

Edition

Second

Introduction to the Design & Analysis of
Algorithms



Anany Levitin

This edition is manufactured in India and is authorized for sale only in
India, Bangladesh, Bhutan, Pakistan, Nepal, Sri Lanka and the Maldives.
Circulation of this edition outside of these territories is UNAUTHORIZED.

Copyrighted material

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Authorized adaptation from the United States edition, entitled *Introduction to The Design & Analysis of Algorithms, 2nd Edition*, ISBN: 9780321358288 by Levitin, Anany; published by Pearson Education, Inc., Copyright © 2007

Indian Subcontinent Adaptation

Copyright © 2008 Dorling Kindersley (India) Pvt. Ltd

This book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, resold, hired out or otherwise circulated without the publisher's prior written consent in any form of binding or cover other than that in which it is published without a similar condition including this condition being imposed on subsequent purchaser and without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of both the copyright owner and above-mentioned publisher of this book.

ISBN 978-81-317-1837-7

First Impression

Second Impression, 2009

This edition is manufactured in India and is authorized for sale only in India, Bangladesh, Bhutan, Pakistan, Nepal, Sri Lanka and the Maldives.

Published by Dorling Kindersley (India) Pvt. Ltd., licensees of Pearson Education in South Asia.

Head Office: 482 F.I.E., Patparganj, Delhi 110 092, India.

Registered Office: 14 Local Shopping Centre, Panchsheel Park, New Delhi 110 017, India.

Printed in India at Saurabh Printers Pvt. Ltd.

Brief Contents

<u>Preface</u>	xv
<u>1 Introduction</u>	1
<u>2 Fundamentals of the Analysis of Algorithm Efficiency</u>	39
<u>3 Brute Force</u>	93
<u>4 Divide-and-Conquer</u>	117
<u>5 Decrease-and-Conquer</u>	149
<u>6 Transform-and-Conquer</u>	187
<u>7 Space and Time Tradeoffs</u>	237
<u>8 Dynamic Programming</u>	265
<u>9 Greedy Technique</u>	291
<u>10 Iterative Improvement</u>	317
<u>11 Limitations of Algorithm Power</u>	359
<u>12 Coping with the Limitations of Algorithm Power</u>	393
<u>Epilogue</u>	441
 APPENDIX A <u>Useful Formulas for the Analysis of Algorithms</u>	445
 APPENDIX B <u>Short Tutorial on Recurrence Relations</u>	449
<u>Bibliography</u>	463
<u>Hints to Selected Exercises</u>	473
<u>Index</u>	505

Contents

Preface

xv

1	Introduction	1
1.1	What is an Algorithm?	3
	Exercises 1.1	7
1.2	Fundamentals of Algorithmic Problem Solving	9
	<u>Understanding the Problem</u>	<u>9</u>
	<u>Ascertaining the Capabilities of a Computational Device</u>	<u>10</u>
	<u>Choosing between Exact and Approximate Problem Solving</u>	<u>11</u>
	<u>Deciding on Appropriate Data Structures</u>	<u>11</u>
	<u>Algorithm Design Techniques</u>	<u>11</u>
	<u>Methods of Specifying an Algorithm</u>	<u>12</u>
	<u>Proving an Algorithm's Correctness</u>	<u>13</u>
	<u>Analyzing an Algorithm</u>	<u>13</u>
	<u>Coding an Algorithm</u>	<u>14</u>
	Exercises 1.2	16
1.3	Important Problem Types	17
	<u>Sorting</u>	<u>18</u>
	<u>Searching</u>	<u>19</u>
	<u>String Processing</u>	<u>19</u>
	<u>Graph Problems</u>	<u>20</u>
	<u>Combinatorial Problems</u>	<u>20</u>
	<u>Geometric Problems</u>	<u>21</u>
	<u>Numerical Problems</u>	<u>21</u>
	Exercises 1.3	22

v

1.4 Fundamental Data Structures	24
<u>Linear Data Structures</u>	24
<u>Graphs</u>	26
<u>Trees</u>	30
<u>Sets and Dictionaries</u>	33
Exercises 1.4	35
Summary	36
2 Fundamentals of the Analysis of Algorithm Efficiency	39
2.1 Analysis Framework	40
<u>Measuring an Input's Size</u>	41
<u>Units for Measuring Running Time</u>	42
<u>Orders of Growth</u>	43
<u>Worst-Case, Best-Case, and Average-Case Efficiencies</u>	44
<u>Recapitulation of the Analysis Framework</u>	47
Exercises 2.1	48
2.2 Asymptotic Notations and Basic Efficiency Classes	49
<u>Informal Introduction</u>	49
<u>O-notation</u>	50
<u>Ω-notation</u>	51
<u>Θ-notation</u>	51
<u>Useful Property Involving the Asymptotic Notations</u>	52
<u>Using Limits for Comparing Orders of Growth</u>	53
<u>Basic Efficiency Classes</u>	55
Exercises 2.2	56
2.3 Mathematical Analysis of Nonrecursive Algorithms	57
Exercises 2.3	63
2.4 Mathematical Analysis of Recursive Algorithms	65
Exercises 2.4	72
2.5 Example: Fibonacci Numbers	74
<u>Explicit Formula for the nth Fibonacci Number</u>	75
<u>Algorithms for Computing Fibonacci Numbers</u>	76
Exercises 2.5	79
2.6 Empirical Analysis of Algorithms	80
Exercises 2.6	85

<u>2.7 Algorithm Visualization</u>	<u>86</u>
<u> Summary</u>	<u>90</u>
 3 Brute Force	93
<u>3.1 Selection Sort and Bubble Sort</u>	<u>94</u>
<u>Selection Sort</u>	<u>94</u>
<u>Bubble Sort</u>	<u>96</u>
<u>Exercises 3.1</u>	<u>97</u>
<u>3.2 Sequential Search and Brute-Force String Matching</u>	<u>98</u>
<u>Sequential Search</u>	<u>99</u>
<u>Brute-Force String Matching</u>	<u>99</u>
<u>Exercises 3.2</u>	<u>101</u>
<u>3.3 Closest-Pair and Convex-Hull Problems by Brute Force</u>	<u>102</u>
<u>Closest-Pair Problem</u>	<u>102</u>
<u>Convex-Hull Problem</u>	<u>103</u>
<u>Exercises 3.3</u>	<u>107</u>
<u>3.4 Exhaustive Search</u>	<u>108</u>
<u>Traveling Salesman Problem</u>	<u>109</u>
<u>Knapsack Problem</u>	<u>110</u>
<u>Assignment Problem</u>	<u>110</u>
<u>Exercises 3.4</u>	<u>113</u>
<u> Summary</u>	<u>115</u>
 4 Divide-and-Conquer	117
<u>4.1 Mergesort</u>	<u>119</u>
<u>Exercises 4.1</u>	<u>121</u>
<u>4.2 Quicksort</u>	<u>123</u>
<u>Exercises 4.2</u>	<u>127</u>
<u>4.3 Binary Search</u>	<u>128</u>
<u>Exercises 4.3</u>	<u>131</u>
<u>4.4 Binary Tree Traversals and Related Properties</u>	<u>132</u>
<u>Exercises 4.4</u>	<u>135</u>

4.5 Multiplication of Large Integers and Strassen's Matrix Multiplication	137
<u>Multiplication of Large Integers</u>	137
<u>Strassen's Matrix Multiplication</u>	139
<u>Exercises 4.5</u>	141
4.6 Closest-Pair and Convex-Hull Problems by Divide-and-Conquer	142
<u>Closest-Pair Problem</u>	142
<u>Convex-Hull Problem</u>	144
<u>Exercises 4.6</u>	146
<u>Summary</u>	147
5 Decrease-and-Conquer	149
5.1 Insertion Sort	152
<u>Exercises 5.1</u>	154
5.2 Depth-First Search and Breadth-First Search	156
<u>Depth-First Search</u>	156
<u>Breadth-First Search</u>	159
<u>Exercises 5.2</u>	161
5.3 Topological Sorting	163
<u>Exercises 5.3</u>	166
5.4 Algorithms for Generating Combinatorial Objects	167
<u>Generating Permutations</u>	168
<u>Generating Subsets</u>	170
<u>Exercises 5.4</u>	171
5.5 Decrease-by-a-Constant-Factor Algorithms	172
<u>Eake-Coin Problem</u>	172
<u>Multiplication à la Russe</u>	173
<u>Josephus Problem</u>	174
<u>Exercises 5.5</u>	176
5.6 Variable-Size-Decrease Algorithms	177
<u>Computing a Median and the Selection Problem</u>	177
<u>Interpolation Search</u>	178
<u>Searching and Insertion in a Binary Search Tree</u>	180

The Game of Nim	181
Exercises 5.6	183
Summary	184
6 Transform-and-Conquer	187
6.1 Presorting	188
Exercises 6.1	191
6.2 Gaussian Elimination	193
LU Decomposition and Other Applications	197
Computing a Matrix Inverse	199
Computing a Determinant	200
Exercises 6.2	201
6.3 Balanced Search Trees	202
AVL Trees	203
2-3 Trees	208
Exercises 6.3	210
6.4 Heaps and Heapsort	211
Notion of the Heap	211
Heapsort	215
Exercises 6.4	217
6.5 Horner's Rule and Binary Exponentiation	218
Horner's Rule	218
Binary Exponentiation	220
Exercises 6.5	223
6.6 Problem Reduction	224
Computing the Least Common Multiple	225
Counting Paths in a Graph	226
Reduction of Optimization Problems	226
Linear Programming	228
Reduction to Graph Problems	230
Exercises 6.6	231
Summary	233
7 Space and Time Tradeoffs	237
7.1 Sorting by Counting	238
Exercises 7.1	241

7.2 Input Enhancement in String Matching	242
Horspool's Algorithm	243
Boyer-Moore Algorithm	246
Exercises 7.2	251
7.3 Hashing	252
Open Hashing (Separate Chaining)	253
Closed Hashing (Open Addressing)	255
Exercises 7.3	257
7.4 B-Trees	258
Exercises 7.4	262
Summary	263
8 Dynamic Programming	265
8.1 Computing a Binomial Coefficient	266
Exercises 8.1	268
8.2 Warshall's and Floyd's Algorithms	270
Warshall's Algorithm	270
Floyd's Algorithm for the All-Pairs Shortest-Paths Problem	273
Exercises 8.2	277
8.3 Optimal Binary Search Trees	278
Exercises 8.3	282
8.4 The Knapsack Problem and Memory Functions	284
Memory Functions	286
Exercises 8.4	287
Summary	288
9 Greedy Technique	291
9.1 Prim's Algorithm	292
Exercises 9.1	297
9.2 Kruskal's Algorithm	298
Disjoint Subsets and Union-Find Algorithms	301
Exercises 9.2	304

9.3 Dijkstra's Algorithm	305
Exercises 9.3	309
9.4 Huffman Trees	311
Exercises 9.4	314
Summary	316
10 Iterative Improvement	317
10.1 The Simplex Method	318
Geometric Interpretation of Linear Programming	319
An Outline of the Simplex Method	323
Further Notes on the Simplex Method	329
Exercises 10.1	331
10.2 The Maximum-Flow Problem	333
Exercises 10.2	343
10.3 Maximum Matching in Bipartite Graphs	345
Exercises 10.3	351
10.4 The Stable Marriage Problem	353
Exercises 10.4	356
Summary	357
11 Limitations of Algorithm Power	359
11.1 Lower-Bound Arguments	360
Trivial Lower Bounds	361
Information-Theoretic Arguments	362
Adversary Arguments	362
Problem Reduction	363
Exercises 11.1	365
11.2 Decision Trees	366
Decision Trees for Sorting Algorithms	367
Decision Trees for Searching a Sorted Array	369
Exercises 11.2	371
11.3 P , NP , and NP -complete Problems	372
P and NP Problems	373

<u>NP-Complete Problems</u>	377
<u>Exercises 11.3</u>	380
11.4 Challenges of Numerical Algorithms	382
<u>Exercises 11.4</u>	390
<u>Summary</u>	391
12 Coping with the Limitations of Algorithm Power	393
12.1 Backtracking	394
<u>n-Queens Problem</u>	395
<u>Hamiltonian Circuit Problem</u>	396
<u>Subset-Sum Problem</u>	397
General Remarks	398
<u>Exercises 12.1</u>	400
12.2 Branch-and-Bound	402
<u>Assignment Problem</u>	403
<u>Knapsack Problem</u>	405
<u>Traveling Salesman Problem</u>	408
<u>Exercises 12.2</u>	410
12.3 Approximation Algorithms for NP-hard Problems	411
<u>Approximation Algorithms for the Traveling Salesman Problem</u>	413
<u>Approximation Algorithms for the Knapsack Problem</u>	423
<u>Exercises 12.3</u>	427
12.4 Algorithms for Solving Nonlinear Equations	429
<u>Bisection Method</u>	430
<u>Method of False Position</u>	433
Newton's Method	434
<u>Exercises 12.4</u>	436
<u>Summary</u>	438
Epilogue	441

APPENDIX A

Useful Formulas for the Analysis of Algorithms	445
<u>Properties of Logarithms</u>	445
<u>Combinatorics</u>	445
<u>Important Summation Formulas</u>	446

<u>Sum Manipulation Rules</u>	<u>446</u>
Approximation of a Sum by a Definite Integral	447
Floor and Ceiling Formulas	447
Miscellaneous	447
 <u>APPENDIX B</u>	
<u>Short Tutorial on Recurrence Relations</u>	<u>449</u>
Sequences and Recurrence Relations	449
Methods for Solving Recurrence Relations	450
Common Recurrence Types in Algorithm Analysis	455
<u>Bibliography</u>	<u>463</u>
<u>Hints to Selected Exercises</u>	<u>473</u>
<u>Index</u>	<u>505</u>

Preface

The most valuable acquisitions in a scientific or technical education are the general-purpose mental tools which remain serviceable for a life-time.

—George Forsythe, "What to do till the computer scientist comes," 1968

Algorithms play the central role in both the science and practice of computing. Recognition of this fact has led to the appearance of a considerable number of textbooks on the subject. By and large, they follow one of two alternatives in presenting algorithms. One classifies algorithms according to a problem type. Such a book would have separate chapters on algorithms for sorting, searching, graphs, and so on. The advantage of this approach is that it allows an immediate comparison of, say, the efficiency of different algorithms for the same problem. The drawback of this approach is that it emphasizes problem types at the expense of algorithm design techniques.

The second alternative organizes the presentation around algorithm design techniques. In this organization, algorithms from different areas of computing are grouped together if they have the same design approach. I share the belief of many (e.g., [BaY95]) that this organization is more appropriate for a basic course on the design and analysis of algorithms. There are three principal reasons for emphasis on algorithm design techniques. First, these techniques provide a student with tools for designing algorithms for new problems. This makes learning algorithm design techniques a very valuable endeavor from the practical standpoint. Second, they seek to classify multitudes of known algorithms according to an underlying design idea. Learning to see such commonality among algorithms from different application areas should be a major goal of computer science education. After all, every science considers classification of its principal subject as a major if not the central point of its discipline. Third, in my opinion, algorithm design techniques have considerable utility as general problem solving strategies, applicable to problems beyond computing.

Unfortunately, the traditional classification of algorithm design techniques has several serious shortcomings from both theoretical and educational points of view. The most significant of these shortcomings is the failure to classify many important algorithms. This limitation has forced the authors of textbooks to depart

from the design technique organization and to include chapters dealing with specific problem types. Such a switch leads to a loss of course coherence and almost unavoidably creates confusion in students' minds.

New Taxonomy of Algorithm Design Techniques

My frustration with the shortcomings of the traditional classification of algorithm design techniques has motivated me to develop a new taxonomy of them [Lev99], which is the basis of this book. Here are the principal advantages of the new taxonomy:

- The new taxonomy is more comprehensive than the traditional one. It includes several strategies—brute force, decrease-and-conquer, transform-and-conquer, space and time tradeoffs, and iterative improvement—that are rarely if ever recognized as important design paradigms.
- The new taxonomy covers naturally many classic algorithms (Euclid's algorithm, heapsort, search trees, hashing, topological sorting, Gaussian elimination, Horner's rule, to name a few) that the traditional taxonomy cannot classify. As a result, the new taxonomy makes it possible to present the standard body of classic algorithms in a unified and coherent fashion.
- It naturally accommodates the existence of important varieties of several design techniques. (For example, it recognizes three variations of decrease-and-conquer and three variations of transform-and-conquer.)
- It is better aligned with analytical methods for efficiency analysis (see Appendix B).

Design Techniques as General Problem Solving Strategies

Most applications of the design techniques in the book are to classic problems of computer science. (The only innovation here is the inclusion of some material on numerical algorithms, which are covered within the same general framework.) But these design techniques can be considered general problem solving tools, whose applications are not limited to traditional computing and mathematical problems. Two factors make this point particularly important. First, more and more computing applications go beyond the traditional domain, and there are reasons to believe that this trend will strengthen in the future. Second, developing students' problem solving skills has come to be recognized as a major goal of college education. Among all the courses in a computer science curriculum, a course on the design and analysis of algorithms is uniquely suitable for this task because it can offer a student specific strategies for solving problems.

I am not proposing that a course on the design and analysis of algorithms should become a course on general problem solving. But I do believe that the unique opportunity provided by studying the design and analysis of algorithms should not be missed. Toward this goal, the book includes applications to puzzles and puzzle-like games. Although using puzzles in teaching algorithms is certainly

not a new idea, the book tries to do so systematically by going well beyond a few standard examples.

Textbook Pedagogy

My goal was to write a text that would not trivialize the subject but would still be readable by most students on their own. Here are some of the things done toward this objective.

- Sharing the opinion of George Forsythe expressed in the epigraph, I have sought to stress major ideas underlying the design and analysis of algorithms. In choosing specific algorithms to illustrate these ideas, I limited the number of covered algorithms to those that most clearly demonstrate an underlying design technique or analysis method. Fortunately, most classic algorithms satisfy this criterion.
- In Chapter 2, which is devoted to efficiency analysis, the methods used for analyzing nonrecursive algorithms are separated from those typically used for analyzing recursive algorithms. The chapter also includes sections devoted to empirical analysis and algorithm visualization.
- The narrative is systematically interrupted by questions to the reader. Some of them are asked rhetorically, in anticipation of a concern or doubt, and are answered immediately. The goal of the others is to prevent the reader from drifting through the text without a satisfactory level of comprehension.
- Each chapter ends with a summary recapping the most important concepts and results discussed in the chapter.
- The book contains about 700 exercises. Some of them are drills; others make important points about the material covered in the body of the text or introduce algorithms not covered there at all. A few exercises take advantage of Internet resources. More difficult problems—there are not many of them—are marked with a special symbol in the Instructor's Manual. (Because designating problems as difficult may discourage some students from trying to tackle them, problems are not marked in the book itself.) Puzzles, games, and puzzle-like questions are marked in the exercises with a special icon.
- The book provides hints to selected exercises. Detailed solutions, except for programming projects, are provided in the Instructor's Manual available at www.pearsoned.co.in/ananylevitin. In addition, slides in PowerPoint are available here to all readers of this book.



Changes for the Second Edition

The most important change in the second edition is a new chapter (Chapter 10) dealing with the iterative-improvement technique. There are several reasons for this addition.

- Some of the most important algorithms in computer science, both from a theoretical and practical standpoint, are based on this technique.

- Iterative improvement is the principal alternative to the greedy strategy already covered in the book.
- Since linear programming is discussed in Chapter 6, it is natural to cover its principal algorithm (the simplex method) as well.
- Iterative improvement is the only design technique of the new taxonomy that was not included in the first edition. Therefore, its inclusion could also be justified by the sheer need for completeness.

The four sections of the new chapter cover the following topics: the simplex method, maximum network flows, maximum cardinality matching in bipartite graphs, and the stable marriage problem. Each of these sections can be studied independently of the rest, as is the case for the other chapters of the book.

There are two other additions in this edition worth mentioning. The first is a new subsection about the game of Nim, included in Section 5.6 on variable-decrease algorithms. The second is an expanded coverage of approximation algorithms for the traveling salesman problem to give a better account of the remarkable progress achieved in this area.

I have also added about 80 new problems to the exercises. About half of them are added to the old chapters, and a majority are puzzles or puzzle-like questions. Judging by the feedback received, the inclusion of puzzles and games in the exercises has been welcome by the book's readers, including instructors and students using it as a textbook. As in the first edition, only the puzzles related to the discussed topics were included in the exercises.

Prerequisites

The book assumes that a reader has gone through an introductory programming course and a standard course on discrete structures. With such a background, he or she should be able to handle the book's material without undue difficulty. Still, fundamental data structures, necessary summation formulas, and recurrence relations are reviewed in Section 1.4, Appendix A, and Appendix B, respectively. Calculus is used in only three sections (Sections 2.2, 11.4, and 12.4) and to a very limited degree; if students lack calculus as an assured part of their background, the portions of these three sections that involve calculus can be omitted without hindering their understanding of the rest of the material.

Use in the Curriculum

The book can serve as a textbook for a basic course on design and analysis of algorithms organized around algorithm design techniques. It might contain too much material for a typical one-semester course. By and large, portions of Chapters 3 through 12 can be skipped without the danger of making later parts of the book incomprehensible to the reader. Any portion of the book can be assigned for self-study. In particular, Sections 2.6 and 2.7 on empirical analysis and algorithm visualization, respectively, can be assigned in conjunction with projects.

Here is a possible plan for a one-semester course; it assumes a 40-class meeting format.

Lecture	Topic	Sections
1	Introduction	1.1–1.3
2, 3	Analysis framework; O , Θ , Ω notations	2.1, 2.2
4	Mathematical analysis of nonrecursive algorithms	2.3
5, 6	Mathematical analysis of recursive algorithms	2.4, 2.5 (+ App. B)
7	Brute-force algorithms	3.1, 3.2 (+ 3.3)
8	Exhaustive search	3.4
9–10	Divide-and-conquer: mergesort, quicksort, binary search	4.1–4.3
11	Other divide-and-conquer examples	4.4 or 4.5 or 4.6
12–14	Decrease-by-one: insertion sort, DFS and BFS, topological sorting	5.1–5.3
15	Decrease-by-a-constant-factor algorithms	5.5
16	Variable-size-decrease algorithms	5.6
17–19	Instance simplification: presorting, Gaussian elimination, balanced search trees	6.1–6.3
20	Representation change: heaps and heapsort or Horner's rule and binary exponentiation	6.4 or 6.5
21	Problem reduction	6.6
22–24	Space-time tradeoffs: string matching, hashing, B-trees	7.2–7.4
25–27	Dynamic programming algorithms	3 from 8.1–8.4
28–30	Greedy algorithms: Prim's, Kruskal's, Dijkstra's, Huffman's	9.1–9.4
31–33	Iterative improvement algorithms	3 from 10.1–10.4
34	Lower-bound arguments	11.1
35	Decision trees	11.2
36	P , NP , and NP -complete problems	11.3
37	Numerical algorithms	11.4 (+ 12.4)
38	Backtracking	12.1
39	Branch-and-bound	12.2
40	Approximation algorithms for NP -hard problems	12.3

Acknowledgments

I would like to express my gratitude to the many people who have shared with me their own and their students' opinions about the first edition and suggested improvements and corrections. I have especially benefitted from the comments by Eric Bach (University of Wisconsin), Stephen Bique (University of Alaska-Fairbanks), Walter Hower (Albstadt-Sigmaringen University), Dan Joyce (Villanova University), Steven Lindell (Haverford College), and Mary-Angela Papalaskari (Villanova University).

My thanks also go to Edward L. Bosworth (Columbus State University), Philip Dorin (Loyola Marymount University), William Ford (University of the Pacific), Jonathan Goldstine (Penn State University), Chin-Cheng Hung (Southern Polytechnic State University), and Brian Rossmaita (Hamilton College)—the reviewers of the chapter on iterative-improvement algorithms—for supporting the inclusion of this chapter in the new edition and for their ideas about presenting these important algorithms.

In addition, I would like to convey my appreciation to the following reviewers: David Avis (McGill University), Gordon Beavers (University of Arkansas), Dinesh Mehta (Colorado School of Mines), Feridoon Moinian (Cameron University), Stephan Olariu (Old Dominion University), Stanley Selkow (WPI), Jerry Trahan (Louisiana State University), and Yang Wang (Missouri State University).

