реализации алго­ритмов и имеет совершенно логичное название **Programming Pearls** [15]. Настоятельно рекомендую почитать рассказы Бентли и Сент-Экзюпери. Не пожалеете! [↑](#footnote-ref-5)
6. Это название можно перевести на русский язык как **Удовольствие от алгоритмов.** — **Прим. ред.** [↑](#footnote-ref-6)
7. Современная народная головоломка. Предположим, что четыре челове­ка движутся по дороге в одном направлении и хотят перейти через мост. Ваша задача помочь им переправиться на другой берег за 17 минут. На дворе ночь, и у них только один фонарик. По мосту одновременно мо­гут следовать не более двух человек (т.е. либо один, либо два), причем у одного из них обязательно должен быть фонарик. Фонарик нельзя перебросить с одного берега реки на другой, его можно только пере­нести по мосту обратно. Каждый человек затрачивает разное время на прохождение моста: первый — 1 минуту, второй — 2 минуты, третий — 5 минут и четвертый — 10 минут. Если по мосту передвигается пара людей, то они идут со скоростью более медлительного из них. Напри­мер, если по мосту передвигается первый и четвертый человек, то они достигнут противоположного берега через 10 минут. Если четвертый человек будет возвращать фонарь на другой берег, то с момента начала задачи пройдет 20 минут, и вы не решите задачу. {Примечание. В In­ternet бродят слухи, что одна из известных компаний по производству программного обеспечения, расположенная вблизи Сиэтла, предлагает решить эту задачу претендентам на собеседовании.) [↑](#footnote-ref-7)
8. ’Очевидно, что для сортировки в порядке убывания достаточно заменить вид сравнения элемен­тов последовательности на обратный. — **Прим. ред.** [↑](#footnote-ref-8)
9. **Мультимножество —** это множество, в котором несколько принадлежащих ему элементов могут иметь одинаковые значения. [↑](#footnote-ref-9)
10. Согласно оценке группы исследователей из университета Нотр-Дама (University of Notre Dame), это количество ссылок составляет всего 19 [6]. [↑](#footnote-ref-10)
11. **Математический термин, означающий некоторое множество, фиксированное в рамках данной математической теории и содержащее в качестве элементов все объекты, рассматриваемые в этой теории. Употребляется как синоним термина универсальное множество. Математическая энцик­лопедия, т. 5, М.: Сов. энц., 1985. — Прим. ред.** [↑](#footnote-ref-11)
12. **Изначально термин “heap” использовался в контексте пирамидальной сортировки (heapsort), но в последнее время его основной смысл изменился, и он стал обозначать память со сборкой мусора (в частности, в языках программирования Lisp и Java) и переводиться как “куча”. Однако в данной книге термину heap (который здесь переводится как “пирамида”) возвращен его первоначальный смысл. — Прим. ред.** [↑](#footnote-ref-12)
13. **В этих определениях фразу “относительно небольшое” следует понимать как количество ребер, малое относительно количества ребер полного графа. — Прим. ред.** [↑](#footnote-ref-13)
14. **В общем случае значения элементов матрицы смежности равны не только 0 или 1 (т.е. матрица не является булевой). Каждый элемент матрицы, находящийся на пересечении г-й строки и j-го столбца, соответствует количеству ребер, соединяющих г-ю и j-ю вершины графа. — Прим. ред.** [↑](#footnote-ref-14)
15. **Подграфом (subgraph) графа G = (V,E) является граф G' = (V\E'), такой, что V' С V и Е' СЕ.** [↑](#footnote-ref-15)
16. **В нашей книге это первый пример компромисса между временем выполнения алгоритма и тре­буемым объемом оперативной памяти. Более подробно мы поговорим об этом в главе 7.** [↑](#footnote-ref-16)
17. В некоторых алгоритмах для оценки размерности входных данных может использоваться сразу несколько параметров (например, количество вершин и ребер для тех алгоритмов, в которых граф представляется в виде связанных списков смежных вершин). [↑](#footnote-ref-17)
18. **Кроме компьютеров с так называемой RISC-архитектурой. В качестве примера сравните вре­менные характеристики команд, приведенные в [[61], с. 185-186].** [↑](#footnote-ref-18)
19. 2 2 2 22 4