I gratefully acknowledge the help I have received from my former student, Rakhi Gandhi, whose assistance has certainly decreased the number of typos and errors in the book.

I appreciate the contributions of all the reviewers of the first edition: Simon Berkovich (George Washington University), Richard Borie (University of Alabama), Douglas M. Campbell (Brigham Young University), Bin Cong (California State University, Fullerton), Steve Homer (Boston University), Roland Hübscher (Auburn University), Sukhamay Kundu (Louisiana State University), Sheau-Dong Lang (University of Central Florida), John C. Lusth (University of Arkansas), John F. Meyer (University of Michigan), Steven R. Seidel (Michigan Technological University), Ali Shokoufandeh (Drexel University), and George H. Williams (Union College).

My thanks go to all the people at Addison-Wesley and their associates who worked on my book. I am especially grateful to my editor Matt Goldstein, Michelle Brown, Katherine Harutunian, Marilyn Lloyd, Joyce Wells, and Paul Anagnostopoulos of Windfall Software. I was lucky to have Richard Camp as the copy editor; his numerous suggestions have definitely improved the book.

Finally, I am indebted to two members of my family. Living with a spouse writing a book is probably more trying than doing the actual writing. My wife, Maria, lived through several years of this, helping me any way she could. And help she did: over 400 figures in the book and the Instructor's Manual were created by her. My daughter Miriam has been my English prose guru over many years. She not only read large portions of the book, but she was also instrumental in finding the chapter epigraphs.

Anany Levitin
anany.levitin@villanova.edu
December 2005

The publishers would like to thank C. V. Guru Rao, Professor and Principal, Kakatiya Institute of Technology and Science, Warangal, for his valuable suggestions and inputs in enhancing the content of this book to suit the requirements of Indian universities.

Introduction

Two ideas lie gleaming on the jeweler's velvet. The first is the calculus, the second, the algorithm. The calculus and the rich body of mathematical analysis to which it gave rise made modern science possible; but it has been the algorithm that has made possible the modern world.

—David Berlinski, *The Advent of the Algorithm*, 2000

Why do you need to study algorithms? If you are going to be a computer professional, there are both practical and theoretical reasons to study algorithms. From a practical standpoint, you have to know a standard set of important algorithms from different areas of computing; in addition, you should be able to design new algorithms and analyze their efficiency. From the theoretical standpoint, the study of algorithms, sometimes called *algorithmics*, has come to be recognized as the cornerstone of computer science. David Harel, in his delightful book pointedly titled *Algorithmics: the Spirit of Computing*, put it as follows:

Algorithmics is more than a branch of computer science. It is the core of computer science, and, in all fairness, can be said to be relevant to most of science, business, and technology. [Har92], p. 6.

But even if you are not a student in a computing-related program, there are compelling reasons to study algorithms. To put it bluntly, computer programs would not exist without algorithms. And with computer applications becoming indispensable in almost all aspects of our professional and personal lives, studying algorithms becomes a necessity for more and more people.

Another reason for studying algorithms is their usefulness in developing analytical skills. After all, algorithms can be seen as special kinds of solutions to problems—not answers but precisely defined procedures for getting answers. Consequently, specific algorithm design techniques can be interpreted as problem-solving strategies that can be useful regardless of whether a computer is involved. Of course, the precision inherently imposed by algorithmic thinking limits the kinds of problems that can be solved with an algorithm. You will not find,

for example, an algorithm for living a happy life or becoming rich and famous. On the other hand, this required precision has an important educational advantage. Donald Knuth, one of the most prominent computer scientists in the history of algorithmics, put it as follows:

A person well-trained in computer science knows how to deal with algorithms: how to construct them, manipulate them, understand them, analyze them. This knowledge is preparation for much more than writing good computer programs; it is a general-purpose mental tool that will be a definite aid to the understanding of other subjects, whether they be chemistry, linguistics, or music, etc. The reason for this may be understood in the following way: It has often been said that a person does not really understand something until after teaching it to someone else. Actually, a person does not *really* understand something until after teaching it to a *computer*, i.e., expressing it as an algorithm...An attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way. [Knu96], p. 9.

We take up the notion of algorithm in Section 1.1. As examples, we use three algorithms for the same problem: computing the greatest common divisor. There are several reasons for this choice. First, it deals with a problem familiar to everybody from their middle-school days. Second, it makes the important point that the same problem can often be solved by several algorithms. Quite typically, these algorithms differ in their idea, level of sophistication, and efficiency. Third, one of these algorithms deserves to be introduced first, both because of its age—it appeared in Euclid's famous treatise more than two thousand years ago—and its enduring power and importance. Finally, the middle-school procedure for computing the greatest common divisor allows us to highlight a critical requirement every algorithm must satisfy.

Section 1.2 deals with algorithmic problem solving. There we discuss several important issues related to the design and analysis of algorithms. The different aspects of algorithmic problem solving range from analysis of the problem and the means of expressing an algorithm to establishing its correctness and analyzing its efficiency. The section does not contain a magic recipe for designing an algorithm for an arbitrary problem. It is a well-established fact that such a recipe does not exist. Still, the material of Section 1.2 should be useful for organizing your work on designing and analyzing algorithms.

Section 1.3 is devoted to a few problem types that have proven to be particularly important to the study of algorithms and their application. In fact, there are textbooks (e.g., [Sed88]) organized around such problem types. I hold the view—shared by many others—that an organization based on algorithm design techniques is superior. In any case, it is very important to be aware of the principal problem types. Not only are they the most commonly encountered problem types in real-life applications, they are used throughout the book to demonstrate particular algorithm design techniques.

Section 1.4 contains a review of fundamental data structures. It is meant to serve as a reference rather than a deliberate discussion of this topic. If you need

a more detailed exposition, there is a wealth of good books on the subject, most of them tailored to a particular programming language.

1.1 What is an Algorithm?

Although there is no universally agreed-on wording to describe this notion, there is general agreement about what the concept means:

An **algorithm** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

This definition can be illustrated by a simple diagram (Figure 1.1).

The reference to “instructions” in the definition implies that there is something or someone capable of understanding and following the instructions given. We call this a “computer,” keeping in mind that before the electronic computer was invented, the word “computer” meant a human being involved in performing numeric calculations. Nowadays, of course, “computers” are those ubiquitous electronic devices that have become indispensable in almost everything we do. Note, however, that although the majority of algorithms are indeed intended for eventual computer implementation, the notion of algorithm does not depend on such an assumption.

As examples illustrating the notion of algorithm, we consider in this section three methods for solving the same problem: computing the greatest common divisor of two integers. These examples will help us to illustrate several important points:

- The nonambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- Several algorithms for solving the same problem may exist.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

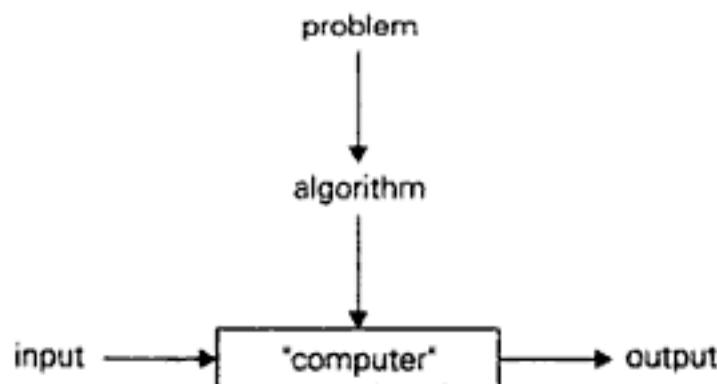


FIGURE 1.1 Notion of algorithm

Recall that the greatest common divisor of two nonnegative, not-both-zero integers m and n , denoted $\gcd(m, n)$, is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero. Euclid of Alexandria (third century B.C.) outlined an algorithm for solving this problem in one of the volumes of his *Elements*, most famous for its systematic exposition of geometry. In modern terms, **Euclid's algorithm** is based on applying repeatedly the equality

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

(where $m \bmod n$ is the remainder of the division of m by n) until $m \bmod n$ is equal to 0; since $\gcd(m, 0) = m$ (why?), the last value of m is also the greatest common divisor of the initial m and n .

For example, $\gcd(60, 24)$ can be computed as follows:

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

(If you are not impressed by this algorithm, try finding the greatest common divisor of larger numbers such as those in Problem 5 of Exercises 1.1.)

Here is a more structured description of this algorithm:

Euclid's algorithm for computing $\gcd(m, n)$

- Step 1** If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.
- Step 2** Divide m by n and assign the value of the remainder to r .
- Step 3** Assign the value of n to m and the value of r to n . Go to Step 1.

Alternatively, we can express the same algorithm in a pseudocode:

ALGORITHM Euclid(m, n)

```
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
return m
```

How do we know that Euclid's algorithm eventually comes to a stop? This follows from the observation that the second number of the pair gets smaller with each iteration and it cannot become negative. Indeed, the new value of n on the next iteration is $m \bmod n$, which is always smaller than n . Hence, the value of the second number in the pair eventually becomes 0, and the algorithm stops.

Just as with many other problems, there are several algorithms for computing the greatest common divisor. Let us look at the other two methods for this

problem. The first is simply based on the definition of the greatest common divisor of m and n as the largest integer that divides both numbers evenly. Obviously, such a common divisor cannot be greater than the smaller of these numbers, which we will denote by $t = \min\{m, n\}$. So we can start by checking whether t divides both m and n : if it does, t is the answer; if it does not, we simply decrease t by 1 and try again. (How do we know that the process will eventually stop?) For example, for numbers 60 and 24, the algorithm will try first 24, then 23, and so on until it reaches 12, where it stops.

Consecutive integer checking algorithm for computing $\gcd(m, n)$

- Step 1** Assign the value of $\min\{m, n\}$ to t .
- Step 2** Divide m by t . If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.
- Step 3** Divide n by t . If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.
- Step 4** Decrease the value of t by 1. Go to Step 2.

Note that unlike Euclid's algorithm, this algorithm, in the form presented, does not work correctly when one of its input numbers is zero. This example illustrates why it is so important to specify the range of an algorithm's inputs explicitly and carefully.

The third procedure for finding the greatest common divisor should be familiar to you from middle school.

Middle-school procedure for computing $\gcd(m, n)$

- Step 1** Find the prime factors of m .
- Step 2** Find the prime factors of n .
- Step 3** Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If p is a common factor occurring p_m and p_n times in m and n , respectively, it should be repeated $\min\{p_m, p_n\}$ times.)
- Step 4** Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Thus, for the numbers 60 and 24, we get

$$\begin{aligned}60 &= 2 \cdot 2 \cdot 3 \cdot 5 \\24 &= 2 \cdot 2 \cdot 2 \cdot 3 \\\gcd(60, 24) &= 2 \cdot 2 \cdot 3 = 12.\end{aligned}$$

Nostalgia for the days when we learned this method should not prevent us from noting that the last procedure is much more complex and slower than Euclid's algorithm. (We will discuss methods for finding and comparing running times of algorithms in the next chapter.) In addition to inferior efficiency, the middle-school procedure does not qualify, in the form presented, as a legitimate algorithm. Why?

Because the prime factorization steps are not defined unambiguously: they require a list of prime numbers, and I strongly suspect that your middle-school teacher did not explain how to obtain such a list. You undoubtedly agree that this is not a matter of unnecessary nitpicking. Unless this issue is resolved, we cannot, say, write a program implementing this procedure. (Incidentally, Step 3 is also not defined clearly enough. Its ambiguity is much easier to rectify than that of the factorization steps, however. How would you find common elements in two sorted lists?)

So let us introduce a simple algorithm for generating consecutive primes not exceeding any given integer n . It was probably invented in ancient Greece and is known as the *sieve of Eratosthenes* (ca. 200 b.c.). The algorithm starts by initializing a list of prime candidates with consecutive integers from 2 to n . Then, on the first iteration of the algorithm, it eliminates from the list all multiples of 2, i.e., 4, 6, and so on. Then it moves to the next item on the list, which is 3, and eliminates its multiples. (In this straightforward version, there is an overhead because some numbers, such as 6, are eliminated more than once.) No pass for number 4 is needed: since 4 itself and all its multiples are also multiples of 2, they were already eliminated on a previous pass. (By similar reasoning, we need not consider multiples of any eliminated number.) The next remaining number on the list, which is used on the third pass, is 5. The algorithm continues in this fashion until no more numbers can be eliminated from the list. The remaining integers of the list are the primes needed.

As an example, consider the application of the algorithm to finding the list of primes not exceeding $n = 25$:

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			25

For this example, no more passes are needed because they would eliminate numbers already eliminated on previous iterations of the algorithm. The remaining numbers on the list are the consecutive primes less than or equal to 25.

In general, what is the largest number p whose multiples can still remain on the list? Before we answer this question, let us first note that if p is a number whose multiples are being eliminated on the current pass, then the first multiple we should consider is $p \cdot p$ because all its smaller multiples $2p, \dots, (p-1)p$ have been eliminated on earlier passes through the list. This observation helps to avoid eliminating the same number more than once. Obviously, $p \cdot p$ should not be greater than n , and therefore p cannot exceed \sqrt{n} rounded down (denoted $\lfloor \sqrt{n} \rfloor$ using the so-called floor function). We assume in the following pseudocode that there is a function available for computing $\lfloor \sqrt{n} \rfloor$; alternatively, we could check the inequality $p \cdot p \leq n$ as the loop continuation condition there.

ALGORITHM Sieve(n)

```

//Implements the sieve of Eratosthenes
//Input: An integer  $n \geq 2$ 
//Output: Array  $L$  of all prime numbers less than or equal to  $n$ 
for  $p \leftarrow 2$  to  $n$  do  $A[p] \leftarrow p$ 
for  $p \leftarrow 2$  to  $\lfloor \sqrt{n} \rfloor$  do //see note before pseudocode
    if  $A[p] \neq 0$  // $p$  hasn't been eliminated on previous passes
         $j \leftarrow p * p$ 
        while  $j \leq n$  do
             $A[j] \leftarrow 0$  //mark element as eliminated
             $j \leftarrow j + p$ 
//copy the remaining elements of  $A$  to array  $L$  of the primes
 $i \leftarrow 0$ 
for  $p \leftarrow 2$  to  $n$  do
    if  $A[p] \neq 0$ 
         $L[i] \leftarrow A[p]$ 
         $i \leftarrow i + 1$ 
return  $L$ 

```

So now we can incorporate the sieve of Eratosthenes into the middle-school procedure to get a legitimate algorithm for computing the greatest common divisor of two positive integers. Note that special care needs to be exercised if one or both input numbers are equal to 1: because mathematicians do not consider 1 to be a prime number, strictly speaking, the method does not work for such inputs.

Before we leave this section, one more comment is in order. The examples considered in this section notwithstanding, the majority of algorithms in use today—even those that are implemented as computer programs—do not deal with mathematical problems. Look around for algorithms helping us through our daily routines, both professional and personal. May this ubiquity of algorithms in today's world strengthen your resolve to learn more about these fascinating engines of the information age.

Exercises 1.1

1. Do some research on al-Khorezmi (also al-Khwarizmi), the man from whose name the word “algorithm” is derived. In particular, you should learn what the origins of the words “algorithm” and “algebra” have in common.
2. Given that the official purpose of the U.S. patent system is the promotion of the “useful arts,” do you think algorithms are patentable in this country? Should they be?

3. a. Write down driving directions for going from your school to your home with the precision required by an algorithm.
 b. Write down a recipe for cooking your favorite dish with the precision required by an algorithm.
4. Design an algorithm for swapping two 3 digit non-zero integers n, m . Besides using arithmetic operations, your algorithm should not use any temporary variable.
5. Design an algorithm for computing $\text{gcd}(m, n)$ using Euclid's algorithm.
6. Prove the equality $\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$ for every pair of positive integers m and n .
7. What does Euclid's algorithm do for a pair of numbers in which the first number is smaller than the second one? What is the largest number of times this can happen during the algorithm's execution on such an input?
8. What is the smallest and the largest number of divisions possible in the algorithm for determining a prime number?
9. a. Euclid's algorithm, as presented in Euclid's treatise, uses subtractions rather than integer divisions. Write a pseudocode for this version of Euclid's algorithm.
 b. *Euclid's game* (see [Bog]) starts with two unequal positive numbers on the board. Two players move in turn. On each move, a player has to write on the board a positive number equal to the difference of two numbers already on the board; this number must be new, i.e., different from all the numbers already on the board. The player who cannot move loses the game. Should you choose to move first or second in this game?
10. The *extended Euclid's algorithm* determines not only the greatest common divisor d of two positive integers m and n but also integers (not necessarily positive) x and y , such that $mx + ny = d$.
 - a. Look up a description of the extended Euclid's algorithm (see, e.g., [Knul], p. 13) and implement it in the language of your choice.
 - b. Modify your program for finding integer solutions to the Diophantine equation $ax + by = c$ with any set of integer coefficients a, b , and c .
11. *Locker doors* There are n lockers in a hallway, numbered sequentially from 1 to n . Initially all the locker doors are closed. You make n passes by the lockers, each time starting with locker #1. On the i th pass, $i = 1, 2, \dots, n$, you toggle the door of every i th locker: if the door is closed, you open it; if it is open, you close it. For example, after the first pass every door is open; on the second pass you only toggle the even-numbered lockers (#2, #4, ...) so that after the second pass the even doors are closed and the odd ones are open; the third time through, you close the door of locker #3 (opened from the first pass), open the door of locker #6 (closed from the second pass), and so on. After the last pass, which locker doors are open and which are closed? How many of them are open?



1.2 Fundamentals of Algorithmic Problem Solving

Let us start by reiterating an important point made in the introduction to this chapter:

We can consider algorithms to be procedural solutions to problems.

These solutions are not answers but specific instructions for getting answers. It is this emphasis on precisely defined constructive procedures that makes computer science distinct from other disciplines. In particular, this distinguishes it from theoretical mathematics whose practitioners are typically satisfied with just proving the existence of a solution to a problem and, possibly, investigating the solution's properties.

We now list and briefly discuss a sequence of steps one typically goes through in designing and analyzing an algorithm (Figure 1.2).

Understanding the Problem

From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's

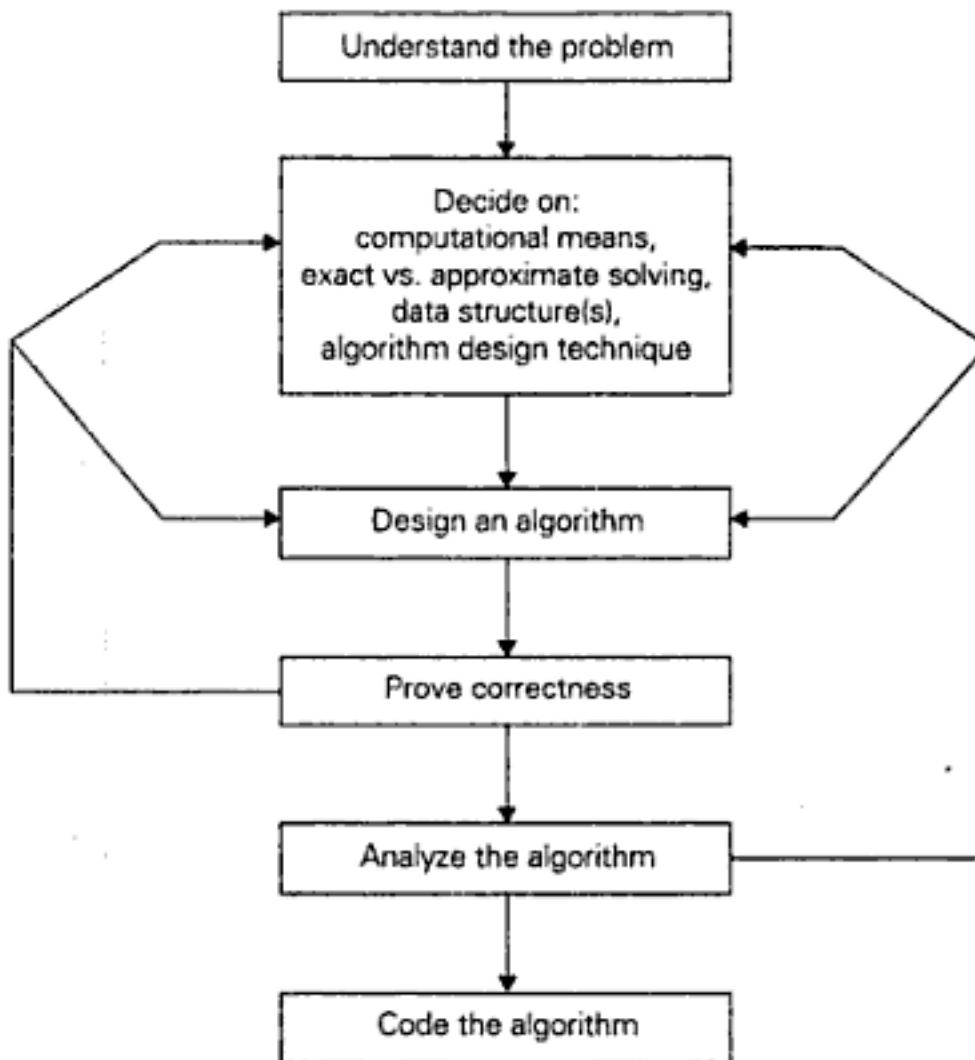


FIGURE 1.2 Algorithm design and analysis process

description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

There are a few types of problems that arise in computing applications quite often. We review them in the next section. If the problem in question is one of them, you might be able to use a known algorithm for solving it. Of course, it helps to understand how such an algorithm works and to know its strengths and weaknesses, especially if you have to choose among several available algorithms. But often, you will not find a readily available algorithm and will have to design your own. The sequence of steps outlined in this section should help you in this exciting but not always easy task.

An input to an algorithm specifies an *instance* of the problem the algorithm solves. It is very important to specify exactly the range of instances the algorithm needs to handle. (As an example, recall the variations in the range of instances for the three greatest common divisor algorithms discussed in the previous section.) If you fail to do this, your algorithm may work correctly for a majority of inputs but crash on some “boundary” value. Remember that a correct algorithm is not one that works most of the time, but one that works correctly for *all* legitimate inputs.

Do not skimp on this first step of the algorithmic problem-solving process; if you do, you will run the risk of unnecessary rework.

Ascertaining the Capabilities of a Computational Device

Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine—a computer architecture outlined by the prominent Hungarian-American mathematician John von Neumann (1903–1957), in collaboration with A. Burks and H. Goldstine, in 1946. The essence of this architecture is captured by the so-called *random-access machine (RAM)*. Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called *sequential algorithms*.

The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called *parallel algorithms*. Still, studying the classic techniques for design and analysis of algorithms under the RAM model remains the cornerstone of algorithmics for the foreseeable future.

Should you worry about the speed and amount of memory of a computer at your disposal? If you are designing an algorithm as a scientific exercise, the answer is a qualified no. As you will see in Section 2.1, most computer scientists prefer to study algorithms in terms independent of specification parameters for a particular computer. If you are designing an algorithm as a practical tool, the answer may depend on a problem you need to solve. Even “slow” computers of today are almost unimaginably fast. Consequently, in many situations, you need not worry about a computer being too slow for the task. There are important

problems, however, that are very complex by their nature, have to process huge volumes of data, or deal with applications where time is critical. In such situations, it is imperative to be aware of the speed and memory available on a particular computer system.

Choosing between Exact and Approximate Problem Solving

The next principal decision is to choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an *exact algorithm*; in the latter case, an algorithm is called an *approximation algorithm*. Why would one opt for an approximation algorithm? First, there are important problems that simply cannot be solved exactly for most of their instances; examples include extracting square roots, solving nonlinear equations, and evaluating definite integrals. Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity. This happens, in particular, for many problems involving a very large number of choices; you will see examples of such difficult problems in Chapters 3, 11, and 12. Third, an approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

Deciding on Appropriate Data Structures

Some algorithms do not demand any ingenuity in representing their inputs. But others are, in fact, predicated on ingenious data structures. In addition, some of the algorithm design techniques we shall discuss in Chapters 6 and 7 depend intimately on structuring or restructuring data specifying a problem's instance. Many years ago, an influential textbook proclaimed the fundamental importance of both algorithms and data structures for computer programming by its very title: *Algorithms + Data Structures = Programs* [Wir76]. In the new world of object-oriented programming, data structures remain crucially important for both design and analysis of algorithms. We review basic data structures in Section 1.4.