    Таким образом, для данного случая можно положить сг = 1/4, ci = 1/2,

    ?>о = 2. [↑](#footnote-ref-19)
20. Есть и четвертый случай, когда предела попросту не существует, однако он встречается доволь­но редко в реальной практике анализа алгоритмов. Тем не менее эта особенность приводит к тому, что подход к оценке порядка роста, основанный на вычислении предела отношения двух рассмат­риваемых функций, является менее универсальным, чем тот, в котором используется определения множеств О, П, 0. [↑](#footnote-ref-20)
21. **Иногда при анализе нерекурсивных алгоритмов применяют не суммы, а рекуррентные отноше­ния, выражающие зависимость количества выполняемых основных операций алгоритма. Создание и решение рекуррентных отношений более типично для анализа рекурсивных алгоритмов (см. раз-**

    **дел. 2.4).** [↑](#footnote-ref-21)
22. **В качестве альтернативы мы могли бы подсчитать количество выполняемых операций сравне­ния п = 0 или, что то же самое, количество вызовов функции F (п) из самой себя, выполняемых в алгоритме (см. задачу №2 упражнения 2.4).** [↑](#footnote-ref-22)
23. **Константу ф называют золотым сечением (golden ratio). В древности такое отношение длин сторон прямоугольника считали наиболее оптимальным и приятным для восприятия человеческим глазом. Поэтому его сознательно использовали древние архитекторы и скульпторы.** [↑](#footnote-ref-23)
24. а) Чему равна эффективность алгоритма вычисления ап, основанного

    на методе грубой силы, как функция от п? А как функция количества бит в двоичном представлении числа п?

    б) Если вы вычисляете ап mod т, где а > 1, ап- большое нату­ральное число, то как вы справитесь с проблемой очень больших значений ап? [↑](#footnote-ref-24)
25. а) Разработайте алгоритм для вычисления значения полинома с помо­

    щью метода грубой силы

    р(х) = апхп + ап-iTn\_1 4 Ь aix + qq [↑](#footnote-ref-25)
26. **Набор кораблей в отечественном “морском бое”, по детским воспоминаниям, следующий, один корабль длиной 4 квадрата, два корабля длиной по 3 квадрата, три — по два квадрата и четыре — по одному. Специальных имен они не носили, а назывались по числу квадратов — от однопалубного до четырехпалубного. — Прим. перев.** [↑](#footnote-ref-26)
27. **Под треугольником, прямоугольником и выпуклым многоугольником мы подразумеваем об­ласть — т.е. множество точек внутри и на границе указанной фигуры.** [↑](#footnote-ref-27)
28. **Для простоты мы считаем, что никакие три точки не лежат на одной прямой. Модификация, необходимая для общего случая, остается читателям в качестве самостоятельного упражнения.** [↑](#footnote-ref-28)
29. **Как мы увидим в разделе 10.2, этот теоретический минимум равен [log2n!"| «** « \п **log2** п — **1.44** п\**.** [↑](#footnote-ref-29)
30. Молодой Хоар изобрел этот алгоритм при попытке отсортировать слова в словаре русского языка для проекта машинного перевода с русского языка на английский. Дадим слово Хоару: “Моя первая мысль была — как бы выполнить задание при помощи пузырьковой сортировки, но тут меня настигло счастливое озарение и второй мыслью стала мысль о быстрой сортировке”. Трудно не согласиться с его оценкой происшедшего: “Я был очень счастлив. Какое везение — начать карьеру в кибернетике с открытия нового алгоритма сортировки!” [51] [↑](#footnote-ref-30)
31. Вычислите 2101 • ИЗО с применением описанного в тексте алгоритма декомпозиции. [↑](#footnote-ref-31)
32. Рассмотрим еще одну версию алгоритма декомпозиции для решения двумерной задачи о паре ближайших точек. В нем мы просто заново сортируем каждое из двух множеств С\ и С2 в возрастающем порядке по координате у при каждом рекурсивном вызове. Полагая, что сорти­ровка выполняется при помощи алгоритма сортировки слиянием, запи­шите приблизительное рекуррентное соотношение для времени работы такого алгоритма в наихудшем случае и решите его для п = 2к. [↑](#footnote-ref-32)
33. а) Разработайте рекурсивный алгоритм на основе уменьшения размера

    на единицу для поиска позиции наименьшего элемента в массиве из п элементов.

    б) Определите временную эффективность этого алгоритма и сравните ее с эффективностью алгоритма грубой силы для решения той же задачи. [↑](#footnote-ref-33)
34. **Чтобы было понятнее, опишем возможный процесс сортировки массива из 16 элементов с использованием последовательности 8,4,2,1. При первом проходе они делятся на 8 групп по** [↑](#footnote-ref-34)
35. **элемента, индексы которых отличаются на 8: (ai, ag),..., (as, а\б), и выполняется сорти­** [↑](#footnote-ref-35)
36. **ровка в пределах каждой группы. При втором проходе получаем 4 группы по 4 элемента — (ai,a5,ag,ai3),..., (a4,ag,ai2,ai6), и сортируем каждую группу на третьем проходе получа­** [↑](#footnote-ref-36)
37. **ются и сортируются две группы по 8 записей (ai, аз,..., ais), (а2, ..., а1б), а на последнем** [↑](#footnote-ref-37)
38. **проходе весь массив рассматривается как одна группа. — Прим. ред.** [↑](#footnote-ref-38)
39. **Честь открытия ряда таких применений в 1970-х годах принадлежит Джону Хопкрофту (John Hopcroft) и Роберту Таржану (Robert Tarjan) [53, 117]; за эти (и другие) работы они удостоились премии Тьюринга — наивысшей награды в области теоретической кибернетики.** [↑](#footnote-ref-39)
40. **В более сложной постановке задачи вам ничего не известно о том, как именно соотносятся вес фальшивой и подлинной монеты, и даже неизвестно, имеется ли она среди п монет. Такая, более сложная версия задачи будет рассмотрена в разделе 10.2.** [↑](#footnote-ref-40)
41. **По другим сведениям, — каждого третьего. О задаче Иосифа и ее разновидностях можно прочесть в книге У. Болл, Г. Коксетер. Математические эссе и развлечения. — М.: Мир, 1986 (стр. 43-47). — Прим. ред.** [↑](#footnote-ref-41)
42. 3) log log n 6 (log n) [↑](#footnote-ref-42)
43. **Алгоритмы поразрядной сортировки** (radix sort) **линейны при рассмотрении зависимости от общего количества входных битов. Эти алгоритмы работают путем сравнения отдельных битов или частей ключей, а не сравнения ключей целиком. Хотя время работы таких алгоритмов пропорцио­нально количеству входных битов, по сути они остаются алгоритмами класса п log п, так как для наличия п различных входных ключей количество битов в одном ключе должно составлять как минимум log2 п.** [↑](#footnote-ref-43)
44. Разработайте алгоритм с использованием предварительной сорти­ровки для решения данной задачи и определите класс его эффек­тивности.

    1. Рассмотрим задачу поиска наибольшего и наименьшего элементов в массиве их п чисел.

    а) Разработайте алгоритм с использованием предварительной сорти­ровки для решения данной задачи и определите класс его эффек­тивности.