Algorithm Design Techniques

Now, with all the components of the algorithmic problem solving in place, how do you design an algorithm to solve a given problem? This is the main question this book seeks to answer by teaching you several general design techniques.

What is an algorithm design technique?

An *algorithm design technique* (or “strategy” or “paradigm”) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Check this book's table of contents and you will see that a majority of its chapters are devoted to individual design techniques. They distill a few key ideas that have proven to be useful in designing algorithms. Learning these techniques is of utmost importance for the following reasons:

First, they provide guidance for designing algorithms for new problems, i.e., problems for which there is no known satisfactory algorithm. Therefore—to use the language of a famous proverb—learning such techniques is akin to learning to fish as opposed to being given a fish caught by somebody else. It is not true, of course, that each of these general techniques will be necessarily applicable to every problem you may encounter. But taken together, they do constitute a powerful collection of tools that you will find quite handy in your studies and work.

Second, algorithms are the cornerstone of computer science. Every science is interested in classifying its principal subject, and computer science is no exception. Algorithm design techniques make it possible to classify algorithms according to an underlying design idea; therefore, they can serve as a natural way to both categorize and study algorithms.

Methods of Specifying an Algorithm

Once you have designed an algorithm, you need to specify it in some fashion. In Section 1.1, to give you an example, we described Euclid's algorithm in words (in a free and also a step-by-step form) and in pseudocode. These are the two options that are most widely used nowadays for specifying algorithms.

Using a natural language has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult. Nevertheless, being able to do this is an important skill that you should strive to develop in the process of learning algorithms.

A *pseudocode* is a mixture of a natural language and programming language-like constructs. A pseudocode is usually more precise than a natural language, and its usage often yields more succinct algorithm descriptions. Surprisingly, computer scientists have never agreed on a single form of pseudocode, leaving textbook authors to design their own “dialects.” Fortunately, these dialects are so close to each other that anyone familiar with a modern programming language should be able to understand them all.

This book's dialect was selected to cause minimal difficulty for a reader. For the sake of simplicity, we omit declarations of variables and use indentation to show the scope of such statements as **for**, **if**, and **while**. As you saw in the previous section, we use an arrow \leftarrow for the assignment operation and two slashes // for comments.

In the earlier days of computing, the dominant vehicle for specifying algorithms was a *flowchart*, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps. This representation technique has proved to be inconvenient for all but very simple algorithms; nowadays, it can be found only in old algorithm books.

The state of the art of computing has not yet reached a point where an algorithm's description—whether in a natural language or a pseudocode—can be fed into an electronic computer directly. Instead, it needs to be converted into a computer program written in a particular computer language. We can look at such a program as yet another way of specifying the algorithm, although it is preferable to consider it as the algorithm's implementation.

Proving an Algorithm's Correctness

Once an algorithm has been specified, you have to prove its *correctness*. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time. For example, correctness of Euclid's algorithm for computing the greatest common divisor stems from correctness of the equality $\gcd(m, n) = \gcd(n, m \bmod n)$ (which, in turn, needs a proof; see Problem 6 in Exercises 1.1), the simple observation that the second number gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second number becomes 0.

For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex. A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs. It might be worth mentioning that although tracing the algorithm's performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm's correctness conclusively. But in order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails. If the algorithm is found to be incorrect, you need to either redesign it under the same decisions regarding the data structures, the design technique, and so on, or, in a more dramatic reversal, to reconsider one or more of those decisions (see Figure 1.2).

The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. For an approximation algorithm, we usually would like to be able to show that the error produced by the algorithm does not exceed a predefined limit. You can find examples of such investigations in Chapter 12.

Analyzing an Algorithm

We usually want our algorithms to possess several qualities. After correctness, by far the most important is efficiency. In fact, there are two kinds of algorithm efficiency: time efficiency and space efficiency. **Time efficiency** indicates how fast the algorithm runs; **space efficiency** indicates how much extra memory the algorithm needs. A general framework and specific techniques for analyzing an algorithm's efficiency appear in Chapter 2.

Another desirable characteristic of an algorithm is *simplicity*. Unlike efficiency, which can be precisely defined and investigated with mathematical rigor, simplicity, like beauty, is to a considerable degree in the eye of the beholder. For example, most people would agree that Euclid's algorithm is simpler than the middle-school procedure for computing $\gcd(m, n)$, but it is not clear whether Euclid's algorithm is simpler than the consecutive integer checking algorithm. Still, simplicity is an important algorithm characteristic to strive for. Why? Because simpler algorithms are easier to understand and easier to program; consequently, the resulting programs usually contain fewer bugs. There is also the undeniable aesthetic appeal of simplicity. Sometimes simpler algorithms are also more efficient than more complicated alternatives. Unfortunately, it is not always true, in which case a judicious compromise needs to be made.

Yet another desirable characteristic of an algorithm is *generality*. There are, in fact, two issues here: generality of the problem the algorithm solves and the range of inputs it accepts. On the first issue, note that it is sometimes easier to design an algorithm for a problem posed in more general terms. Consider, for example, the problem of determining whether two integers are relatively prime, i.e., whether their only common divisor is equal to 1. It is easier to design an algorithm for a more general problem of computing the greatest common divisor of two integers and, to solve the former problem, check whether the gcd is 1 or not. There are situations, however, where designing a more general algorithm is unnecessary or difficult or even impossible. For example, it is unnecessary to sort a list of n numbers to find its median, which is its $[n/2]$ th smallest element. To give another example, the standard formula for roots of a quadratic equation cannot be generalized to handle polynomials of arbitrary degrees.

As to the range of inputs, your main concern should be designing an algorithm that can handle a range of inputs that is natural for the problem at hand. For example, excluding integers equal to 1 as possible inputs for a greatest common divisor algorithm would be quite unnatural. On the other hand, although the standard formula for the roots of a quadratic equation holds for complex coefficients, we would normally not implement it on this level of generality unless this capability is explicitly required.

If you are not satisfied with the algorithm's efficiency, simplicity, or generality, you must return to the drawing board and redesign the algorithm. In fact, even if your evaluation is positive, it is still worth searching for other algorithmic solutions. Recall the three different algorithms in the previous section for computing the greatest common divisor; generally, you should not expect to get the best algorithm on the first try. At the very least, you should try to fine-tune the algorithm you already have. For example, we made several improvements in our implementation of the sieve of Eratosthenes compared with its initial outline in Section 1.1. (Can you identify them?) You will do well if you keep in mind the following observation of Antoine de Saint-Exupéry, the French writer, pilot, and aircraft designer: "A designer knows he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away."¹

Coding an Algorithm

Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril and an opportunity. The peril lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently. Some influential computer scientists strongly believe that unless the correctness of a computer program is proven with full mathematical rigor, the program cannot be considered correct. They have developed special techniques for doing such proofs (see [Gri81]), but the power of these techniques of formal verification is limited so far to very small programs.

1. I found this call for design simplicity in an essay collection by Jon Bentley [Ben00]; the essays deal with a variety of issues in algorithm design and implementation, and are justifiably titled *Programming Pearls*. I wholeheartedly recommend writings of both Jon Bentley and Antoine de Saint-Exupéry.

As a practical matter, the validity of programs is still established by testing. Testing of computer programs is an art rather than a science, but that does not mean that there is nothing in it to learn. Look up books devoted to testing and debugging; even more important, test and debug your program thoroughly whenever you implement an algorithm.

Also note that throughout the book, we assume that inputs to algorithms fall within their specified ranges and hence require no verification. When implementing algorithms as programs to be used in actual applications, you should provide such verifications.

Of course, implementing an algorithm correctly is necessary but not sufficient: you would not like to diminish your algorithm's power by an inefficient implementation. Modern compilers do provide a certain safety net in this regard, especially when they are used in their code optimization mode. Still, you need to be aware of such standard tricks as computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, and so on. (See [Ker99] and [Ben00] for a good discussion of code tuning and other issues related to algorithm programming.) Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by orders of magnitude. But once an algorithm is selected, a 10–50% speedup may be worth an effort.

A working program provides an additional opportunity in allowing an empirical analysis of the underlying algorithm. The analysis is based on timing the program on several inputs and then analyzing the results obtained. We discuss the advantages and disadvantages of this approach to analyzing algorithms in Section 2.6.

In conclusion, let us emphasize again the main lesson of the process depicted in Figure 1.2:

As a rule, a good algorithm is a result of repeated effort and rework.

Even if you have been fortunate enough to get an algorithmic idea that seems perfect, you should still try to see whether it can be improved.

Actually, this is good news since it makes the ultimate result so much more enjoyable. (Yes, I did think of naming this book *The Joy of Algorithms*.) On the other hand, how does one know when to stop? In the real world, more often than not the project's schedule or the patience of your boss will stop you. And so it should be: perfection is expensive and in fact not always called for. Designing an algorithm is an engineering-like activity that calls for compromises among competing goals under the constraints of available resources, with the designer's time being one of the resources.

In the academic world, the question leads to an interesting but usually difficult investigation of an algorithm's *optimality*. Actually, this question is not about the efficiency of an algorithm but about the complexity of the problem it solves: what is the minimum amount of effort *any* algorithm will need to exert to solve the problem in question? For some problems, the answer to this question is known. For example, any algorithm that sorts an array by comparing values of its elements

needs about $n \log_2 n$ comparisons for some arrays of size n (see Section 11.2). But for many seemingly easy problems, such as matrix multiplication, computer scientists do not yet have a final answer.

Another important issue of algorithmic problem solving is the question of whether or not every problem can be solved by an algorithm. We are not talking here about problems that do not have a solution, such as finding real roots of a quadratic equation with a negative discriminant. For such cases, an output indicating that the problem does not have a solution is all we can and should expect from an algorithm. Nor are we talking about ambiguously stated problems. Even some unambiguous problems that must have a simple yes or no answer are “undecidable,” i.e., unsolvable by any algorithm. An important example of such a problem appears in Section 11.3. Fortunately, a vast majority of problems in practical computing *can* be solved by an algorithm.

Before leaving this section, let us be sure that you do not have the misconception—possibly caused by the somewhat mechanical nature of the diagram of Figure 1.2—that designing an algorithm is a dull activity. There is nothing further from the truth: inventing (or discovering?) algorithms is a very creative and rewarding process. This book is designed to convince you that this is the case.

Exercises 1.2



1. *Old World puzzle* A peasant finds himself on a riverbank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river in his boat. However, the boat has room for only the peasant himself and one other item (either the wolf, the goat, or the cabbage). In his absence, the wolf would eat the goat, and the goat would eat the cabbage. Solve this problem for the peasant or prove it has no solution. (Note: The peasant is a vegetarian but does not like cabbage and hence can eat neither the goat nor the cabbage to help him solve the problem. And it goes without saying that the wolf is a protected species.)



2. *New World puzzle* There are four people who want to cross a bridge; they all begin on the same side. You have 17 minutes to get them all across to the other side. It is night, and they have one flashlight. A maximum of two people can cross the bridge at one time. Any party that crosses, either one or two people, must have the flashlight with them. The flashlight must be walked back and forth; it cannot be thrown, for example. Person 1 takes 1 minute to cross the bridge, person 2 takes 2 minutes, person 3 takes 5 minutes, and person 4 takes 10 minutes. A pair must walk together at the rate of the slower person’s pace. For example, if person 1 and person 4 walk across first, 10 minutes have elapsed when they get to the other side of the bridge. If person 4 returns the flashlight, a total of 20 minutes have passed and you have failed the mission. (Note: According to a rumor on the Internet, interviewers at a well-known software company located near Seattle have given this problem to interviewees.)

3. Design an algorithm to compute the area and circumference of a circle.

4. Write a pseudocode for an algorithm for finding real roots of equation $ax^2 + bx + c = 0$ for arbitrary real coefficients a , b , and c . (You may assume the availability of the square root function $\text{sqrt}(x)$.)
5. Design an algorithm to convert a binary number to a decimal integer.
6. Describe the algorithm used by your favorite ATM machine in dispensing cash. (You may give your description in either English or a pseudocode, whichever you find more convenient.)
7.
 - a. Can the problem of computing the number π be solved exactly?
 - b. How many instances does this problem have?
 - c. Look up an algorithm for this problem on the World Wide Web.
8. How many algorithms can you write for solving to find the prime numbers? Indicate which is the simplest and the most efficient?
9. Consider the following algorithm for finding the distance between the two closest elements in an array of numbers.

```
ALGORITHM MinDistance( $A[0..n - 1]$ )  
//Input: Array  $A[0..n - 1]$  of numbers  
//Output: Minimum distance between two of its elements  
 $dmin \leftarrow \infty$   
for  $i \leftarrow 0$  to  $n - 1$  do  
    for  $j \leftarrow 0$  to  $n - 1$  do  
        if  $i \neq j$  and  $|A[i] - A[j]| < dmin$   
             $dmin \leftarrow |A[i] - A[j]|$   
return  $dmin$ 
```

Make as many improvements as you can in this algorithmic solution to the problem. (If you need to, you may change the algorithm altogether; if not, improve the implementation given.)

10. One of the most influential books on problem solving, titled *How to Solve It* [Pol57], was written by the Hungarian-American mathematician George Polya (1887–1985). Polya summarized his ideas in a four-point summary. Find this summary on the Web or, better yet, in his book, and compare it with the plan outlined in Section 1.2. What do they have in common? How are they different?

1.3 Important Problem Types

In the limitless sea of problems one encounters in computing, there are a few areas that have attracted particular attention from researchers. By and large, interest has been driven either by the problem's practical importance or by some specific characteristics making the problem an interesting research subject; fortunately, these two motivating forces reinforce each other in most cases.

In this section, we take up the most important problem types:

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

These problems are used in subsequent chapters of the book to illustrate different algorithm design techniques and methods of algorithm analysis.

Sorting

The **sorting problem** asks us to rearrange the items of a given list in ascending order. Of course, for this problem to be meaningful, the nature of the list items must allow such an ordering. (Mathematicians would say that there must exist a relation of total ordering.) As a practical matter, we usually need to sort lists of numbers, characters from an alphabet, character strings, and, most important, records similar to those maintained by schools about their students, libraries about their holdings, and companies about their employees. In the case of records, we need to choose a piece of information to guide sorting. For example, we can choose to sort student records in alphabetical order of names or by student number or by student grade point average. Such a specially chosen piece of information is called a **key**. Computer scientists often talk about sorting a list of keys even when the list's items are not records but, say, just integers.

Why would we want a sorted list? Well, sorting makes many questions about the list easier to answer. The most important of them is searching: it is why dictionaries, telephone books, class lists, and so on are sorted. You will see other examples of the usefulness of list presorting in Section 6.1. In a similar vein, sorting is used as an auxiliary step in several important algorithms in other areas, e.g., geometric algorithms.

By now, computer scientists have discovered dozens of different sorting algorithms. In fact, inventing a new sorting algorithm has been likened to designing the proverbial mousetrap. And I am happy to report that the hunt for a better sorting mousetrap continues. This perseverance is admirable in view of the following facts. On the one hand, there are a few good sorting algorithms that sort an arbitrary array of size n using about $n \log_2 n$ comparisons. On the other hand, no algorithm that sorts by key comparisons (as opposed to, say, comparing small pieces of keys) can do substantially better than that.

There is a reason for this embarrassment of algorithmic riches in the land of sorting. Although some algorithms are indeed better than others, there is no algorithm that would be the best solution in all situations. Some of the algorithms are simple but relatively slow while others are faster but more complex; some work better on randomly ordered inputs while others do better on almost sorted lists; some are suitable only for lists residing in the fast memory while others can be adapted for sorting large files stored on a disk; and so on.

Two properties of sorting algorithms deserve special mention. A sorting algorithm is called **stable** if it preserves the relative order of any two equal elements in its input. In other words, if an input list contains two equal elements in positions i and j where $i < j$, then in the sorted list they have to be in positions i' and j' , respectively, such that $i' < j'$. This property can be desirable if, for example, we have a list of students sorted alphabetically and we want to sort it according to student GPA: a stable algorithm will yield a list in which students with the same GPA will still be sorted alphabetically. Generally speaking, algorithms that can exchange keys located far apart are not stable but they usually work faster; you will see how this general comment applies to important sorting algorithms later in the book.

The second notable feature of a sorting algorithm is the amount of extra memory the algorithm requires. An algorithm is said to be **in place** if it does not require extra memory, except, possibly, for a few memory units. There are important sorting algorithms that are in place and those that are not.

Searching

The **searching problem** deals with finding a given value, called a **search key**, in a given set (or a multiset, which permits several elements to have the same value). There are plenty of searching algorithms to choose from. They range from the straightforward sequential search to a spectacularly efficient but limited binary search and algorithms based on representing the underlying set in a different form more conducive to searching. The latter algorithms are of particular importance for real-life applications because they are indispensable for storing and retrieving information from large databases.

For searching, too, there is no single algorithm that fits all situations best. Some algorithms work faster than others but require more memory; some are very fast but applicable only to sorted arrays; and so on. Unlike with sorting algorithms, there is no stability problem, but different issues arise. Specifically, in applications where the underlying data may change frequently relative to the number of searches, searching has to be considered in conjunction with two other operations: addition to and deletion from the data set of an item. In such situations, data structures and algorithms should be chosen to strike a balance among the requirements of each operation. Also, organizing very large data sets for efficient searching poses special challenges with important implications for real-life applications.

String Processing

In recent years, the rapid proliferation of applications dealing with nonnumerical data has intensified the interest of researchers and computing practitioners in string-handling algorithms. A **string** is a sequence of characters from an alphabet. Strings of particular interest are text strings, which comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones; and gene sequences, which can be modeled by strings of characters from the four-character alphabet {A, C, G, T}. It should be pointed out, however, that string-processing algorithms have been important for computer science for a long time in conjunction with computer languages and compiling issues.

One particular problem—that of searching for a given word in a text—has attracted special attention from researchers. They call it **string matching**. Several algorithms that exploit the special nature of this type of searching have been invented. We introduce one very simple algorithm in Chapter 3, and discuss two algorithms based on a remarkable idea by R. Boyer and J. Moore in Chapter 7.

Graph Problems

One of the oldest and most interesting areas in algorithmics is graph algorithms. Informally, a **graph** can be thought of as a collection of points called vertices, some of which are connected by line segments called edges. (A more formal definition is given in the next section.) Graphs are an interesting subject to study for both theoretical and practical reasons. Graphs can be used for modeling a wide variety of real-life applications, including transportation and communication networks, project scheduling, and games. One interesting recent application is an estimation of the Web's diameter, which is the maximum number of links one needs to follow to reach one Web page from another by the most direct route between them.²

Basic graph algorithms include graph traversal algorithms (How can one visit all the points in a network?), shortest-path algorithms (What is the best route between two cities?), and topological sorting for graphs with directed edges (Is a set of courses with their prerequisites consistent or self-contradictory?). Fortunately, these algorithms can be considered illustrations of general design techniques; accordingly, you will find them in corresponding chapters of the book.

Some graph problems are computationally very hard; the most well-known examples are the traveling salesman problem and the graph-coloring problem. The **traveling salesman problem (TSP)** is the problem of finding the shortest tour through n cities that visits every city exactly once. In addition to obvious applications involving route planning, it arises in such modern applications as circuit board and VLSI chip fabrication, X-ray crystallography, and genetic engineering. The **graph-coloring problem** asks us to assign the smallest number of colors to vertices of a graph so that no two adjacent vertices are the same color. This problem arises in several applications, such as event scheduling: if the events are represented by vertices that are connected by an edge if and only if the corresponding events cannot be scheduled in the same time, a solution to the graph-coloring problem yields an optimal schedule.

Combinatorial Problems

From a more abstract perspective, the traveling salesman problem and the graph-coloring problem are examples of **combinatorial problems**. These are problems that ask (explicitly or implicitly) to find a combinatorial object—such as a permutation, a combination, or a subset—that satisfies certain constraints and has some desired property (e.g., maximizes a value or minimizes a cost).

2. This number, according to an estimate by a group of researchers at the University of Notre Dame [Alb99], is just 19.

Generally speaking, combinatorial problems are the most difficult problems in computing, from both the theoretical and practical standpoints. Their difficulty stems from the following facts. First, the number of combinatorial objects typically grows extremely fast with a problem's size, reaching unimaginable magnitudes even for moderate-sized instances. Second, there are no known algorithms for solving most such problems exactly in an acceptable amount of time. Moreover, most computer scientists believe that such algorithms do not exist. This conjecture has been neither proved nor disproved, and it remains the most important unresolved issue in theoretical computer science. We discuss this topic in more detail in Section 11.3.

Some combinatorial problems can be solved by efficient algorithms, but they should be considered fortunate exceptions to the rule. The shortest-path problem mentioned earlier is among such exceptions.

Geometric Problems

Geometric algorithms deal with geometric objects such as points, lines, and polygons. Ancient Greeks were very much interested in developing procedures (they did not call them algorithms, of course) for solving a variety of geometric problems, including problems of constructing simple geometric shapes—triangles, circles, and so on—with an unmarked ruler and a compass. Then, for about 2000 years, intense interest in geometric algorithms disappeared, to be resurrected in the age of computers—no more rulers and compasses, just bits, bytes, and good old human ingenuity. Of course, today people are interested in geometric algorithms with quite different applications in mind, such as computer graphics, robotics, and tomography.

We will discuss algorithms for only two classic problems of computational geometry: the closest-pair problem and the convex-hull problem. The **closest-pair problem** is self-explanatory: given n points in the plane, find the closest pair among them. The **convex-hull problem** asks to find the smallest convex polygon that would include all the points of a given set. If you are interested in other geometric algorithms, you will find a wealth of material in specialized monographs (e.g., [ORo98]), or corresponding chapters of textbooks organized around problem types (e.g., [Sed88]).

Numerical Problems

Numerical problems, another large special area of applications, are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on. The majority of such mathematical problems can be solved only approximately. Another principal difficulty stems from the fact that such problems typically require manipulating real numbers, which can be represented in a computer only approximately. Moreover, a large number of arithmetic operations performed on approximately represented numbers can lead to an accumulation of the round-off error to a point where it can drastically distort an output produced by a seemingly sound algorithm.

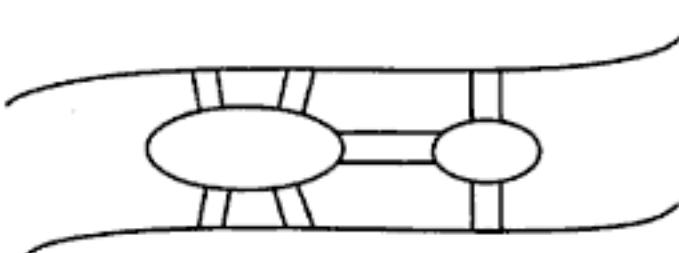
Many sophisticated algorithms have been developed over the years in this area, and they continue to play a critical role in many scientific and engineering applications. But in the last 25 years or so, the computing industry has shifted its focus to business applications. These new applications require primarily algorithms for information storage, retrieval, transportation through networks, and presentation to users. As a result of this revolutionary change, numerical analysis has lost its formerly dominating position in both industry and computer science programs. Still, it is important for any computer-literate person to have at least a rudimentary idea about numerical algorithms. We discuss several classical numerical algorithms in Sections 6.2, 11.4, and 12.4.