    б) Сравните эффективность трех алгоритмов: 1) алгоритма грубой си­лы, 2) алгоритма с использованием предварительной сортировки и 3) алгоритма декомпозиции (см. упражнение 4.1.2). [↑](#footnote-ref-44)
45. **Метод назван по имени Карла Фридриха Гаусса (Carl Friedrich Gauss) (1777-1855), который, как и другие гиганты в истории математики, например Исаак Ньютон (Isaac Newton) и Леонард Эйлер (Leonard Euler), выполнил ряд фундаментальных работ как в области теоретической, так и вычислительной математики.** [↑](#footnote-ref-45)
46. **Подробнее об ошибках округления мы поговорим в разделе 10.4.** [↑](#footnote-ref-46)
47. Как упоминалось в разделе 2.1, на некоторых компьютерах умножение не обязательно более дорогостоящее по сравнению со сложением и вычитанием. Для данного алгоритма это не имеет [↑](#footnote-ref-47)
48. Некоторые авторы требуют, чтобы ключ в каждом узле не превышал ключи в дочерних узлах. [↑](#footnote-ref-48)
49. Набросайте схему алгоритма для проверки того, является ли пирамидой массив Н[1..п\, и определите его временную эффективность. [↑](#footnote-ref-49)
50. а) Найдите наименьшее и наибольшее количество ключей, которые

    может содержать пирамида высотой /г.

    б) Докажите, что высота пирамиды с п узлами равна [log2 п\. [↑](#footnote-ref-50)
51. Эта формула получена путем рассмотрения ord (с\*) как цифр числа в системе счисления по основанию С, вычисления его десятичного значения по схеме Горнера и поиска остатка от деления получившегося числа на га. [↑](#footnote-ref-51)
52. **Эта задача была решена в 1962 году молодым аспирантом Дональдом Кнутом (Donald Knuth),** [↑](#footnote-ref-52)
53. **алгоритмах.**

    1. Для входных данных 30, 20, 56, 75, 31, 19 и хеш-функции h(K) = = К mod 11

    а) постройте открытую хеш-таблицу; [↑](#footnote-ref-53)
54. **Узел, изображенный на рис. 7.7, называется n-узлом (n-node). Таким образом, все узлы в клас­сическом бинарном дереве поиска являются 2-узлами; 2-3-дерево, рассматривавшееся в разделе 6.3, содержит 2-узлы и 3-узлы.** [↑](#footnote-ref-54)
55. При использовании реализации с разделением на приграничные и незамеченные вершины все незамеченные вершины, смежные с и\*, должны быть перенесены во множество приграничных вершин. [↑](#footnote-ref-55)
56. В скобках показаны метки, указывающие ближайшую вершину дерева и вес ребра; выбираемые для добавления в дерево вершины показаны полужирным шрифтом

    Рис. 9.2. Применение алгоритма Прима [↑](#footnote-ref-56)
57. Выбранные ребра показаны полужирным шрифтом

    Рис. 9.4. Применение алгоритма Крускала [↑](#footnote-ref-57)
58. Это — пример полезности амортизированной эффективности, о которой упоминалось в гла­ве 2. Временная эффективность любой последовательности из п операций union-by-size выше эффективности в наихудшем случае отдельной операции, выполненной п раз. [↑](#footnote-ref-58)
59. Эдсгер В. Дейкстра (Edsger W. Dijkstra) (1930-2002), знаменитый голландский ученый в области кибернетики, открыл этот алгоритм в средине 1950-х годов. Сам Дейкстра говорил об этом алгорит­ме: “Это была первая задача, посвященная графам, которую я самостоятельно поставил и решил. Как это ни удивительно, но я не опубликовал ее. В те времена алгоритмы рассматривались сугубо как научные работы.” [↑](#footnote-ref-59)
60. Более подробно деревья принятия решений рассматриваются в разделе 10.2. [↑](#footnote-ref-60)
61. Найдите тривиальные классы нижних границ для следующих задач и по возможности укажите, плотная ли эта граница.