Exercises 1.3

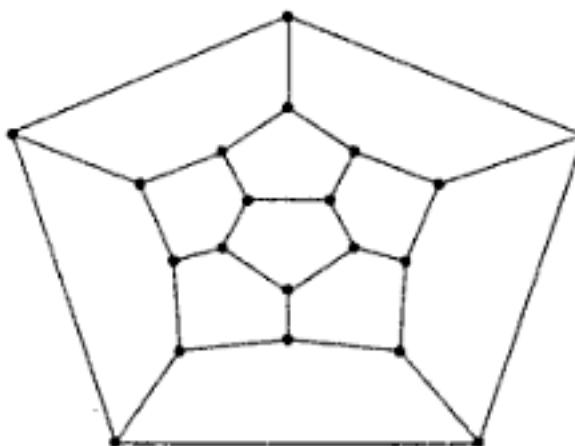
1. Consider the following algorithm for the searching problem:

```
ALGORITHM Linearssearch ( $A[0..n - 1]$ , key)
//Searches an array for a key value by Linear search
//Input: Array  $A[0..n - 1]$  of values and a key value to search
//Output: Returns index if search is successful
for  $i \leftarrow 0$  to  $n - 1$  do
    if (key ==  $A[i]$ )
        return  $i$ 
```

- a. Apply this algorithm to search the list 10, 92, 38, 74, 56, 19, 82, 37 for a key value 74.
b. Is this algorithm efficient?
c. When can this algorithm be used?
2. Name the algorithms for the sorting problem that you already know. Give a good and succinct description of each algorithm in English. (If you do not know any such algorithms, use this opportunity to design one.)
3. Design a simple algorithm for the string-matching problem.
4. *Königsberg bridges* The Königsberg bridge puzzle is universally accepted as the problem that gave birth to graph theory. It was solved by the great Swiss-born mathematician Leonhard Euler (1707–1783). The problem asked whether one could, in a single stroll, cross all seven bridges of the city of Königsberg exactly once and return to a starting point. Following is a sketch of the river with its two islands and seven bridges:

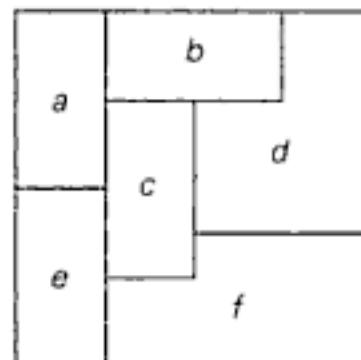


- State the problem as a graph problem.
 - Does this problem have a solution? If you believe it does, draw such a stroll; if you believe it does not, explain why and indicate the smallest number of new bridges that would be required to make such a stroll possible.
-  5. *Icosian Game* A century after Euler's discovery (see Problem 4), another famous puzzle—this one invented by the renown Irish mathematician Sir William Hamilton (1805–1865)—was presented to the world under the name of the Icosian Game. The game was played on a circular wooden board on which the following graph was carved:



Find a **Hamiltonian circuit**—a path that visits all the graph's vertices exactly once before returning to the starting vertex—for this graph.

- The problem's statement is somewhat vague, which is typical of real-life problems. In particular, what reasonable criterion can be used for defining the “best” route?
 - How would you model this problem by a graph?
- Rephrase the water jug problem in combinatorial object terms.
 - Rephrase the 8-Queen problem in combinatorial object terms.
- Consider the following map:



- a. Explain how we can use the graph-coloring problem to color the map so that no two neighboring regions are colored the same.
- b. Use your answer to part (a) to color the map with the smallest number of colors.
9. Design an algorithm for the following problem: Given a set of n points in the Cartesian plane, determine whether all of them lie on the same circumference.
10. Write a program that reads as its inputs the (x, y) coordinates of the endpoints of two line segments P_1Q_1 and P_2Q_2 and determines whether the segments have a common point.

1.4 Fundamental Data Structures

Since the vast majority of algorithms of interest operate on data, particular ways of organizing data play a critical role in the design and analysis of algorithms. A ***data structure*** can be defined as a particular scheme of organizing related data items. The nature of the data items is dictated by a problem at hand; they can range from elementary data types (e.g., integers or characters) to data structures (e.g., a one-dimensional array of one-dimensional arrays is often used for implementing matrices). There are a few data structures that have proved to be particularly important for computer algorithms. Since you are undoubtedly familiar with most if not all of them, just a quick review is provided here.

Linear Data Structures

The two most important elementary data structures are the array and the linked list. A (one-dimensional) ***array*** is a sequence of n items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's ***index*** (Figure 1.3).

In the majority of cases, the index is an integer either between 0 and $n - 1$ (as shown in Figure 1.3) or between 1 and n . Some computer languages allow an array index to range between any two integer bounds *low* and *high*, and some even permit nonnumerical indices to specify, for example, data items corresponding to the 12 months of the year by the month names.

Each and every element of an array can be accessed in the same constant amount of time regardless of where in the array the element in question is located. This feature positively distinguishes arrays from linked lists (see below). It is also assumed that every element of an array occupies the same amount of computer storage.

Item [0]	Item [1]	...	Item [n-1]
----------	----------	-----	------------

FIGURE 1.3 Array of n elements

Arrays are used for implementing a variety of other data structures. Prominent among them is the ***string***, a sequence of characters from an alphabet terminated by a special character indicating the string's end. Strings composed of zeros and ones are called ***binary strings*** or ***bit strings***. Strings are indispensable for processing textual data, defining computer languages and compiling programs written in them, and studying abstract computational models. Operations we usually perform on strings differ from those we typically perform on other arrays (say, arrays of numbers). They include computing the string length, comparing two strings to determine which one precedes the other according to the so-called lexicographic order, i.e., in a dictionary, and concatenating two strings (forming one string from two given strings by appending the second to the end of the first).

A ***linked list*** is a sequence of zero or more elements called ***nodes*** each containing two kinds of information: some data and one or more links called ***pointers*** to other nodes of the linked list. (A special pointer called "null" is used to indicate the absence of a node's successor.) In a ***singly linked list***, each node except the last one contains a single pointer to the next element (Figure 1.4).

To access a particular node of a linked list, we start with the list's first node and traverse the pointer chain until the particular node is reached. Thus, the time needed to access an element of a singly linked list, unlike that of an array, depends on where in the list the element is located. On the positive side, linked lists do not require any preliminary reservation of the computer memory, and insertions and deletions can be made quite efficiently in a linked list by reconnecting a few appropriate pointers.

We can exploit flexibility of the linked list structure in a variety of ways. For example, it is often convenient to start a linked list with a special node called the ***header***. This node often contains information about the linked list such as its current length; it may also contain, in addition to a pointer to the first element, a pointer to the linked list's last element.

Another extension is the structure called the ***doubly linked list***, in which every node, except the first and the last, contains pointers to both its successor and its predecessor (Figure 1.5).

The array and linked list are two principal choices in representing a more abstract data structure called a linear list or simply a list. A ***list*** is a finite sequence of data items, i.e., a collection of data items arranged in a certain linear order. The basic operations performed on this data structure are searching for, inserting, and deleting an element.

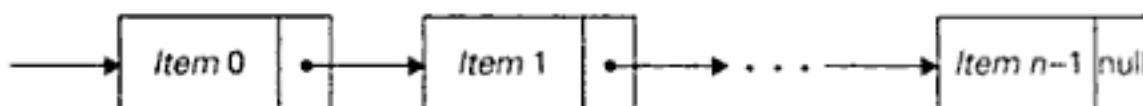


FIGURE 1.4 Singly linked list of n elements

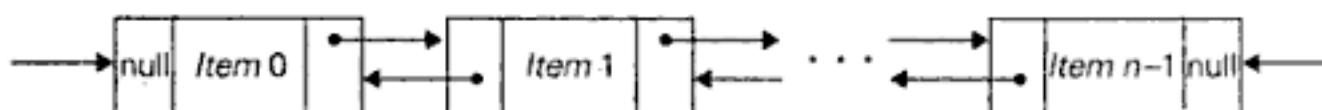


FIGURE 1.5 Doubly linked list of n elements

Two special types of lists, stacks and queues, are particularly important. A **stack** is a list in which insertions and deletions can be done only at the end. This end is called the **top** because a stack is usually visualized not horizontally but vertically (akin to a stack of plates whose “operations” it mimics very closely). As a result, when elements are added to (pushed onto) a stack and deleted from (popped off) it, the structure operates in the “last-in-first-out” (LIFO) fashion, exactly as the stack of plates does if we can remove only the top plate or add another plate to top of the stack. Stacks have a multitude of applications; in particular, they are indispensable for implementing recursive algorithms.

A **queue**, on the other hand, is a list from which elements are deleted from one end of the structure, called the **front** (this operation is called **dequeue**), and new elements are added to the other end, called the **rear** (this operation is called **enqueue**). Consequently, a queue operates in the “first-in-first-out” (FIFO) fashion (akin, say, to a queue of customers served by a single teller in a bank). Queues also have many important applications, including several algorithms for graph problems.

Many important applications require selection of an item of the highest priority among a dynamically changing set of candidates. A data structure that seeks to satisfy the needs of such applications is called a **priority queue**. A **priority queue** is a collection of data items from a totally ordered universe (most often, integer or real numbers). The principal operations on a priority queue are finding its largest element, deleting its largest element, and adding a new element. Of course, a priority queue must be implemented so that the last two operations yield another priority queue. Straightforward implementations of this data structure can be based on either an array or a sorted array, but neither of these options yields the most efficient solution possible. A better implementation of a priority queue is based on an ingenious data structure called the **heap**. We discuss heaps (and an important sorting algorithm based on them) in Section 6.4.

Graphs

As mentioned in the previous section, a graph is informally thought of as a collection of points in the plane called “vertices” or “nodes,” some of them connected by line segments called “edges” or “arcs.” Formally, a **graph** $G = (V, E)$ is defined by a pair of two sets: a finite set V of items called **vertices** and a set E of pairs of these items called **edges**. If these pairs of vertices are unordered, i.e., a pair of vertices (u, v) is the same as the pair (v, u) , we say that the vertices u and v are **adjacent** to each other and that they are connected by the **undirected edge** (u, v) . We call the vertices u and v **endpoints** of the edge (u, v) and say that u and v are **incident** to this edge; we also say that the edge (u, v) is incident to its endpoints u and v . A graph G is called **undirected** if every edge in it is undirected.

If a pair of vertices (u, v) is not the same as the pair (v, u) , we say that the edge (u, v) is **directed** from the vertex u , called the edge’s **tail**, to the vertex v , called the edge’s **head**. We also say that the edge (u, v) leaves u and enters v . A graph whose every edge is directed is called **directed**. Directed graphs are also called **digraphs**.

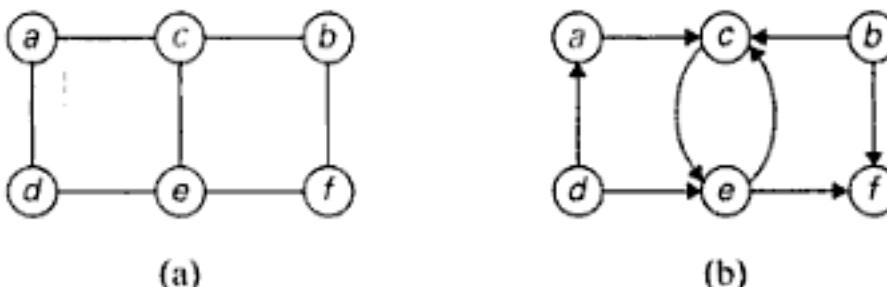


FIGURE 1.6 (a) Undirected graph. (b) Digraph.

It is normally convenient to label vertices of a graph or a digraph with letters, integer numbers, or, if an application calls for it, character strings (Figure 1.6). The graph in Figure 1.6a has six vertices and seven edges:

$$V = \{a, b, c, d, e, f\}, E = \{(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)\}.$$

The digraph in Figure 1.6b has six vertices and eight directed edges:

$$V = \{a, b, c, d, e, f\},$$

$$E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}.$$

Our definition of a graph does not forbid *loops*, or edges connecting vertices to themselves. Unless explicitly stated otherwise, we will consider graphs without loops. Since our definition disallows multiple edges between the same vertices of an undirected graph, we have the following inequality for the number of edges $|E|$ possible in an undirected graph with $|V|$ vertices and no loops:

$$0 \leq |E| \leq |V|(|V| - 1)/2.$$

(We get the largest number of edges in a graph if there is an edge connecting each of its $|V|$ vertices with all $|V| - 1$ other vertices. We have to divide product $|V|(|V| - 1)$ by 2, however, because it includes every edge twice.)

A graph with every pair of its vertices connected by an edge is called **complete**. A standard notation for the complete graph with $|V|$ vertices is $K_{|V|}$. A graph with relatively few possible edges missing is called **dense**; a graph with few edges relative to the number of its vertices is called **sparse**. Whether we are dealing with a dense or sparse graph may influence how we choose to represent the graph and, consequently, the running time of an algorithm being designed or used.

Graph representations Graphs for computer algorithms can be represented in two principal ways: the adjacency matrix and adjacency lists. The **adjacency matrix** of a graph with n vertices is an n -by- n boolean matrix with one row and one column for each of the graph's vertices, in which the element in the i th row and the j th column is equal to 1 if there is an edge from the i th vertex to the j th vertex, and equal to 0 if there is no such edge. For example, the adjacency matrix for the graph in Figure 1.6a is given in Figure 1.7a. Note that the adjacency matrix of an undirected graph is always symmetric, i.e., $A[i, j] = A[j, i]$ for every $0 \leq i, j \leq n - 1$ (why?).

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	
<i>a</i>	0	0	1	1	0	0	→ <i>c</i> → <i>d</i>
<i>b</i>	0	0	1	0	0	1	→ <i>c</i> → <i>f</i>
<i>c</i>	1	1	0	0	1	0	→ <i>a</i> → <i>b</i> → <i>e</i>
<i>d</i>	1	0	0	0	1	0	→ <i>a</i> → <i>e</i>
<i>e</i>	0	0	1	1	0	1	→ <i>c</i> → <i>d</i> → <i>f</i>
<i>f</i>	0	1	0	0	1	0	→ <i>b</i> → <i>e</i>

(a)

(b)

FIGURE 1.7 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure 1.6a

The **adjacency lists** of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex (i.e., all the vertices connected to it by an edge). Usually, such lists start with a header identifying a vertex for which the list is compiled. For example, Figure 1.7b represents the graph in Figure 1.6a via its adjacency lists. To put it another way, adjacency lists indicate columns of the adjacency matrix that, for a given vertex, contain 1's.

If a graph is sparse, the adjacency list representation may use less space than the corresponding adjacency matrix despite the extra storage consumed by pointers of the linked lists; the situation is exactly opposite for dense graphs. In general, which of the two representations is more convenient depends on the nature of the problem, on the algorithm used for solving it, and, possibly, on the type of input graph (sparse or dense).

Weighted graphs A **weighted graph** (or weighted digraph) is a graph (or digraph) with numbers assigned to its edges. These numbers are called **weights** or **costs**. An interest in such graphs is motivated by numerous real-life applications, such as finding the shortest path between two points in a transportation or communication network or the traveling salesman problem mentioned earlier.

Both principal representations of a graph can be easily adopted to accommodate weighted graphs. If a weighted graph is represented by its adjacency matrix, then its element $A[i,j]$ will simply contain the weight of the edge from the i th to the j th vertex if there is such an edge and a special symbol, e.g., ∞ , if there is no such edge. Such a matrix is called the **weight matrix** or **cost matrix**. This approach is illustrated in Figure 1.8b. (For some applications, it is more convenient to put 0's on the main diagonal of the adjacency matrix.) Adjacency lists for a weighted graph have to include in their nodes not only the name of an adjacent vertex but also the weight of the corresponding edge (Figure 1.8c).

Paths and cycles Among many interesting properties of graphs, two are important for a great number of applications: **connectivity** and **acyclicity**. Both are based on the notion of a path. A **path** from vertex u to vertex v of a graph G can be defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v . If all vertices of a path are distinct, the path is said to be **simple**. The **length** of a path is the total number of vertices in a vertex sequence defining the path minus one, which is the same as the number of edges in the path.

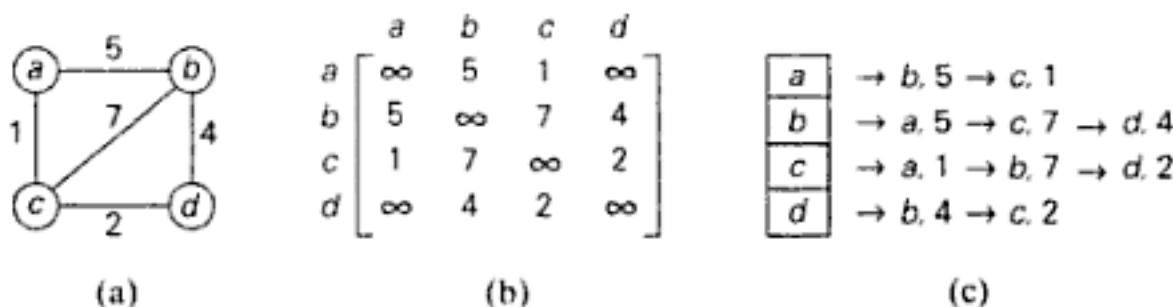


FIGURE 1.8 (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

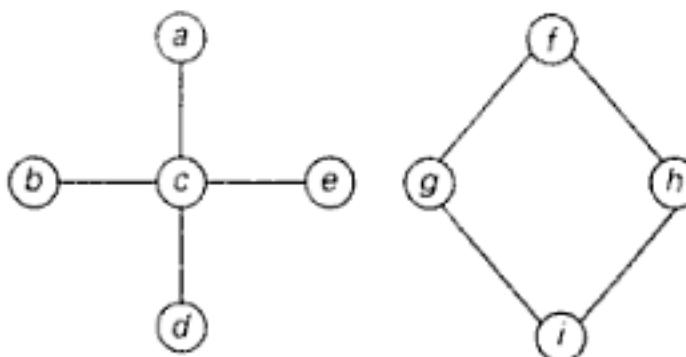


FIGURE 1.9 Graph that is not connected

For example, a, c, b, f is a simple path of length 3 from a to f in the graph of Figure 1.6a, whereas a, c, e, c, b, f is a path (not simple) of length 5 from a to f .

In the case of a directed graph, we are usually interested in directed paths. A **directed path** is a sequence of vertices in which every consecutive pair of the vertices is connected by an edge directed from the vertex listed first to the vertex listed next. For example, a, c, e, f is a directed path from a to f in the graph of Figure 1.6b.

A graph is said to be **connected** if for every pair of its vertices u and v there is a path from u to v . Informally, this property means that if we make a model of a connected graph by connecting some balls representing the graph's vertices with strings representing the edges, it will be a single piece. If a graph is not connected, such a model will consist of several connected pieces that are called connected components of the graph. Formally, a **connected component** is a maximal (not expandable via an inclusion of an extra vertex) connected subgraph³ of a given graph. For example, the graphs of Figures 1.6a and 1.8a are connected, while the graph in Figure 1.9 is not because there is no path, for example, from a to f . The graph in Figure 1.9 has two connected components with vertices $\{a, b, c, d, e\}$ and $\{f, g, h, i\}$, respectively.

Graphs with several connected components do happen in real-life applications. A graph representing the Interstate highway system of the United States would be an example (why?).

It is important to know for many applications whether or not a graph under consideration has cycles. A **cycle** is a path of a positive length that starts and ends

3. A **subgraph** of a given graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$.

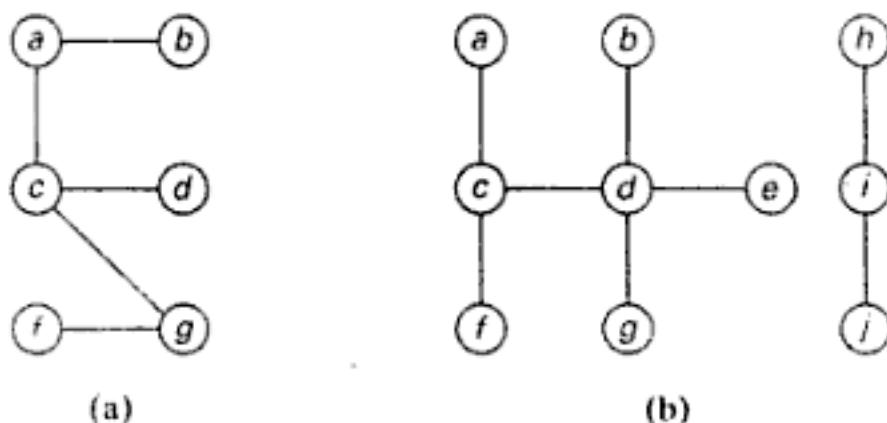


FIGURE 1.10 (a) Tree. (b) Forest.

at the same vertex and does not traverse the same edge more than once. For example, f, h, i, g, f is a cycle in the graph of Figure 1.9. A graph with no cycles is said to be *acyclic*. We discuss acyclic graphs in the next subsection.

Trees

A *tree* (more accurately, a *free tree*) is a connected acyclic graph (Figure 1.10a). A graph that has no cycles but is not necessarily connected is called a *forest*: each of its connected components is a tree (Figure 1.10b).

Trees have several important properties other graphs do not have. In particular, the number of edges in a tree is always one less than the number of its vertices:

$$|E| = |V| - 1.$$

As the graph of Figure 1.9 demonstrates, this property is necessary but not sufficient for a graph to be a tree. However, for connected graphs it is sufficient and hence provides a convenient way of checking whether a connected graph has a cycle.

Rooted trees Another very important property of trees is the fact that for every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other. This property makes it possible to select an arbitrary vertex in a free tree and consider it as the *root* of the so-called *rooted tree*. A rooted tree is usually depicted by placing its root on the top (level 0 of the tree), the vertices adjacent to the root below it (level 1), the vertices two edges apart from the root below that (level 2), and so on. Figure 1.11 presents such a transformation from a free tree to a rooted tree.

Rooted trees play a very important role in computer science, a much more important one than free trees do; in fact, for the sake of brevity, they are often referred to as simply “trees.” Obvious applications of trees are for describing hierarchies, from file directories to organizational charts of enterprises. There are many less obvious applications, such as implementing dictionaries (see below), efficient storage of very large data sets (Section 7.4), and data encoding (Section 9.4). As we discuss in Chapter 2, trees also are helpful in analysis of recursive algorithms. To finish this far-from-complete list of tree applications, we

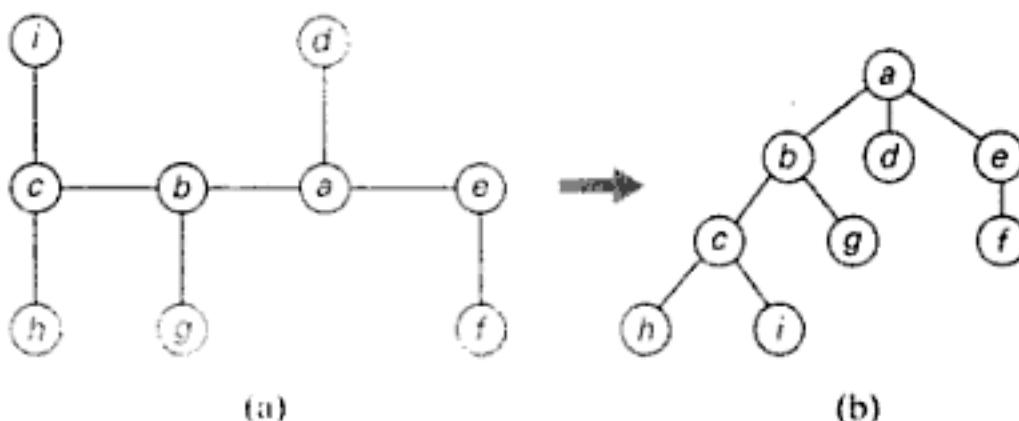


FIGURE 1.11 (a) Free tree. (b) Its transformation into a rooted tree.

should mention the so-called *state-space trees* that underline two important algorithm design techniques: backtracking and branch-and-bound (Sections 12.1 and 12.2).

For any vertex v in a tree T , all the vertices on the simple path from the root to that vertex are called *ancestors* of v . The vertex itself is usually considered its own ancestor; the set of ancestors that excludes the vertex itself is referred to as *proper ancestors*. If (u, v) is the last edge of the simple path from the root to vertex v (and $u \neq v$), u is said to be the *parent* of v and v is called a *child* of u ; vertices that have the same parent are said to be *siblings*. A vertex with no children is called a *leaf*; a vertex with at least one child is called *parental*. All the vertices for which a vertex v is an ancestor are said to be *descendants* of v ; the *proper descendants* exclude the vertex v itself. All the descendants of a vertex v with all the edges connecting them form the *subtree* of T rooted at that vertex. Thus, for the tree of Figure 1.11b, the root of the tree is a ; vertices d, g, f, h , and i are leaves, while vertices a, b, e , and c are parental; the parent of b is a ; the children of b are c and g ; the siblings of b are d and e ; the vertices of the subtree rooted at b are $\{b, c, g, h, i\}$.

The *depth* of a vertex v is the length of the simple path from the root to v . The *height* of a tree is the length of the longest simple path from the root to a leaf. For example, the depth of vertex c in the tree in Figure 1.11b is 2, and the height of the tree is 3. Thus, if we count tree levels top down starting with 0 for the root's level, the depth of a vertex is simply its level in the tree, and the tree's height is the maximum level of its vertices. (You should be alert to the fact that some authors define the height of a tree as the number of levels in it; this makes the height of a tree larger by 1 than the height defined as the length of the longest simple path from the root to a leaf.)

Ordered trees An *ordered tree* is a rooted tree in which all the children of each vertex are ordered. It is convenient to assume that in a tree's diagram, all the children are ordered left to right. A *binary tree* can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either a *left child* or a *right child* of its parent. The subtree with its root at the left (right) child of a vertex is called the *left (right) subtree* of that vertex. An example of a binary tree is given in Figure 1.12a.

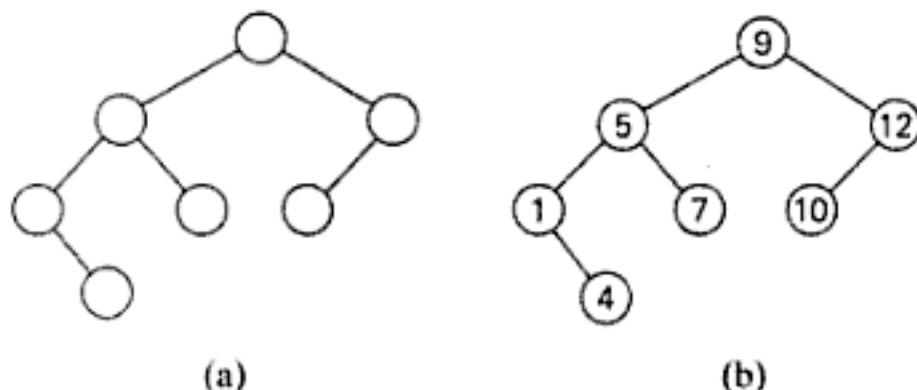


FIGURE 1.12 (a) Binary tree. (b) Binary search tree.

In Figure 1.12b, some numbers are assigned to vertices of the binary tree in Figure 1.12a. Note that a number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree. Such trees are called ***binary search trees***. Binary trees and binary search trees have a wide variety of applications in computer science; you will encounter some of them throughout the book. In particular, binary search trees can be generalized to more general kinds of search trees called ***multiway search trees***, which are indispensable for efficient storage of very large files on disks.

As you will see later in the book, the efficiency of most important algorithms for binary search trees and their extensions depends on the tree's height. Therefore, the following inequalities for the height h of a binary tree with n nodes are especially important for analysis of such algorithms:

$$\lfloor \log_2 n \rfloor \leq h \leq n - 1.$$

A binary tree is usually implemented for computing purposes by a collection of nodes corresponding to vertices of the tree. Each node contains some information associated with the vertex (its name or some value assigned to it) and two pointers to the nodes representing the left child and right child of the vertex, respectively. Figure 1.13 illustrates such an implementation for the binary search tree in Figure 1.12b.

A computer representation of an arbitrary ordered tree can be done by simply providing a parental vertex with the number of pointers equal to the number of its children. This representation may prove to be inconvenient if the number of children varies widely among the nodes. We can avoid this inconvenience by using nodes with just two pointers, as we did for binary trees. Here, however, the left pointer will point to the first child of the vertex, while the right pointer will point to its next sibling. Accordingly, this representation is called the ***first child–next sibling representation***. Thus, all the siblings of a vertex are linked (via the nodes' right pointers) in a singly linked list, with the first element of the list pointed to by the left pointer of their parent. Figure 1.14a illustrates this representation for the tree in Figure 1.11b. It is not difficult to see that this representation effectively transforms an ordered tree into a binary tree said to be associated with the ordered tree. We get this representation by “rotating” the pointers about 45 degrees clockwise (see Figure 1.14b).

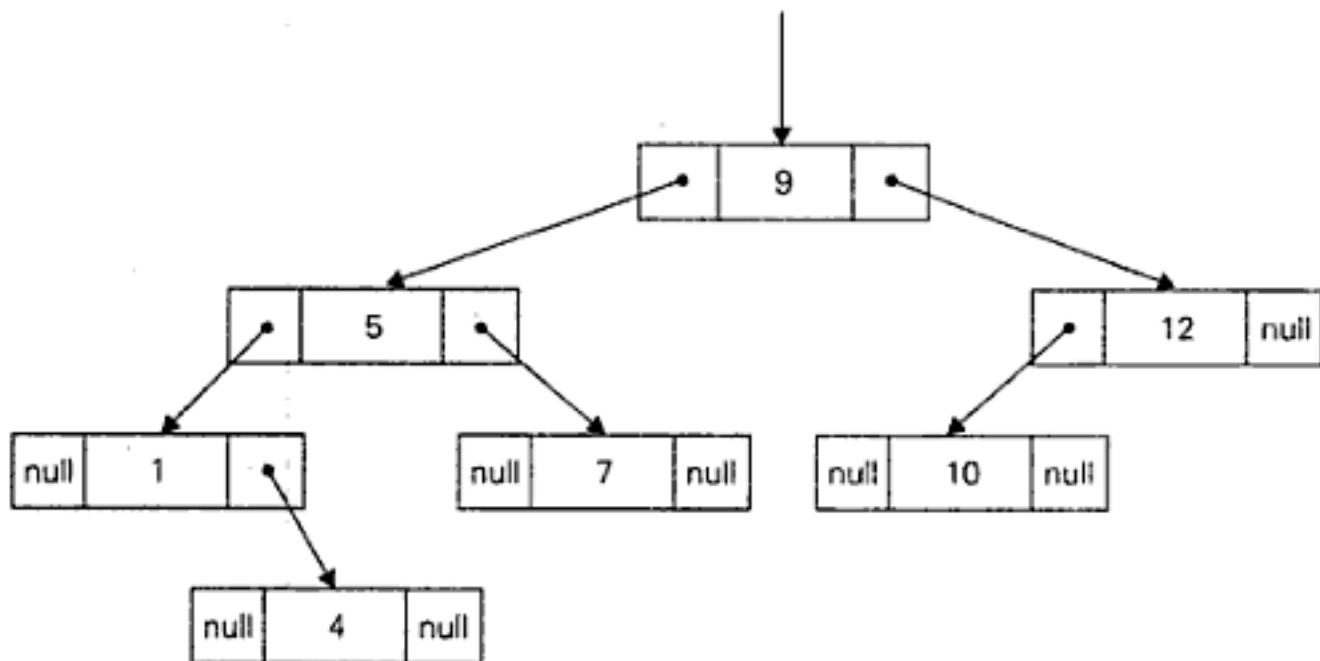


FIGURE 1.13 Standard implementation of the binary search tree in Figure 1.12b

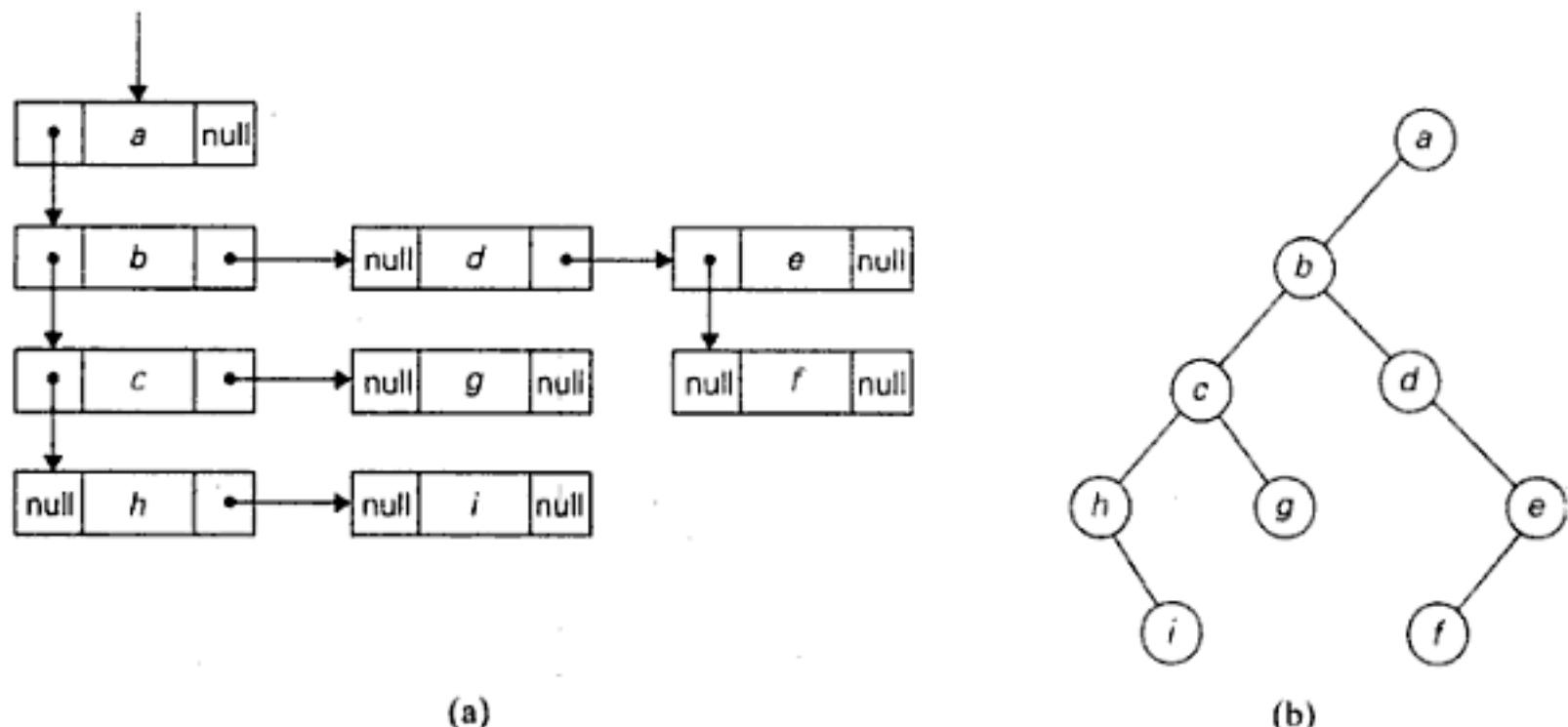


FIGURE 1.14 (a) First child–next sibling representation of the graph in Figure 1.11b.
(b) Its binary tree representation.

Sets and Dictionaries

The notion of a set plays a central role in mathematics. A **set** can be described as an unordered collection (possibly empty) of distinct items called **elements** of the set. A specific set is defined either by an explicit listing of its elements (e.g., $S = \{2, 3, 5, 7\}$) or by specifying a property that all the set's elements and only they must satisfy (e.g., $S = \{n : n \text{ is a prime number and } n < 10\}$). The most important set operations are checking membership of a given item in a given set (whether a given item is among the elements of the set), finding the union of two

sets (which set comprises all the elements that belong to either of the two sets or to both of them), and finding the intersection of two sets (which set comprises all the elements that belong to both sets).

Sets can be implemented in computer applications in two ways. The first considers only sets that are subsets of some large set U , called the *universal set*. If set U has n elements, then any subset S of U can be represented by a bit string of size n , called a *bit vector*, in which the i th element is 1 if and only if the i th element of U is included in set S . Thus, to continue with our example, if $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, then $S = \{2, 3, 5, 7\}$ will be represented by the bit string 011010100. This way of representing sets makes it possible to implement the standard set operations very fast but at the expense of potentially using a large amount of storage.

The second and more common way to represent a set for computing purposes is to use the list structure to indicate the set's elements. (Of course, this option, too, is feasible only for finite sets; fortunately, unlike mathematics, this is the kind of sets most computer applications need.) Note, however, the two principal points of distinction between sets and lists. First, a set cannot contain identical elements; a list can. This requirement for uniqueness is sometimes circumvented by the introduction of a *multiset* or a *bag*, an unordered collection of items that are not necessarily distinct. Second, a set is an unordered collection of items; therefore, changing the order of its elements does not change the set. A list, defined as an ordered collection of items, is exactly the opposite. This is an important theoretical distinction, but fortunately it is not important for many applications. It is also worth mentioning that if a set is represented by a list, depending on the application at hand, it might be worth maintaining the list in a sorted order.

In computing, the operations we need to perform for a set or a multiset most often are searching for a given item, adding a new item, and deleting an item from the collection. A data structure that implements these three operations is called the *dictionary*. Note the relationship between this data structure and the problem of searching mentioned in Section 1.3; obviously, we are dealing here with searching in a dynamic context. Consequently, an efficient implementation of a dictionary has to strike a compromise between the efficiency of searching and the efficiencies of the other two operations. There are quite a few ways a dictionary can be implemented. They range from an unsophisticated use of arrays (sorted or not) to much more sophisticated techniques such as hashing and balanced search trees, which we discuss later in the book.

A number of applications in computing require a dynamic partition of some n -element set into a collection of disjoint subsets. After being initialized as a collection of n one-element subsets, the collection is subjected to a sequence of intermixed union and search operations. This problem is called the *set union problem*. We discuss efficient algorithmic solutions to this problem in Section 9.2 in conjunction with one of its most important applications.

You may have noticed that in our review of basic data structures we almost always mentioned specific operations that are typically performed for the structure in question. This intimate relationship between data and operations has been recognized by computer scientists for a long time. It has led them in particular to the idea of an *abstract data type (ADT)*: a set of abstract objects representing data items with a collection of operations that can be performed on them. As

illustrations of this notion, reread, say, our definitions of priority queue and dictionary. Although abstract data types could be implemented in older procedural languages such as Pascal (see, e.g., [Aho83]), it is much more convenient to do so in object-oriented languages, such as C++ and Java, that support abstract data types by means of *classes*.

Exercises 1.4

1. How are matrices stored in memory? Describe briefly the two standard conventions that are followed along with their indexing formulae.

2. Compare the advantages and disadvantages when lists of n numbers are represented using

- a. an array.
- b. a linked list.

3. a. Show the stack after each operation of the following sequence that starts with the empty stack:

push(a), push(b), pop, push(c), push(d), pop

- b. Show the queue after each operation of the following sequence that starts with the empty queue:

enqueue(a), enqueue(b), dequeue, enqueue(c), enqueue(d), dequeue

4. a. Let A be the adjacency matrix of an undirected graph. Explain what property of the matrix indicates that

- i. the graph is complete.
- ii. the graph has a loop, i.e., an edge connecting a vertex to itself.
- iii. the graph has an isolated vertex, i.e., a vertex with no edges incident to it.

- b. Answer the same questions for the adjacency list representation.

5. Give a detailed description of an algorithm for transforming a free tree into a tree rooted at a given vertex of the free tree.

6. Prove the inequalities that bracket the height of a binary tree with n vertices:

$$\lfloor \log_2 n \rfloor \leq h \leq n - 1.$$

7. Indicate how the ADT priority queue can be implemented as

- a. an (unsorted) array.
- b. a sorted array.
- c. a binary search tree.

8. How would you implement a dictionary of a reasonably small size n if you knew that all its elements are distinct (e.g., names of 50 states of the United States)? Specify an implementation of each dictionary operation.
9. For each of the following applications, indicate the most appropriate data structures:
 - a. issuing bus tickets to passengers in the order they arrive at the counter.
 - b. to serve food for guests who pick up plates from the top, if the plates are stacked on the dining table for a dinner.
 - c. conversion of infix expression into postfix expression by a computer.
10. *Anagram checking* Design an algorithm for checking whether two given words are anagrams, i.e., whether one word can be obtained by permuting the letters of the other. (For example, the words *tea* and *eat* are anagrams.)



SUMMARY

- An *algorithm* is a sequence of nonambiguous instructions for solving a problem in a finite amount of time. An input to an algorithm specifies an *instance* of the problem the algorithm solves.
- Algorithms can be specified in a natural language or a pseudocode; they can also be implemented as computer programs.
- Among several ways to classify algorithms, the two principal alternatives are:
 - to group algorithms according to types of problems they solve;
 - to group algorithms according to underlying design techniques they are based upon.
- The important problem types are sorting, searching, string processing, graph problems, combinatorial problems, geometric problems, and numerical problems.
- Algorithm *design techniques* (or “strategies” or “paradigms”) are general approaches to solving problems algorithmically, applicable to a variety of problems from different areas of computing.
- Although designing an algorithm is undoubtedly a creative activity, one can identify a sequence of interrelated actions involved in such a process. They are summarized in Figure 1.2.
- A good algorithm is usually a result of repeated efforts and rework.
- The same problem can often be solved by several algorithms. For example, three algorithms were given for computing the greatest common divisor of two integers: *Euclid’s algorithm*, the consecutive integer checking algorithm, and the middle-school algorithm (enhanced by the *sieve of Eratosthenes* for generating a list of primes).

- Algorithms operate on data. This makes the issue of data structuring critical for efficient algorithmic problem solving. The most important elementary data structures are the *array* and the *linked list*. They are used for representing more abstract data structures such as the *list*, the *stack*, the *queue*, the *graph* (via its *adjacency matrix* or *adjacency lists*), the *binary tree*, and the *set*.
- An abstract collection of objects with several operations that can be performed on them is called an *abstract data type (ADT)*. The *list*, the *stack*, the *queue*, the *priority queue*, and the *dictionary* are important examples of abstract data types. Modern object-oriented languages support implementation of ADTs by means of classes.

Fundamentals of the Analysis of Algorithm Efficiency

I often say that when you can measure what you are speaking about and express it in numbers you know something about it; but when you cannot express it in numbers your knowledge is a meagre and unsatisfactory kind: it may be the beginning of knowledge but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be.

—Lord Kelvin (1824–1907)

Not everything that can be counted counts, and not everything that counts can be counted.

Albert Einstein (1879–1955)

This chapter is devoted to analysis of algorithms. The *American Heritage Dictionary* defines “analysis” as “the separation of an intellectual or substantial whole into its constituent parts for individual study.” Accordingly, each of the principal dimensions of an algorithm pointed out in Section 1.2 is both a legitimate and desirable subject of study. But the term “analysis of algorithms” is usually used in a narrower technical sense to mean an investigation of an algorithm’s efficiency with respect to two resources: running time and memory space. This emphasis on efficiency is easy to explain. First, unlike such dimensions as simplicity and generality, efficiency can be studied in precise quantitative terms. Second, one can argue—although this is hardly always the case, given the speed and memory of today’s computers—that the efficiency considerations are of primary importance from the practical point of view. In this chapter, we too limit the discussion to an algorithm’s efficiency.

We start with a general framework for analyzing algorithm efficiency in Section 2.1. This section is arguably the most important in the chapter; the

fundamental nature of the topic makes it also one of the most important sections in the entire book.

In Section 2.2, we introduce three notations: O (“big oh”), Ω (“big omega”), and Θ (“big theta”). Borrowed from mathematics, these notations have become the language for discussing an algorithm’s efficiency.

In Section 2.3, we show how the general framework outlined in Section 2.1 can be systematically applied to analyzing the efficiency of nonrecursive algorithms. The main tool of such an analysis is setting up a sum representing the algorithm’s running time and then simplifying the sum by using standard sum manipulation techniques.

In Section 2.4, we show how the general framework outlined in Section 2.1 can be systematically applied to analyzing the efficiency of recursive algorithms. Here, the main tool is not a sum but a special kind of equation called a recurrence relation. We explain how such recurrence relations can be set up and then introduce a method for solving them.

Although we illustrate the analysis framework and the methods of its applications by a variety of examples in the first four sections of this chapter, Section 2.5 is devoted to yet another example—that of the Fibonacci numbers. Introduced 800 years ago, this remarkable sequence appears in a variety of applications both within and outside computer science. A discussion of the Fibonacci sequence serves as a natural vehicle for introducing an important class of recurrence relations not solvable by the method of Section 2.4. We also discuss several algorithms for computing the Fibonacci numbers, mostly for the sake of a few general observations about the efficiency of algorithms and methods of analyzing them.

The methods of Sections 2.3 and 2.4 provide a powerful technique for analyzing the efficiency of many algorithms with mathematical clarity and precision, but these methods are far from being foolproof. The last two sections of the chapter deal with two approaches—empirical analysis and algorithm visualization—that complement the pure mathematical techniques of Sections 2.3 and 2.4. Much newer and, hence, less developed than their mathematical counterparts, these approaches promise to play an important role among the tools available for analysis of algorithm efficiency.

2.1 Analysis Framework

In this section, we outline a general framework for analyzing the efficiency of algorithms. To begin with, there are two kinds of efficiency: time efficiency and space efficiency. **Time efficiency** indicates how fast an algorithm in question runs; **space efficiency** deals with the extra space the algorithm requires. In the early days of electronic computing, both resources—time and space—were at a premium. Half a century of relentless technological innovations have improved the computer’s speed and memory size by many orders of magnitude. Now the amount of extra space required by an algorithm is typically not of as much concern, with the caveat that there is still, of course, a difference between the fast main memory, the slower secondary memory, and the cache. The time issue has not diminished quite to the

same extent, however. In addition, the research experience has shown that for most problems, we can achieve much more spectacular progress in speed than in space. Therefore, following a well-established tradition of algorithm textbooks, we primarily concentrate on time efficiency, but the analytical framework introduced here is applicable to analyzing space efficiency as well.

Measuring an Input's Size

Let us start with the obvious observation that almost all algorithms run longer on larger inputs. For example, it takes longer to sort larger arrays, multiply larger matrices, and so on. Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.¹ In most cases, selecting such a parameter is quite straightforward. For example, it will be the size of the list for problems of sorting, searching, finding the list's smallest element, and most other problems dealing with lists. For the problem of evaluating a polynomial $p(x) = a_n x^n + \dots + a_0$ of degree n , it will be the polynomial's degree or the number of its coefficients, which is larger by one than its degree. You will see from the discussion that such a minor difference is inconsequential for the efficiency analysis.

There are situations, of course, where the choice of a parameter indicating an input size does matter. One such example is computing the product of two n -by- n matrices. There are two natural measures of size for this problem. The first and more frequently used is the matrix order n . But the other natural contender is the total number of elements N in the matrices being multiplied. (The latter is also more general since it is applicable to matrices that are not necessarily square.) Since there is a simple formula relating these two measures, we can easily switch from one to the other, but the answer about an algorithm's efficiency will be qualitatively different depending on which of the two measures we use (see Problem 2 in Exercises 2.1).

The choice of an appropriate size metric can be influenced by operations of the algorithm in question. For example, how should we measure an input's size for a spell-checking algorithm? If the algorithm examines individual characters of its input, then we should measure the size by the number of characters; if it works by processing words, we should count their number in the input.

We should make a special note about measuring the size of inputs for algorithms involving properties of numbers (e.g., checking whether a given integer n is prime). For such algorithms, computer scientists prefer measuring size by the number b of bits in the n 's binary representation:

$$b = \lceil \log_2 n \rceil + 1. \quad (2.1)$$

This metric usually gives a better idea about the efficiency of algorithms in question.

1. Some algorithms require more than one parameter to indicate the size of their inputs (e.g., the number of vertices and the number of edges for algorithms on graphs represented by adjacency lists).

Units for Measuring Running Time

The next issue concerns units for measuring an algorithm's running time. Of course, we can simply use some standard unit of time measurement—a second, a millisecond, and so on—to measure the running time of a program implementing the algorithm. There are obvious drawbacks to such an approach, however: dependence on the speed of a particular computer, dependence on the quality of a program implementing the algorithm and of the compiler used in generating the machine code, and the difficulty of clocking the actual running time of the program. Since we are after a measure of an *algorithm*'s efficiency, we would like to have a metric that does not depend on these extraneous factors.

One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both excessively difficult and, as we shall see, usually unnecessary. The thing to do is to identify the most important operation of the algorithm, called the **basic operation**, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

As a rule, it is not difficult to identify the basic operation of an algorithm: it is usually the most time-consuming operation in the algorithm's innermost loop. For example, most sorting algorithms work by comparing elements (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key comparison. As another example, algorithms for matrix multiplication and polynomial evaluation require two arithmetic operations: multiplication and addition. On most computers, multiplication of two numbers takes longer than addition, making the former an unquestionable choice for the basic operation.²

Thus, the established framework for the analysis of an algorithm's time efficiency suggests measuring it by counting the number of times the algorithm's basic operation is executed on inputs of size n . We will find out how to compute such a count for nonrecursive and recursive algorithms in Sections 2.3 and 2.4, respectively.