    а) Поиск наибольшего элемента массива.

    б) Проверка полноты графа, представленного матрицей смежности.

    в) Генерация всех подмножеств n-элементного множества.

    г) Определение того, все ли п действительных чисел различны. [↑](#footnote-ref-61)
62. **]Это только один из множества вкладов в теооретическую кибернетику, сделанный английским ученым Аланом Тьюрингом (Alan Turing) (1912-1954). В ознаменование его заслуг ACM (Associa­tion for Computing Machinery — Ассоциация по вычислительной технике), общество, объединяющее профессионалов и исследователей в области вычислительной техники, учредило премию его име­ни, присуждаемую за выдающийся вклад в теоретическую кибернетику. В своем выступлении при вручении ему премии Ричард Карп (Richard Karp) [59] привел множество интересных исторических фактов о развитии теории сложности.** [↑](#footnote-ref-62)
63. Как часто случается в истории науки, переломные открытия совершаются независимо и прак­тически одновременно несколькими учеными. На самом деле Левин рассматривал более общее понятие, чем АГР-полнота, которое не ограничивалось задачами принятия решения, но его статья **[73]** была опубликована на два года позже работы Кука. [↑](#footnote-ref-63)
64. Решение систем линейных уравнений и вычисление полиномов, рассматривавшихся в разде­лах 6.2 и 6.5 соответственно, являются редкими исключениями из этого правила. [↑](#footnote-ref-64)
65. Используются также термины “ошибка метода”, “ошибка обрыва'. — Прим. перев. [↑](#footnote-ref-65)
66. **Джордж Форсайт (1917-1972), знаменитый ученый, игравший ведущую роль в становлении** [↑](#footnote-ref-66)
67. **информатики как отдельной академической дисциплины в США. Его слова использованы в качестве эпиграфа в предисловии к данной книге.** [↑](#footnote-ref-67)
68. **]Эта граница должна быть нижней границей для задачи минимизации и верхней — для задачи максимизации.** [↑](#footnote-ref-68)
69. **Понятие N**Р-сложной **задачи может быть более строго определено путем расширения понятия полиномиальной приводимости к задачам, которые не обязательно входят в класс NP, включая задачи оптимизации, рассматривающиеся в этом разделе (см. главу 5 в [40]).** [↑](#footnote-ref-69)
70. **Открытие Руффини было полностью проигнорировано всеми ведущими математиками того времени. Абель умер молодым, прожив трудную жизнь в нищете. Галуа был убит на дуэли, ко­гда ему был всего лишь 21 год. Их результаты по решению уравнений высших степеней сейчас рассматриваются как одни из высочайших достижений математики.** [↑](#footnote-ref-70)
71. В отечественной литературе встречаются и другие названия метода, такие как метод ложного положения, метод хорд. — Прим. ред. [↑](#footnote-ref-71)
72. Количество подмножеств n-элементного множества: 2П [↑](#footnote-ref-72)
73. Строго говоря, корректность общей формулы необходимо доказать методом математической ин­дукции по г. Однако зачастую проще сначала получить решение, а потом проверить его (например, как мы делали это ранее для х (п) = п (п + 1)/2). [↑](#footnote-ref-73)
74. **В используемой нами терминологии при а = 1 соотношение описывает алгоритмы уменьшения** [↑](#footnote-ref-74)
75. **на постоянный множитель, а не декомпозиции.** [↑](#footnote-ref-75)
76. **Можно перенумеровать столбцы и строки — это даст ту же степень эффективности решения поставленной задачи. — Прим. ред.** [↑](#footnote-ref-76)
77. **Подумайте также о возможном использовании двух стеков. — Прим. ред.** [↑](#footnote-ref-77)
78. **Такой информационно-теоретический подход был предложен Брассардом (Brassard) и Брейтли (Bratley) [22].** [↑](#footnote-ref-78)