Here is an important application. Let c_{op} be the execution time of an algorithm's basic operation on a particular computer, and let $C(n)$ be the number of times this operation needs to be executed for this algorithm. Then we can estimate the running time $T(n)$ of a program implementing this algorithm on that computer by the formula

$$T(n) \approx c_{op}C(n).$$

Of course, this formula should be used with caution. The count $C(n)$ does not contain any information about operations that are not basic, and, in fact, the count itself is often computed only approximately. Further, the constant c_{op} is also an approximation whose reliability is not always easy to assess. Still, unless n is extremely large or very small, the formula can give a reasonable estimate of the algorithm's running time. It also makes it possible to answer such questions as "How much faster would this algorithm run on a machine that is ten times faster

2. On some computers based on the so-called RISC architecture, it is not necessarily the case (see, for example, the timing data provided by Kernighan and Pike [Ker99], pp. 185–186).

than the one we have?" The answer is, obviously, ten times. Or, assuming that $C(n) = \frac{1}{2}n(n - 1)$, how much longer will the algorithm run if we double its input size? The answer is about four times longer. Indeed, for all but very small values of n ,

$$C(n) = \frac{1}{2}n(n - 1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

and therefore

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4.$$

Note that we were able to answer the last question without actually knowing the value of c_{op} : it was neatly cancelled out in the ratio. Also note that $\frac{1}{2}$, the multiplicative constant in the formula for the count $C(n)$, was also cancelled out. It is for these reasons that the efficiency analysis framework ignores multiplicative constants and concentrates on the count's **order of growth** to within a constant multiple for large-size inputs.

Orders of Growth

Why this emphasis on the count's order of growth for large input sizes? A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones. When we have to compute, for example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid's algorithm is compared to the other two algorithms discussed in Section 1.1 or even why we should care which of them is faster and by how much. It is only when we have to find the greatest common divisor of two large numbers that the difference in algorithm efficiencies becomes both clear and important. For large values of n , it is the function's order of growth that counts: just look at Table 2.1, which contains values of a few functions particularly important for analysis of algorithms.

The magnitude of the numbers in Table 2.1 has a profound significance for the analysis of algorithms. The function growing the slowest among these is the logarithmic function. It grows so slowly, in fact, that we should expect a program

TABLE 2.1. Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

implementing an algorithm with a logarithmic basic-operation count to run practically instantaneously on inputs of all realistic sizes. Also note that although specific values of such a count depend, of course, on the logarithm's base, the formula

$$\log_a n = \log_a b \log_b n$$

makes it possible to switch from one base to another, leaving the count logarithmic but with a new multiplicative constant. This is why we omit a logarithm's base and write simply $\log n$ in situations where we are interested just in a function's order of growth to within a multiplicative constant.

On the other end of the spectrum are the exponential function 2^n and the factorial function $n!$ Both these functions grow so fast that their values become astronomically large even for rather small values of n . (This is the reason why we did not include their values for $n > 10^2$ in Table 2.1.) For example, it would take about $4 \cdot 10^{10}$ years for a computer making one trillion (10^{12}) operations per second to execute 2^{100} operations. Though this is incomparably faster than it would have taken to execute $100!$ operations, it is still longer than 4.5 billion ($4.5 \cdot 10^9$) years—the estimated age of the planet Earth. There is a tremendous difference between the orders of growth of the functions 2^n and $n!$, yet both are often referred to as “exponential-growth functions” (or simply “exponential”) despite the fact that, strictly speaking, only the former should be referred to as such. The bottom line, which is important to remember, is this:

Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.

Another way to appreciate the qualitative difference among the orders of growth of the functions in Table 2.1 is to consider how they react to, say, a twofold increase in the value of their argument n . The function $\log_2 n$ increases in value by just 1 (because $\log_2 2n = \log_2 2 + \log_2 n = 1 + \log_2 n$); the linear function increases twofold; the “ n -log- n ” function $n \log_2 n$ increases slightly more than twofold; the quadratic function n^2 and cubic function n^3 increase fourfold and eightfold, respectively (because $(2n)^2 = 4n^2$ and $(2n)^3 = 8n^3$); the value of 2^n gets squared (because $2^{2n} = (2^n)^2$); and $n!$ increases much more than that (yes, even mathematics refuses to cooperate to give a neat answer for $n!$).

Worst-Case, Best-Case, and Average-Case Efficiencies

In the beginning of this section, we established that it is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algorithm's input. But there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input. Consider, as an example, sequential search. This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted. Here is the algorithm's pseudocode, in which, for simplicity, a list is implemented as an array. (It also assumes that the second condition $A[i] \neq K$

will not be checked if the first one, which checks that the array's index does not exceed its upper bound, fails.)

ALGORITHM *SequentialSearch*($A[0..n - 1]$, K)

```

//Searches for a given value in a given array by sequential search
//Input: An array  $A[0..n - 1]$  and a search key  $K$ 
//Output: The index of the first element of  $A$  that matches  $K$ 
//          or  $-1$  if there are no matching elements
i  $\leftarrow 0$ 
while  $i < n$  and  $A[i] \neq K$  do
    i  $\leftarrow i + 1$ 
if  $i < n$  return i
else return  $-1$ 
```

Clearly, the running time of this algorithm can be quite different for the same list size n . In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n : $C_{\text{worst}}(n) = n$.

The *worst-case efficiency* of an algorithm is its efficiency for the worst-case input of size n , which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size. The way to determine the worst-case efficiency of an algorithm is, in principle, quite straightforward: we analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count $C(n)$ among all possible inputs of size n and then compute this worst-case value $C_{\text{worst}}(n)$. (For sequential search, the answer was obvious. The methods for handling less trivial situations are explained in subsequent sections of this chapter.) Clearly, the worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size n , the running time will not exceed $C_{\text{worst}}(n)$, its running time on the worst-case inputs.

The *best-case efficiency* of an algorithm is its efficiency for the best-case input of size n , which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size. Accordingly, we can analyze the best-case efficiency as follows. First, we determine the kind of inputs for which the count $C(n)$ will be the smallest among all possible inputs of size n . (Note that the best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.) Then we ascertain the value of $C(n)$ on these most convenient inputs. For example, for sequential search, best-case inputs are lists of size n with their first elements equal to a search key; accordingly, $C_{\text{best}}(n) = 1$.

The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency. But it is not completely useless, either. Though we should not expect to get best-case inputs, we might be able to take advantage of the fact that for some algorithms a good best-case performance extends to some useful types of inputs close to being the best-case ones. For example, there is a sorting algorithm (insertion sort) for which the best-case inputs are already sorted arrays

on which the algorithm works very fast. Moreover, this good best-case efficiency deteriorates only slightly for almost sorted arrays. Thus, such an algorithm might well be the method of choice for applications dealing with almost sorted arrays. And, of course, if the best-case efficiency of an algorithm is unsatisfactory, we can immediately discard it without further analysis.

It should be clear from our discussion, however, that neither the worst-case analysis nor its best-case counterpart yields the necessary information about an algorithm's behavior on a "typical" or "random" input. This is the information that the *average-case efficiency* seeks to provide. To analyze the algorithm's average-case efficiency, we must make some assumptions about possible inputs of size n .

Let us consider again sequential search. The standard assumptions are that (a) the probability of a successful search is equal to p ($0 \leq p \leq 1$) and (b) the probability of the first match occurring in the i th position of the list is the same for every i . Under these assumptions—the validity of which is usually difficult to verify, their reasonableness notwithstanding—we can find the average number of key comparisons $C_{avg}(n)$ as follows. In the case of a successful search, the probability of the first match occurring in the i th position of the list is p/n for every i , and the number of comparisons made by the algorithm in such a situation is obviously i . In the case of an unsuccessful search, the number of comparisons is n with the probability of such a search being $(1 - p)$. Therefore,

$$\begin{aligned} C_{avg}(n) &= \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n} \right] + n \cdot (1 - p) \\ &= \frac{p}{n} [1 + 2 + \cdots + i + \cdots + n] + n(1 - p) \\ &= \frac{p}{n} \frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p). \end{aligned}$$

This general formula yields some quite reasonable answers. For example, if $p = 1$ (i.e., the search must be successful), the average number of key comparisons made by sequential search is $(n+1)/2$; i.e., the algorithm will inspect, on average, about half of the list's elements. If $p = 0$ (i.e., the search must be unsuccessful), the average number of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

As you can see from this very elementary example, investigation of the average-case efficiency is considerably more difficult than investigation of the worst-case and best-case efficiencies. The direct approach for doing this involves dividing all instances of size n into several classes so that for each instance of the class the number of times the algorithm's basic operation is executed is the same. (What were these classes for sequential search?) Then a probability distribution of inputs is obtained or assumed so that the expected value of the basic operation's count can be found.

The technical implementation of this plan is rarely easy, however, and probabilistic assumptions underlying it in each particular case are usually difficult to verify. Given our quest for simplicity, we will mostly quote known results

about average-case efficiency of algorithms under discussion. If you are interested in derivations of these results, consult such books as [Baa00], [Sed96], [KnuI], [KnuII], and [KnuIII].

Does one really need the average-case efficiency information? The answer is unequivocally yes: there are many important algorithms for which the average-case efficiency is much better than the overly pessimistic worst-case efficiency would lead us to believe. So, without the average-case analysis, computer scientists could have missed many important algorithms. Finally, it should be clear from the preceding discussion that the average-case efficiency cannot be obtained by taking the average of the worst-case and the best-case efficiencies. Even though this average does occasionally coincide with the average-case cost, it is not a legitimate way of performing the average-case analysis.

Yet another type of efficiency is called *amortized efficiency*. It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure. It turns out that in some situations a single operation can be expensive, but the total time for an entire sequence of n such operations is always significantly better than the worst-case efficiency of that single operation multiplied by n . So we can “amortize” the high cost of such a worst-case occurrence over the entire sequence in a manner similar to the way a business would amortize the cost of an expensive item over the years of the item’s productive life. This sophisticated approach was discovered by the American computer scientist Robert Tarjan, who used it, among other applications, in developing an interesting variation of the classic binary search tree (see [Tar87] for a quite readable nontechnical discussion and [Tar85] for a technical account). We will see an example of the usefulness of amortized efficiency in Section 9.2, when we consider algorithms for finding unions of disjoint sets.

Recapitulation of the Analysis Framework

Before we leave this section, let us summarize the main points of the framework outlined above.

- Both time and space efficiencies are measured as functions of the algorithm’s input size.
- Time efficiency is measured by counting the number of times the algorithm’s basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.
- The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies.
- The framework’s primary interest lies in the order of growth of the algorithm’s running time (extra memory units consumed) as its input size goes to infinity.

In the next section, we look at formal means to investigate orders of growth. In Sections 2.3 and 2.4, we discuss particular methods for investigating nonrecursive

and recursive algorithms, respectively. It is there that you will see how the analysis framework outlined here can be applied to investigating efficiency of specific algorithms. You will encounter many more examples throughout the rest of the book.

Exercises 2.1

- 1.** For each of the following algorithms, indicate (i) a natural size metric for its inputs; (ii) its basic operation; (iii) whether the basic operation count can be different for inputs of the same size:
 - a. computing the sum of n numbers
 - b. computing $n!$
 - c. finding the largest element in a list of n numbers
 - d. Euclid's algorithm
 - e. sieve of Eratosthenes
 - f. pen-and-pencil algorithm for multiplying two 1 -digit decimal integers
- 2.** Define time complexity and space complexity. Write an algorithm for adding ' n ' natural numbers and find the time and space required by that algorithm.
- 3.** What are the best, average, and worst case inputs for the algorithm of sequential search given in this chapter.
- 4.** 
 - a. *Glove selection* There are 22 gloves in a drawer: 5 pairs of red gloves, 4 pairs of yellow, and 2 pairs of green. You select the gloves in the dark and can check them only after a selection has been made. What is the smallest number of gloves you need to select to have at least one matching pair in the best case? in the worst case? (after [Mos01], #18)
 - b. *Missing socks* Imagine that after washing 5 distinct pairs of socks, you discover that two socks are missing. Of course, you would like to have the largest number of complete pairs remaining. Thus, you are left with 4 complete pairs in the best-case scenario and with 3 complete pairs in the worst case. Assuming that the probability of disappearance for each of the 10 socks is the same, find the probability of the best-case scenario; the probability of the worst-case scenario; the number of pairs you should expect in the average case. (after [Mos01], #48)
- 5.** 
 - a. Prove formula (2.1) for the number of bits in the binary representation of a positive decimal integer.
 - b. What would be the analogous formula for the number of decimal digits?
 - c. Explain why, within the accepted analysis framework, it does not matter whether we use binary or decimal digits in measuring n 's size.
- 6.** Suggest how any sorting algorithm can be augmented in a way to make the best-case count of its key comparisons equal to just $n - 1$ (n is a list's size, of course). Do you think it would be a worthwhile addition to any sorting algorithm?

7. Gaussian elimination, the classic algorithm for solving systems of n linear equations in n unknowns, requires about $\frac{1}{3}n^3$ multiplications, which is the algorithm's basic operation.
- How much longer should you expect Gaussian elimination to work on a system of 1000 equations versus a system of 500 equations?
 - You are considering buying a computer that is 1000 times faster than the one you currently have. By what factor will the faster computer increase the sizes of systems solvable in the same amount of time as on the old computer?
8. For each of the following functions, indicate how much the function's value will change if its argument is increased fourfold.
- $\log_2 n$
 - \sqrt{n}
 - n
 - n^2
 - n^3
 - 2^n
9. Compare the two functions 2^n and n^2 for various values of n . Determine when the second function will become the same, smaller, and larger than the first function.
10. *Invention of chess* According to a well-known legend, the game of chess was invented many centuries ago in northwestern India by a sage named Shashi. When he took his invention to his king, the king liked the game so much that he offered the inventor any reward he wanted. Sashi asked for some grain to be obtained as follows: just a single grain of wheat was to be placed on the first square of the chess board, two on the second, four on the third, eight on the fourth, and so on, until all 64 squares had been filled. What would the ultimate result of this algorithm have been?



2.2 Asymptotic Notations and Basic Efficiency Classes

As pointed out in the previous section, the efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency. To compare and rank such orders of growth, computer scientists use three notations: O (big oh), Ω (big omega), and Θ (big theta). First, we introduce these notations informally, and then, after several examples, formal definitions are given. In the following discussion, $t(n)$ and $g(n)$ can be any nonnegative functions defined on the set of natural numbers. In the context we are interested in, $t(n)$ will be an algorithm's running time (usually indicated by its basic operation count $C(n)$), and $g(n)$ will be some simple function to compare the count with.

Informal Introduction

Informally, $O(g(n))$ is the set of all functions with a smaller or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). Thus, to give

a few examples, the following assertions are all true:

$$n \in O(n^2), \quad 100n + 5 \in O(n^2), \quad \frac{1}{2}n(n - 1) \in O(n^2).$$

Indeed, the first two functions are linear and hence have a smaller order of growth than $g(n) = n^2$, while the last one is quadratic and hence has the same order of growth as n^2 . On the other hand,

$$n^3 \notin O(n^2), \quad 0.00001n^3 \notin O(n^2), \quad n^4 + n + 1 \notin O(n^2).$$

Indeed, the functions n^3 and $0.00001n^3$ are both cubic and hence have a higher order of growth than n^2 ; and so has the fourth-degree polynomial $n^4 + n + 1$.

The second notation, $\Omega(g(n))$, stands for the set of all functions with a larger or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). For example,

$$n^3 \in \Omega(n^2), \quad \frac{1}{2}n(n - 1) \in \Omega(n^2), \quad \text{but } 100n + 5 \notin \Omega(n^2).$$

Finally, $\Theta(g(n))$ is the set of all functions that have the same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). Thus, every quadratic function $an^2 + bn + c$ with $a > 0$ is in $\Theta(n^2)$, but so are, among infinitely many others, $n^2 + \sin n$ and $n^2 + \log n$. (Can you explain why?)

Hopefully, the preceding informal discussion has made you comfortable with the idea behind the three asymptotic notations. So now come the formal definitions.

O-notation

DEFINITION 1 A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n , i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \quad \text{for all } n \geq n_0.$$

The definition is illustrated in Figure 2.1 where, for the sake of visual clarity, n is extended to be a real number.

As an example, let us formally prove one of the assertions made in the introduction: $100n + 5 \in O(n^2)$. Indeed,

$$100n + 5 \leq 100n + n \quad (\text{for all } n \geq 5) = 101n \leq 101n^2.$$

Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5, respectively.

Note that the definition gives us a lot of freedom in choosing specific values for constants c and n_0 . For example, we could also reason that

$$100n + 5 \leq 100n + 5n \quad (\text{for all } n \geq 1) = 105n$$

to complete the proof with $c = 105$ and $n_0 = 1$.

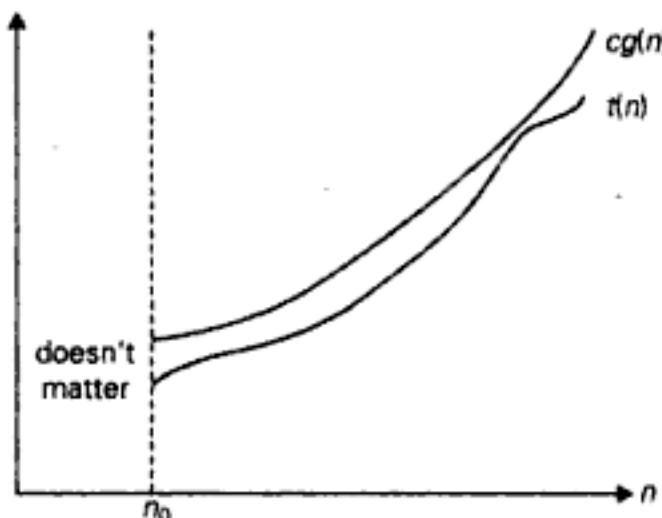


FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$

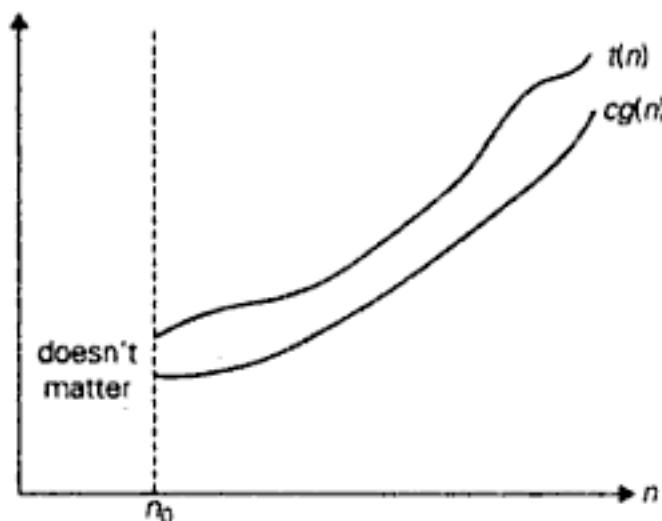


FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Ω -notation

DEFINITION 2 A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n , i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \quad \text{for all } n \geq n_0.$$

The definition is illustrated in Figure 2.2.

Here is an example of the formal proof that $n^3 \in \Omega(n^2)$:

$$n^3 \geq n^2 \quad \text{for all } n \geq 0,$$

i.e., we can select $c = 1$ and $n_0 = 0$.

Θ -notation

DEFINITION 3 A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of

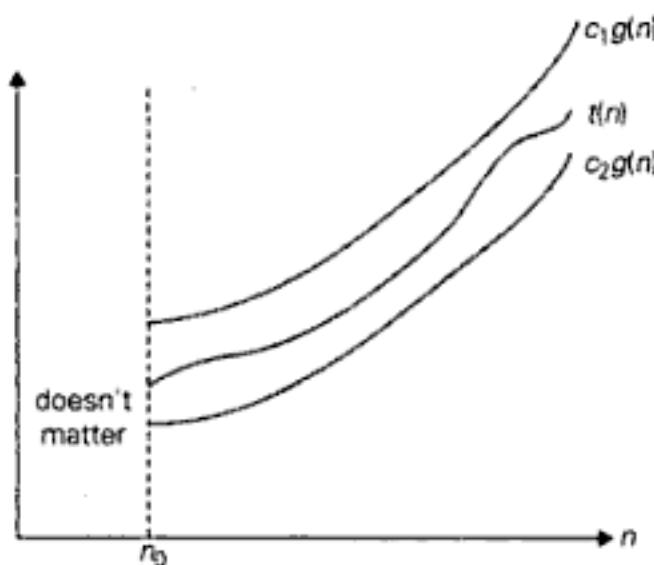


FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

$g(n)$ for all large n , i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2g(n) \leq t(n) \leq c_1g(n) \quad \text{for all } n \geq n_0.$$

The definition is illustrated in Figure 2.3.

For example, let us prove that $\frac{1}{2}n(n - 1) \in \Theta(n^2)$. First, we prove the right inequality (the upper bound):

$$\frac{1}{2}n(n - 1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \text{ for all } n \geq 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n - 1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n \frac{1}{2}n \text{ (for all } n \geq 2) = \frac{1}{4}n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

Useful Property Involving the Asymptotic Notations

Using the formal definitions of the asymptotic notations, we can prove their general properties (see Problem 7 in Exercises 2.2 for a few simple examples). The following property, in particular, is useful in analyzing algorithms that comprise two consecutively executed parts.

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

(The analogous assertions are true for the Ω and Θ notations as well.)

PROOF (As you will see, the proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1 , b_1 , a_2 , and b_2 : if $a_1 \leq b_1$ and $a_2 \leq b_2$, then $a_1 + a_2 \leq 2 \max\{b_1, b_2\}$.) Since $t_1(n) \in O(g_1(n))$, there exist some

positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \leq c_1 g_1(n) \quad \text{for all } n \geq n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \leq c_2 g_2(n) \quad \text{for all } n \geq n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \geq \max\{n_1, n_2\}$ so that we can use both inequalities. Adding the two inequalities above yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) = c_3 [g_1(n) + g_2(n)] \\ &\leq c_3 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2 \max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ■

So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a larger order of growth, i.e., its least efficient part:

$t_1(n) \in O(g_1(n))$
$t_2(n) \in O(g_2(n))$

 $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$

For example, we can check whether an array has identical elements by means of the following two-part algorithm: first, sort the array by applying some known sorting algorithm; second, scan the sorted array to check its consecutive elements for equality. If, for example, a sorting algorithm used in the first part makes no more than $\frac{1}{2}n(n - 1)$ comparisons (and hence is in $O(n^2)$) while the second part makes no more than $n - 1$ comparisons (and hence is in $O(n)$), the efficiency of the entire algorithm will be in $O(\max\{n^2, n\}) = O(n^2)$.

Using Limits for Comparing Orders of Growth

Though the formal definitions of O , Ω , and Θ are indispensable for proving their abstract properties, they are rarely used for comparing the orders of growth of two specific functions. A much more convenient method for doing so is based on computing the limit of the ratio of two functions in question. Three principal cases may arise:

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n) \\ c > 0 & \text{implies that } t(n) \text{ has the same order of growth as } g(n) \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

3. The fourth case, in which such a limit does not exist, rarely happens in the actual practice of analyzing algorithms. Still, this possibility makes the limit-based approach to comparing orders of growth less general than the one based on the definitions of O , Ω , and Θ .

Note that the first two cases mean that $t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, and the second case means that $t(n) \in \Theta(g(n))$.

The limit-based approach is often more convenient than the one based on the definitions because it can take advantage of the powerful calculus techniques developed for computing limits, such as L'Hôpital's rule

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{t'(n)}{g'(n)}$$

and Stirling's formula

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \text{ for large values of } n.$$

Here are three examples of using the limit-based approach to comparing orders of growth of two functions.

EXAMPLE 1 Compare the orders of growth of $\frac{1}{2}n(n - 1)$ and n^2 . (This is one of the examples we used at the beginning of this section to illustrate the definitions.)

$$\lim_{n \rightarrow \infty} \frac{\frac{1}{2}n(n - 1)}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \frac{n^2 - n}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right) = \frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n - 1) \in \Theta(n^2)$. ■

EXAMPLE 2 Compare the orders of growth of $\log_2 n$ and \sqrt{n} . (Unlike Example 1, the answer here is not immediately obvious.)

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n \rightarrow \infty} \frac{(\log_2 e) \frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2 \log_2 e \lim_{n \rightarrow \infty} \frac{\sqrt{n}}{n} = 0.$$

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than \sqrt{n} . (Since $\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = 0$, we can use the so-called *little-o notation*: $\log_2 n \in o(\sqrt{n})$. Unlike the big-oh, the little-o notation is rarely used in analysis of algorithms.) ■

EXAMPLE 3 Compare the orders of growth of $n!$ and 2^n . (We discussed this issue informally in the previous section.) Taking advantage of Stirling's formula, we get

$$\lim_{n \rightarrow \infty} \frac{n!}{2^n} = \lim_{n \rightarrow \infty} \frac{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n}{2^n} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \frac{n^n}{2^n e^n} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \left(\frac{n}{2e}\right)^n = \infty.$$

Thus, though 2^n grows very fast, $n!$ grows still faster. We can write symbolically that $n! \in \Omega(2^n)$; note, however, that while big-omega notation does not preclude

the possibility that $n!$ and 2^n have the same order of growth, the limit computed here certainly does. ■

Basic Efficiency Classes

Even though the efficiency analysis framework puts together all the functions whose orders of growth differ by a constant multiple, there are still infinitely many such classes. (For example, the exponential functions a^n have different orders of growth for different values of base a .) Therefore, it may come as a surprise that the time efficiencies of a large number of algorithms fall into only a few classes. These classes are listed in Table 2.2 in increasing order of their orders of growth, along with their names and a few comments.

TABLE 2.2. Basic asymptotic efficiency classes

Class	Name	Comments
1	<i>constant</i>	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	<i>logarithmic</i>	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 5.5). Note that a logarithmic algorithm cannot take into account all its input (or even a fixed fraction of it): any algorithm that does so will have at least linear running time.
n	<i>linear</i>	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	<i>"n-log-n"</i>	Many divide-and-conquer algorithms (see Chapter 4), including mergesort and quicksort in the average case, fall into this category.
n^2	<i>quadratic</i>	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on n -by- n matrices are standard examples.
n^3	<i>cubic</i>	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	<i>exponential</i>	Typical for algorithms that generate all subsets of an n -element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	<i>factorial</i>	Typical for algorithms that generate all permutations of an n -element set.

You could raise a concern that classifying algorithms by their asymptotic efficiency would be of little practical use since the values of multiplicative constants are usually left unspecified. This leaves open a possibility of an algorithm in a worse efficiency class running faster than an algorithm in a better efficiency class for inputs of realistic sizes. For example, if the running time of one algorithm is n^3 while the running time of the other is 10^6n^2 , the cubic algorithm will outperform the quadratic algorithm unless n exceeds 10^6 . A few such anomalies are indeed known. For example, there exist algorithms for matrix multiplication with a better asymptotic efficiency than the cubic efficiency of the definition-based algorithm (see Section 4.5). Because of their much larger multiplicative constants, however, the value of these more sophisticated algorithms is mostly theoretical.

Fortunately, multiplicative constants usually do not differ that drastically. As a rule, you should expect an algorithm from a better asymptotic efficiency class to outperform an algorithm from a worse class even for moderately sized inputs. This observation is especially true for an algorithm with a better than exponential running time versus an exponential (or worse) algorithm.

Exercises 2.2

1. Use the most appropriate notation among O , Θ and Ω to indicate the time efficiency class of binary search
 - a. in the worst case.
 - b. in the best case.
 - c. in the average case.
2. From the following equalities, indicate the ones that are incorrect.
 - a. $6n^2 - 8n = \Theta(n^2)$
 - b. $12n^2 + 8 = O(n)$
 - c. $3n^2 3^n + n \log n = \Theta(n^2 3^n)$
 - d. $3n^2 \log n = \Theta(n^2)$
3. For each of the following functions, indicate the class $\Theta(g(n))$ the function belongs to. (Use the simplest $g(n)$ possible in your answers.) Prove your assertions.
 - a. $(n^2 + 1)^{10}$
 - b. $\sqrt{10n^2 + 7n + 3}$
 - c. $2n \lg(n+2)^2 + (n+2)^2 \lg \frac{n}{2}$
 - d. $2^{n+1} + 3^{n-1}$
 - e. $\lfloor \log_2 n \rfloor$
4. a. Table 2.1 contains values of several functions that often arise in analysis of algorithms. These values certainly suggest that the functions

$$\log n, n, n \log n, n^2, n^3, 2^n, n!$$

are listed in increasing order of their order of growth. Do these values prove this fact with mathematical certainty?

- b. Prove that the functions are indeed listed in increasing order of their order of growth.
5. Arrange the following functions according to their order of decay (from the highest to the lowest):

$$(n+1)!2^{3n}, 2n^4 + 2n^3 + 4, n \log n, \log n, 6n, 8n^2.$$

6. a. Prove that every polynomial of degree k , $p(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_0$, with $a_k > 0$ belongs to $\Theta(n^k)$.
- b. Prove that exponential functions a^n have different orders of growth for different values of base $a > 0$.
7. Prove (by using the definitions of the notations involved) or disprove (by giving a specific counterexample) the following assertions.
- If $t(n) \in O(g(n))$, then $g(n) \in \Omega(t(n))$.
 - $\Theta(\alpha g(n)) = \Theta(g(n))$ where $\alpha > 0$.
 - $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.
 - For any two nonnegative functions $t(n)$ and $g(n)$ defined on the set of nonnegative integers, either $t(n) \in O(g(n))$, or $t(n) \in \Omega(g(n))$, or both.
8. Prove the section's theorem for
- Ω -notation.
 - Θ -notation.
9. We mentioned in this section that one can check whether all elements of an array are distinct by a two-part algorithm based on the array's presorting.
- If the presorting is done by an algorithm with the time efficiency in $\Theta(n \log n)$, what will be the time efficiency class of the entire algorithm?
 - If the sorting algorithm used for presorting needs an extra array of size n , what will be the space efficiency class of the entire algorithm?
10. *Door in a wall* You are facing a wall that stretches infinitely in both directions. There is a door in the wall, but you know neither how far away nor in which direction. You can see the door only when you are right next to it. Design an algorithm that enables you to reach the door by walking at most $O(n)$ steps where n is the (unknown to you) number of steps between your initial position and the door. [Par95], #652



2.3 Mathematical Analysis of Nonrecursive Algorithms

In this section, we systematically apply the general framework outlined in Section 2.1 to analyzing the time efficiency of nonrecursive algorithms. Let us start with a very simple example that demonstrates all the principal steps typically taken in analyzing such algorithms.

EXAMPLE 1 Consider the problem of finding the value of the largest element in a list of n numbers. For simplicity, we assume that the list is implemented as an array. The following is a pseudocode of a standard algorithm for solving the problem.

ALGORITHM MaxElement($A[0..n - 1]$)

```
//Determines the value of the largest element in a given array
//Input: An array  $A[0..n - 1]$  of real numbers
//Output: The value of the largest element in  $A$ 
maxval  $\leftarrow A[0]$ 
for  $i \leftarrow 1$  to  $n - 1$  do
    if  $A[i] > maxval$ 
        maxval  $\leftarrow A[i]$ 
return maxval
```

The obvious measure of an input's size here is the number of elements in the array, i.e., n . The operations that are going to be executed most often are in the algorithm's **for** loop. There are two operations in the loop's body: the comparison $A[i] > maxval$ and the assignment $maxval \leftarrow A[i]$. Which of these two operations should we consider basic? Since the comparison is executed on each repetition of the loop and the assignment is not, we should consider the comparison to be the algorithm's basic operation. (Note that the number of comparisons will be the same for all arrays of size n ; therefore, in terms of this metric, there is no need to distinguish among the worst, average, and best cases here.)

Let us denote $C(n)$ the number of times this comparison is executed and try to find a formula expressing it as a function of size n . The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and $n - 1$ (inclusively). Therefore, we get the following sum for $C(n)$:

$$C(n) = \sum_{i=1}^{n-1} 1.$$

This is an easy sum to compute because it is nothing else but 1 repeated $n - 1$ times. Thus,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

■

Here is a general plan to follow in analyzing nonrecursive algorithms.

General Plan for Analyzing Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in its innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property,

the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.

4. Set up a sum expressing the number of times the algorithm's basic operation is executed.⁴
5. Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.

Before proceeding with further examples, you may want to review Appendix A, which contains a list of summation formulas and rules that are often useful in analysis of algorithms. In particular, we use especially frequently two basic rules of sum manipulation

$$\sum_{i=l}^u ca_i = c \sum_{i=l}^u a_i \quad (\text{R1})$$

$$\sum_{i=l}^u (a_i \pm b_i) = \sum_{i=l}^u a_i \pm \sum_{i=l}^u b_i \quad (\text{R2})$$

and two summation formulas

$$\sum_{i=l}^u 1 = u - l + 1 \text{ where } l \leq u \text{ are some lower and upper integer limits} \quad (\text{S1})$$

$$\sum_{i=0}^n i = \sum_{i=1}^n i = 1 + 2 + \dots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \quad (\text{S2})$$

(Note that the formula $\sum_{i=1}^{n-1} 1 = n - 1$, which we used in Example 1, is a special case of formula (S1) for $l = 1$ and $u = n - 1$.)

EXAMPLE 2 Consider the *element uniqueness problem*: check whether all the elements in a given array are distinct. This problem can be solved by the following straightforward algorithm.

ALGORITHM UniqueElements($A[0..n - 1]$)

```

//Determines whether all the elements in a given array are distinct
//Input: An array  $A[0..n - 1]$ 
//Output: Returns "true" if all the elements in  $A$  are distinct
//         and "false" otherwise
for  $i \leftarrow 0$  to  $n - 2$  do
    for  $j \leftarrow i + 1$  to  $n - 1$  do
        if  $A[i] = A[j]$  return false
return true

```

-
4. Sometimes, an analysis of a nonrecursive algorithm requires setting up not a sum but a recurrence relation for the number of times its basic operation is executed. Using recurrence relations is much more typical for analyzing recursive algorithms (see Section 2.4).

The natural measure of the input's size here is again the number of elements in the array, i.e., n . Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation. Note, however, that the number of element comparisons will depend not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only.

By definition, the worst case input is an array for which the number of element comparisons $C_{\text{worst}}(n)$ is the largest among all arrays of size n . An inspection of the innermost loop reveals that there are two kinds of worst-case inputs (inputs for which the algorithm does not exit the loop prematurely): arrays with no equal elements and arrays in which the last two elements are the only pair of equal elements. For such inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop's variable j between its limits $i + 1$ and $n - 1$; and this is repeated for each value of the outer loop, i.e., for each value of the loop's variable i between its limits 0 and $n - 2$. Accordingly, we get

$$\begin{aligned} C_{\text{worst}}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\ &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\ &= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \end{aligned}$$

We also could have computed the sum $\sum_{i=0}^{n-2} (n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \dots + 1 \stackrel{(S2)}{=} \frac{(n-1)n}{2}.$$

Note that this result was perfectly predictable: in the worst case, the algorithm needs to compare all $n(n-1)/2$ distinct pairs of its n elements. ■

EXAMPLE 3 Given two n -by- n matrices A and B , find the time efficiency of the definition-based algorithm for computing their product $C = AB$. By definition, C is an n -by- n matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B :

$$\begin{array}{c} A \\ \text{row } i \\ \left[\begin{array}{cccc} & & & \\ \square & \square & \square & \square \\ & & & \end{array} \right] \end{array} * \begin{array}{c} B \\ \text{col. } j \\ \left[\begin{array}{c} \square \\ \square \\ \square \\ \vdots \\ \square \end{array} \right] \end{array} = \begin{array}{c} C \\ C[i,j] \\ \left[\begin{array}{c} \square \\ \vdots \\ \square \end{array} \right] \end{array}$$

where $C[i,j] = A[i,0]B[0,j] + \dots + A[i,k]B[k,j] + \dots + A[i,n-1]B[n-1,j]$ for every pair of indices $0 \leq i, j \leq n-1$.

```
ALGORITHM MatrixMultiplication(A[0..n - 1, 0..n - 1], B[0..n - 1, 0..n - 1])
  //Multiplies two n-by-n matrices by the definition-based algorithm
  //Input: Two n-by-n matrices A and B
  //Output: Matrix C = AB
  for i ← 0 to n - 1 do
    for j ← 0 to n - 1 do
      C[i, j] ← 0.0
      for k ← 0 to n - 1 do
        C[i, j] ← C[i, j] + A[i, k] * B[k, j]
  return C
```

We measure an input's size by matrix order *n*. In the algorithm's innermost loop are two arithmetical operations—multiplication and addition—that, in principle, can compete for designation as the algorithm's basic operation. We consider multiplication as the algorithm's basic operation (see Section 2.1). Note that for this algorithm, we do not have to choose between these two operations because on each repetition of the innermost loop, each of the two is executed exactly once. So by counting one we automatically count the other. Let us set up a sum for the total number of multiplications *M(n)* executed by the algorithm. (Since this count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.)

Obviously, there is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable *k* ranging from the lower bound 0 to the upper bound *n* - 1. Therefore, the number of multiplications made for every pair of specific values of variables *i* and *j* is

$$\sum_{k=0}^{n-1} 1,$$

and the total number of multiplications *M(n)* is expressed by the following triple sum:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

Now we can compute this sum by using formula (S1) and rule (R1) (see above). Starting with the innermost sum $\sum_{k=0}^{n-1} 1$, which is equal to *n* (why?), we get

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

(This example is simple enough so that we could get this result without all the summation machinations. How? The algorithm computes n^2 elements of the product matrix. Each of the product's elements is computed as the scalar (dot) product of an *n*-element row of the first matrix and an *n*-element column of the second

matrix, which takes n multiplications. So the total number of multiplications is $n \cdot n^2 = n^3$. It is this kind of reasoning we expected you to employ when answering this question in Problem 2 of Exercises 2.1.)

If we now want to estimate the running time of the algorithm on a particular machine, we can do it by the product

$$T(n) \approx c_m M(n) = c_m n^3,$$

where c_m is the time of one multiplication on the machine in question. We would get a more accurate estimate if we took into account the time spent on the additions, too:

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a)n^3,$$

where c_a is the time of one addition. Note that the estimates differ only by their multiplicative constants, not by their order of growth. ■

You should not have the erroneous impression that the plan outlined above always succeeds in analyzing a nonrecursive algorithm. An irregular change in a loop's variable, a sum too complicated to analyze, and the difficulties intrinsic to the average-case analysis are just some of the obstacles that can prove to be insurmountable. These caveats notwithstanding, the plan does work for many simple nonrecursive algorithms, as you will see throughout the subsequent chapters of the book.

As a last example, let us consider an algorithm in which the loop's variable changes in a different manner from that of the previous examples.

EXAMPLE 4 The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

ALGORITHM *Binary*(n)

```
//Input: A positive decimal integer  $n$ 
//Output: The number of binary digits in  $n$ 's binary representation
count ← 1
while  $n > 1$  do
    count ← count + 1
     $n \leftarrow \lfloor n/2 \rfloor$ 
return count
```

First, notice that the most frequently executed operation here is not inside the **while** loop but rather the comparison $n > 1$ that determines whether the loop's body will be executed. Since the number of times the comparison will be executed is larger than the number of repetitions of the loop's body by exactly 1, the choice is not that important.

A more significant feature of this example is the fact that the loop's variable takes on only a few values between its lower and upper limits; therefore we have to use an alternative way of computing the number of times the loop is executed. Since the value of n is about halved on each repetition of the loop, the answer

should be about $\log_2 n$. The exact formula for the number of times the comparison $n > 1$ will be executed is actually $\lfloor \log_2 n \rfloor + 1$ —the number of bits in the binary representation of n according to formula (2.1). We could also get this answer by applying the analysis technique based on recurrence relations; we discuss this technique in the next section because it is more pertinent to the analysis of recursive algorithms. ■

Exercises 2.3

- 1.** Compute the following sums.

- a. $1 + 3 + 5 + 7 + \dots + 999$
- b. $2 + 4 + 8 + 16 + \dots + 1024$
- c. $\sum_{i=3}^{n+1} 1$
- d. $\sum_{i=3}^{n+1} i$
- e. $\sum_{i=0}^{n-1} i(i+1)$
- f. $\sum_{j=1}^n 3^{j+1}$
- g. $\sum_{i=1}^n \sum_{j=1}^n ij$
- h. $\sum_{i=1}^n 1/i(i+1)$

- 2.** Find the order of growth of the following sums.

- a. $\sum_{i=0}^{n-1} (i^2 + 1)^2$
- b. $\sum_{i=2}^{n-1} \lg i^2$
- c. $\sum_{i=1}^n (i+1)2^{i-1}$
- d. $\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} (i+j)$

Use the $\Theta(g(n))$ notation with the simplest function $g(n)$ possible.

- 3.** The sample variance of n measurements x_1, \dots, x_n can be computed as

$$\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1} \text{ where } \bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

or

$$\frac{\sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2/n}{n-1}.$$

Find and compare the number of divisions, multiplications, and additions/subtractions (additions and subtractions are usually bunched together) that are required for computing the variance according to each of these formulas.

- 4.** Consider the following algorithm:

ALGORITHM *Sum* (n)

```
//Input: A nonnegative integer  $n$ 
S ← 0
for  $i \leftarrow 1$  to  $n$  do
    S ← S + i
return S
```

- a. What does this algorithm compute?
 - b. What is its basic operation?
 - c. How many times is the basic operation executed?
 - d. What is the efficiency class of this algorithm?
 - e. Suggest an improved algorithm and indicate its efficiency class. If you cannot do it, try to prove that it cannot be done.
5. Consider the following algorithm.

ALGORITHM Fibonacci (n)

```

//Input: A nonnegative integer  $n$ 
//Output: finding the  $n^{\text{th}}$  Fibonacci number
if  $n < 1$ 
    return  $n$ 
else
     $F1 \leftarrow 0, F2 \leftarrow 1;$ 
    for  $i \leftarrow 2$  to  $n$  do
        |
         $F \leftarrow F1 + F2$ 
         $F1 \leftarrow F2$ 
         $F2 \leftarrow F$ 
    |
return  $F$ 
```

Answer questions (c)–(d) of Problem 4 about this algorithm.

6. Write a program that implements the addition of two matrices A, B of order p -by- q in a language of your choice.
7. Improve the implementation of the matrix multiplication algorithm (see Example 3) by reducing the number of additions made by the algorithm. What effect will this change have on the algorithm's efficiency?
8. Determine the asymptotic order of growth for the total number of times all the doors are toggled in the locker doors puzzle (Problem 11 in Exercises 1.1).
9. Prove the formula

$$\sum_{i=1}^n i = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

either by mathematical induction or by following the insight of a 10-year old schoolboy named Karl Friedrich Gauss (1777–1855), who grew up to become one of the greatest mathematicians of all times.

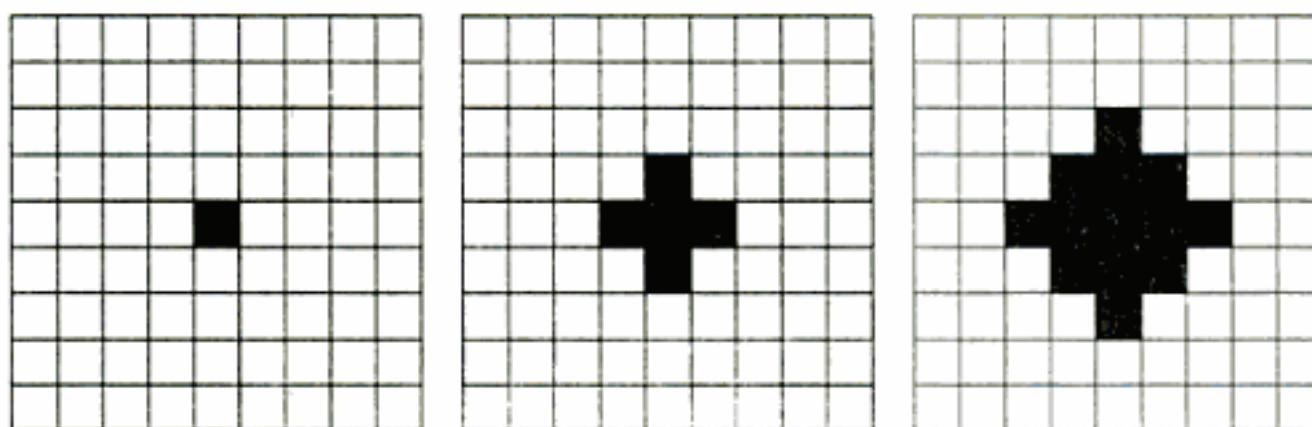
10. Consider the following version of an important algorithm that we will study later in the book.

ALGORITHM $GE(A[0..n-1, 0..n])$ //Input: An n -by- $n+1$ matrix $A[0..n-1, 0..n]$ of real numbers**for** $i \leftarrow 0$ **to** $n-2$ **do** **for** $j \leftarrow i+1$ **to** $n-1$ **do** **for** $k \leftarrow i$ **to** n **do** $A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] / A[i, i]$

- a. Find the time efficiency class of this algorithm.
 b. What glaring inefficiency does this pseudocode contain and how can it be eliminated to speed the algorithm up?



11. *von Neumann's neighborhood* How many one-by-one squares are generated by the algorithm that starts with a single square and on each of its n iterations adds new squares all round the outside? [Gar99], p. 88. (In the parlance of cellular automata theory, the answer is the number of cells in the von Newmann neighborhood of range n .) The results for $n = 0, 1$, and 2 are illustrated below.



2.4 Mathematical Analysis of Recursive Algorithms

In this section, we will see how to apply the general framework for analysis of algorithms to recursive algorithms. We start with an example often used to introduce novices to the idea of a recursive algorithm.

EXAMPLE 1 Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n . Since

$$n! = 1 \cdot \dots \cdot (n-1) \cdot n = (n-1)! \cdot n \quad \text{for } n \geq 1$$

and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm.

ALGORITHM $F(n)$

```
//Computes  $n!$  recursively
//Input: A nonnegative integer  $n$ 
//Output: The value of  $n!$ 
if  $n = 0$  return 1
else return  $F(n - 1) * n$ 
```

For simplicity, we consider n itself as an indicator of this algorithm's input size (rather than the number of bits in its binary expansion). The basic operation of the algorithm is multiplication,⁵ whose number of executions we denote $M(n)$. Since the function $F(n)$ is computed according to the formula

$$F(n) = F(n - 1) \cdot n \quad \text{for } n > 0,$$

the number of multiplications $M(n)$ needed to compute it must satisfy the equality

$$M(n) = M(n - 1) + \begin{array}{c} 1 \\ \text{to compute} \\ F(n - 1) \end{array} \quad \begin{array}{c} \text{to multiply} \\ F(n - 1) \text{ by } n \end{array} \quad \text{for } n > 0.$$

Indeed, $M(n - 1)$ multiplications are spent to compute $F(n - 1)$, and one more multiplication is needed to multiply the result by n .

The last equation defines the sequence $M(n)$ that we need to find. This equation defines $M(n)$ not explicitly, i.e., as a function of n , but implicitly as a function of its value at another point, namely $n - 1$. Such equations are called **recurrence relations** or, for brevity, **recurrences**. Recurrence relations play an important role not only in analysis of algorithms but also in some areas of applied mathematics. They are usually studied in detail in courses on discrete mathematics or discrete structures; a very brief tutorial on them is provided in Appendix B. Our goal now is to solve the recurrence relation $M(n) = M(n - 1) + 1$, i.e., to find an explicit formula for $M(n)$ in terms of n only.

Note, however, that there is not one but infinitely many sequences that satisfy this recurrence. (Can you give examples of, say, two of them?) To determine a solution uniquely, we need an **initial condition** that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if $n = 0$ return 1.

This tells us two things. First, since the calls stop when $n = 0$, the smallest value of n for which this algorithm is executed and hence $M(n)$ defined is 0. Second, by inspecting the pseudocode's exiting line, we can see that when $n = 0$, the algorithm performs no multiplications. Thus, the initial condition we are after is

5. Alternatively, we could count the number of times the comparison $n = 0$ is executed, which is the same as counting the total number of calls made by the algorithm (see Problem 2 in Exercises 2.4).

$$M(0) = 0.$$

↑ ↑

the calls stop when $n = 0$ no multiplications when $n = 0$

Thus, we succeed in setting up the recurrence relation and initial condition for the algorithm's number of multiplications $M(n)$:

$$\begin{aligned} M(n) &= M(n - 1) + 1 \quad \text{for } n > 0, \\ M(0) &= 0. \end{aligned} \tag{2.2}$$

Before we embark on a discussion of how to solve this recurrence, let us pause to reiterate an important point. We are dealing here with two recursively defined functions. The first is the factorial function $F(n)$ itself; it is defined by the recurrence

$$\begin{aligned} F(n) &= F(n - 1) \cdot n \quad \text{for every } n > 0, \\ F(0) &= 1. \end{aligned}$$

The second is the number of multiplications $M(n)$ needed to compute $F(n)$ by the recursive algorithm whose pseudocode was given at the beginning of the section. As we just showed, $M(n)$ is defined by recurrence (2.2). And it is recurrence (2.2) that we need to solve now.

Though it is not difficult to "guess" the solution (what sequence starts with 0 when $n = 0$ and increases by 1 at each step?), it will be more useful to arrive at it in a systematic fashion. Among several techniques available for solving recurrence relations, we use what can be called the ***method of backward substitutions***. The method's idea (and the reason for the name) is immediately clear from the way it applies to solving our particular recurrence:

$$\begin{aligned} M(n) &= M(n - 1) + 1 && \text{substitute } M(n - 1) = M(n - 2) + 1 \\ &= [M(n - 2) + 1] + 1 = M(n - 2) + 2 && \text{substitute } M(n - 2) = M(n - 3) + 1 \\ &= [M(n - 3) + 1] + 2 = M(n - 3) + 3. \end{aligned}$$

After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line (what would it be?) but also a general formula for the pattern: $M(n) = M(n - i) + i$. Strictly speaking, the correctness of this formula should be proved by mathematical induction, but it is easier to get the solution as follows and then verify its correctness.

What remains to be done is to take advantage of the initial condition given. Since it is specified for $n = 0$, we have to substitute $i = n$ in the pattern's formula to get the ultimate result of our backward substitutions:

$$M(n) = M(n - 1) + 1 = \dots = M(n - i) + i = \dots = M(n - n) + n = n.$$

You should not be disappointed after exerting so much effort to get this "obvious" answer. The benefits of the method illustrated in this simple example will

become clear very soon, when we have to solve more difficult recurrences. Also note that the simple iterative algorithm that accumulates the product of n consecutive integers requires the same number of multiplications, and it does so without the overhead of time and space used for maintaining the recursion's stack.

The issue of time efficiency is actually not that important for the problem of computing $n!$, however. As we saw in Section 2.1, the function's values get so large so fast that we can realistically compute its values only for very small n 's. Again, we use this example just as a simple and convenient vehicle to introduce the standard approach to analyzing recursive algorithms. ■

Generalizing our experience with investigating the recursive algorithm for computing $n!$, we can now outline a general plan for investigating recursive algorithms.

General Plan for Analyzing Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or at least ascertain the order of growth of its solution.

EXAMPLE 2 As our next example, we consider another educational workhorse of recursive algorithms: the *Tower of Hanoi* puzzle. In this puzzle, we (or mythical monks, if you do not like to move disks) have n disks of different sizes and three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.

The problem has an elegant recursive solution that is illustrated in Figure 2.4. To move $n > 1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary), we first move recursively $n - 1$ disks from peg 1 to peg 2 (with peg 3 as auxiliary), then move the largest disk directly from peg 1 to peg 3, and, finally, move recursively $n - 1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary). Of course, if $n = 1$, we can simply move the single disk directly from the source peg to the destination peg.

Let us apply the general plan to the Tower of Hanoi problem. The number of disks n is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. Clearly, the number of moves $M(n)$ depends on n only, and we get the following recurrence equation for it:

$$M(n) = M(n - 1) + 1 + M(n - 1) \quad \text{for } n > 1.$$

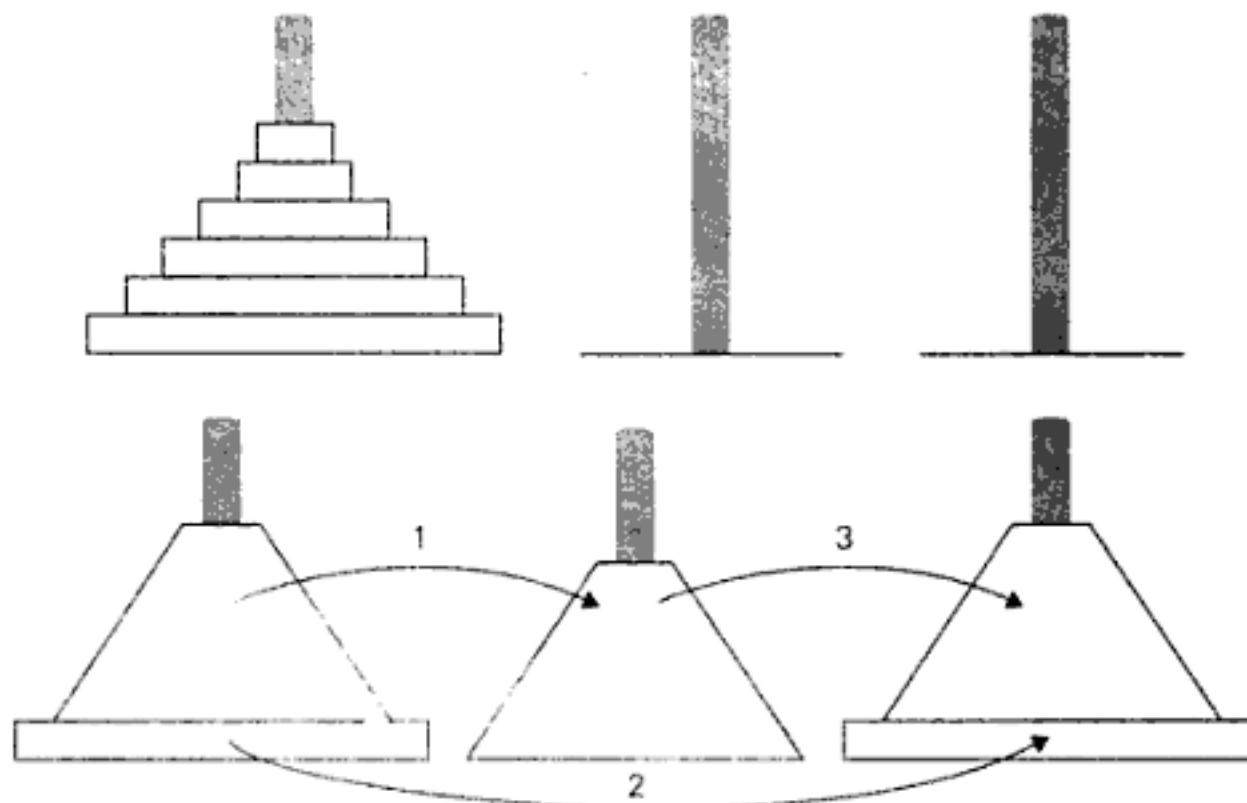


FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

$$M(n) = 2M(n - 1) + 1 \quad \text{for } n > 1, \quad (2.3)$$

$$M(1) = 1.$$

We solve this recurrence by the same method of backward substitutions:

$$\begin{aligned} M(n) &= 2M(n - 1) + 1 && \text{sub. } M(n - 1) = 2M(n - 2) + 1 \\ &= 2[2M(n - 2) + 1] + 1 = 2^2M(n - 2) + 2 + 1 && \text{sub. } M(n - 2) = 2M(n - 3) + 1 \\ &= 2^2[2M(n - 3) + 1] + 2 + 1 = 2^3M(n - 3) + 2^2 + 2 + 1. \end{aligned}$$

The pattern of the first three sums on the left suggests that the next one will be $2^4M(n - 4) + 2^3 + 2^2 + 2 + 1$ and, generally, after i substitutions, we get

$$M(n) = 2^iM(n - i) + 2^{i-1} + 2^{i-2} + \dots + 2 + 1 = 2^iM(n - i) + 2^i - 1.$$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence (2.3):

$$\begin{aligned} M(n) &= 2^{n-1}M(n - (n - 1)) + 2^{n-1} - 1 \\ &= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1. \end{aligned}$$

Thus, we have an exponential algorithm, which will run for an unimaginably long time even for moderate values of n (see Problem 5 in Exercises 2.4). This is

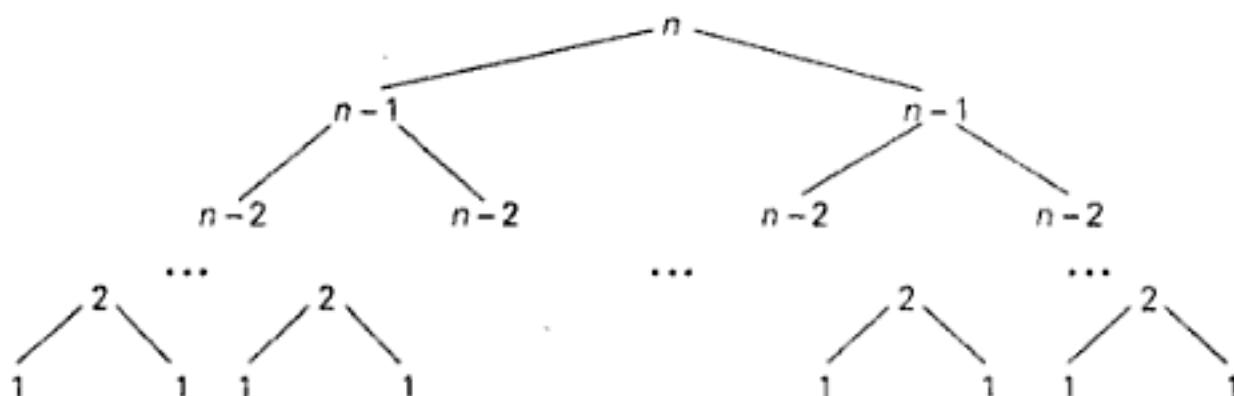


FIGURE 2.5 Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle

not due to the fact that this particular algorithm is poor; in fact, it is not difficult to prove that this is the most efficient algorithm possible for this problem. It is the problem's intrinsic difficulty that makes it so computationally hard. Still, this example makes an important general point:

One should be careful with recursive algorithms because their succinctness may mask their inefficiency.

When a recursive algorithm makes more than a single call to itself, it is useful for analysis purposes to construct a tree of its recursive calls. In this tree, nodes correspond to recursive calls, and we can label them with the value of the parameter (or, more generally, parameters) of the calls. For the Tower of Hanoi example, the tree is given in Figure 2.5. By counting the number of nodes in the tree, we can get the total number of calls made by the Tower of Hanoi algorithm:

$$C(n) = \sum_{l=0}^{n-1} 2^l \quad (\text{where } l \text{ is the level in the tree above}) = 2^n - 1.$$

The number agrees, as it should, with the move count obtained earlier. ■

EXAMPLE 3 As our next example, we investigate a recursive version of the algorithm discussed at the end of Section 2.3.

ALGORITHM *BinRec(n)*

```

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if n = 1 return 1
else return BinRec( $\lfloor n/2 \rfloor$ ) + 1

```

Let us set up a recurrence and an initial condition for the number of additions $A(n)$ made by the algorithm. The number of additions made in computing $\text{BinRec}(\lfloor n/2 \rfloor)$ is $A(\lfloor n/2 \rfloor)$, plus one more addition is made by the algorithm to

increase the returned value by 1. This leads to the recurrence

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1. \quad (2.4)$$

Since the recursive calls end when n is equal to 1 and there are no additions made then, the initial condition is

$$A(1) = 0.$$

The presence of $\lfloor n/2 \rfloor$ in the function's argument makes the method of backward substitutions stumble on values of n that are not powers of 2. Therefore, the standard approach to solving such a recurrence is to solve it only for $n = 2^k$ and then take advantage of the theorem called the *smoothness rule* (see Appendix B), which claims that under very broad assumptions the order of growth observed for $n = 2^k$ gives a correct answer about the order of growth for all values of n . (Alternatively, after obtaining a solution for powers of 2, we can sometimes fine-tune this solution to get a formula valid for an arbitrary n .) So let us apply this recipe to our recurrence, which for $n = 2^k$ takes the form

$$\begin{aligned} A(2^k) &= A(2^{k-1}) + 1 && \text{for } k > 0, \\ A(2^0) &= 0. \end{aligned}$$

Now backward substitutions encounter no problems:

$$\begin{aligned} A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\ &= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\ &= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots \\ &\quad \dots && \\ &= A(2^{k-i}) + i && \\ &\quad \dots && \\ &= A(2^{k-k}) + k. && \end{aligned}$$

Thus, we end up with

$$A(2^k) = A(1) + k = k$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$

In fact, we can prove (Problem 6 in Exercises 2.4) that the exact solution for an arbitrary value of n is given by just a slightly more refined formula $A(n) = \lfloor \log_2 n \rfloor$. ■

This section provides an introduction to analysis of recursive algorithms. These techniques will be used throughout the book and expanded further as

necessary. In the next section, we discuss the Fibonacci numbers; their analysis involves more difficult recurrence relations to be solved by a method different from backward substitutions.

Exercises 2.4

- Solve the following recurrence relations.
 - $x(n) = x(n - 1) + 5$ for $n > 1$, $x(1) = 0$
 - $x(n) = 3x(n - 1)$ for $n > 1$, $x(1) = 4$
 - $x(n) = x(n - 1) + n$ for $n > 0$, $x(0) = 0$
 - $x(n) = x(n/2) + n$ for $n > 1$, $x(1) = 1$ (solve for $n = 2^k$)
 - $x(n) = x(n/3) + 1$ for $n > 1$, $x(1) = 1$ (solve for $n = 3^k$)
- Set up and solve a recurrence relation for the number of calls made by $F(n)$, the recursive algorithm for computing $n!$.
- Consider the following recursive algorithm for computing the sum of n numbers.

ALGORITHM *Add(a, n)*

```
//Input: A positive integer  $n$  and an array  $a[n]$  of  $n$  numbers
//Output: The sum of the  $n$  given numbers
if  $n < 0$  return 0
else return Add ( $a, n - 1$ ) +  $a[n]$ 
```

- Setup and solve a recurrence relation for the number of times the algorithm's basic operation is executed.
- How does this algorithm compare with the straightforward nonrecursive algorithm for computing the sum?
- Consider the following recursive algorithm.

ALGORITHM *Q(n)*

```
//Input: A positive integer  $n$ 
if  $n = 1$  return 1
else return Q( $n - 1$ ) +  $2 * n - 1$ 
```

- Set up a recurrence relation for this function's values and solve it to determine what this algorithm computes.
- Set up a recurrence relation for the number of multiplications made by this algorithm and solve it.
- Set up a recurrence relation for the number of additions/subtractions made by this algorithm and solve it.
- a.** *Tower of Hanoi* In the original version of the Tower of Hanoi puzzle, as it was published by Edouard Lucas, a French mathematician, in the 1890s,



the world will end after 64 disks have been moved from a mystical Tower of Brahma. Estimate the number of years it will take if monks could move one disk per minute. (Assume that monks do not eat, sleep, or die.)

- b. How many moves are made by the i th largest disk ($1 \leq i \leq n$) in this algorithm?
- c. Design a nonrecursive algorithm for the Tower of Hanoi puzzle.
6. a. Prove that the exact number of additions made by the recursive algorithm $\text{BinRec}(n)$ for an arbitrary positive decimal integer n is $\lfloor \log_2 n \rfloor$.
b. Set up a recurrence relation for the number of additions made by the nonrecursive version of this algorithm (see Section 2.3, Example 4) and solve it.
7. a. Design a recursive algorithm for computing 2^n for any nonnegative integer n that is based on the formula: $2^n = 2^{n-1} + 2^{n-1}$.
b. Set up a recurrence relation for the number of additions made by the algorithm and solve it.
c. Draw a tree of recursive calls for this algorithm and count the number of calls made by the algorithm.
d. Is it a good algorithm for solving this problem?
8. Consider the following recursive algorithm.

ALGORITHM $\text{Min1}(A[0..n - 1])$
//Input: An array $A[0..n - 1]$ of real numbers
if $n = 1$ **return** $A[0]$
else $\text{temp} \leftarrow \text{Min1}(A[0..n - 2])$
 if $\text{temp} \leq A[n - 1]$ **return** temp
 else **return** $A[n - 1]$

- a. What does this algorithm compute?
- b. Set up a recurrence relation for the algorithm's basic operation count and solve it.
9. Consider another algorithm for solving the problem of Exercise 8, which recursively divides an array into two halves: call $\text{Min2}(A[0..n - 1])$ where

ALGORITHM $\text{Min2}(A[l..r])$
if $l = r$ **return** $A[l]$
else $\text{temp1} \leftarrow \text{Min2}(A[l..(l+r)/2])$
 $\text{temp2} \leftarrow \text{Min2}(A[(l+r)/2+1..r])$
 if $\text{temp1} \leq \text{temp2}$ **return** temp1
 else **return** temp2

- a. Set up a recurrence relation for the algorithm's basic operation count and solve it.

- b.** Which of the algorithms *Min1* or *Min2* is faster? Can you suggest an algorithm for the problem they solve that would be more efficient than both of them?
- 10.** The determinant of an n -by- n matrix

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & & a_{2n} \\ \vdots & & \\ a_{n1} & & a_{nn} \end{bmatrix},$$

denoted $\det A$, can be defined as a_{11} for $n = 1$ and, for $n > 1$, by the recursive formula

$$\det A = \sum_{j=1}^n s_j a_{1j} \det A_j,$$

where s_j is $+1$ if j is odd and -1 if j is even, a_{1j} is the element in row 1 and column j , and A_j is the $(n - 1)$ -by- $(n - 1)$ matrix obtained from matrix A by deleting its row 1 and column j .



- a.** Set up a recurrence relation for the number of multiplications made by the algorithm implementing this recursive definition.
- b.** Without solving the recurrence, what can you say about the solution's order of growth as compared to $n!$?
- 11.** *von Newmann's neighborhood revisited* Find the number of cells in the von Newmann neighborhood of range n (see Problem 11 in Exercises 2.3) by setting up and solving a recurrence relation.

2.5 Example: Fibonacci Numbers

In this section, we consider the *Fibonacci numbers*, a famous sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots \quad (2.5)$$

that can be defined by the simple recurrence

$$F(n) = F(n - 1) + F(n - 2) \quad \text{for } n > 1 \quad (2.6)$$

and two initial conditions

$$F(0) = 0, F(1) = 1. \quad (2.7)$$

The Fibonacci numbers were introduced by Leonardo Fibonacci in 1202 as a solution to a problem about the size of a rabbit population. Many more examples of Fibonacci-like numbers have since been discovered in the natural world, and they have even been used in predicting the prices of stocks and commodities.

There are some interesting applications of the Fibonacci numbers in computer science as well. For example, worst-case inputs for Euclid's algorithm happen to be consecutive elements of the Fibonacci sequence. Our discussion goals are quite limited here, however. First, we find an explicit formula for the n th Fibonacci number $F(n)$, and then we briefly discuss algorithms for computing it.

Explicit Formula for the n th Fibonacci Number

If we try to apply the method of backward substitutions to solve recurrence (2.6), we will fail to get an easily discernible pattern. Instead, let us take advantage of a theorem that describes solutions to a *homogeneous second-order linear recurrence with constant coefficients*

$$ax(n) + bx(n - 1) + cx(n - 2) = 0, \quad (2.8)$$

where a , b , and c are some fixed real numbers ($a \neq 0$) called the coefficients of the recurrence and $x(n)$ is the generic term of an unknown sequence to be found. According to this theorem—see Theorem 1 in Appendix B—recurrence (2.8) has an infinite number of solutions that can be obtained by one of the three formulas. Which of the three formulas applies to a particular case depends on the number of real roots of the quadratic equation with the same coefficients as recurrence (2.8):

$$ar^2 + br + c = 0. \quad (2.9)$$

Quite logically, equation (2.9) is called the *characteristic equation* for recurrence (2.8).

Let us apply this theorem to the case of the Fibonacci numbers. To do so, recurrence (2.6) needs to be rewritten as

$$F(n) - F(n - 1) - F(n - 2) = 0. \quad (2.10)$$

Its characteristic equation is

$$r^2 - r - 1 = 0,$$

with the roots

$$r_{1,2} = \frac{1 \pm \sqrt{1 - 4(-1)}}{2} = \frac{1 \pm \sqrt{5}}{2}.$$

Since this characteristic equation has two distinct real roots, we have to use the formula indicated in Case 1 of Theorem 1:

$$F(n) = \alpha \left(\frac{1 + \sqrt{5}}{2} \right)^n + \beta \left(\frac{1 - \sqrt{5}}{2} \right)^n.$$

So far, we have ignored initial conditions (2.7). Now we take advantage of them to find specific values of parameters α and β . We do this by substituting 0 and 1—the values of n for which the initial conditions are given—into the last

formula and equating the results to 0 and 1 (the values of $F(0)$ and $F(1)$ according to (2.7)), respectively:

$$F(0) = \alpha \left(\frac{1+\sqrt{5}}{2}\right)^0 + \beta \left(\frac{1-\sqrt{5}}{2}\right)^0 = 0$$

$$F(1) = \alpha \left(\frac{1+\sqrt{5}}{2}\right)^1 + \beta \left(\frac{1-\sqrt{5}}{2}\right)^1 = 1.$$

After some standard algebraic simplifications, we get the following system of two linear equations in two unknowns α and β :

$$\begin{array}{rcl} \alpha & + & \beta = 0 \\ \left(\frac{1+\sqrt{5}}{2}\right)\alpha & + & \left(\frac{1-\sqrt{5}}{2}\right)\beta = 1. \end{array}$$

Solving the system (e.g., by substituting $\beta = -\alpha$ into the second equation and solving the equation obtained for α), we get the values $\alpha = 1/\sqrt{5}$ and $\beta = -1/\sqrt{5}$ for the unknowns. Thus,

$$F(n) = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2}\right)^n = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n), \quad (2.11)$$

where $\phi = (1 + \sqrt{5})/2 \approx 1.61803$ and $\hat{\phi} = -1/\phi \approx -0.61803$.⁶ It is hard to believe that formula (2.11), which includes arbitrary integer powers of irrational numbers, yields nothing else but all the elements of Fibonacci sequence (2.5), but it does!

One of the benefits of formula (2.11) is that it immediately implies that $F(n)$ grows exponentially (remember Fibonacci's rabbits?), i.e., $F(n) \in \Theta(\phi^n)$. This follows from the observation that $\hat{\phi}$ is between -1 and 0 , and, hence, $\hat{\phi}^n$ gets infinitely small as n goes to infinity. In fact, one can prove that the impact of the second term $\frac{1}{\sqrt{5}}\hat{\phi}^n$ on the value of $F(n)$ can be obtained by rounding off the value of the first term to the nearest integer. In other words, for every nonnegative integer n ,

$$F(n) = \frac{1}{\sqrt{5}}\phi^n \text{ rounded to the nearest integer.} \quad (2.12)$$

Algorithms for Computing Fibonacci Numbers

Though the Fibonacci numbers have many fascinating properties, we limit our discussion to a few remarks about algorithms for computing them. Actually, the sequence grows so fast that it is the size of the numbers rather than a time-efficient method for computing them that should be of primary concern here. Also, for the sake of simplicity, we consider such operations as additions and multiplications at

6. Constant ϕ is known as the **golden ratio**. Since antiquity, it has been considered the most pleasing ratio of a rectangle's two sides to the human eye and might have been consciously used by ancient architects and sculptors.

unit cost in the algorithms that follow. Since the Fibonacci numbers grow infinitely large (and grow rapidly), a more detailed analysis than the one offered here is warranted. These caveats notwithstanding, the algorithms we outline and their analysis are useful examples for a student of the design and analysis of algorithms.

To begin with, we can use recurrence (2.6) and initial condition (2.7) for the obvious recursive algorithm for computing $F(n)$.

ALGORITHM $F(n)$

```
//Computes the  $n$ th Fibonacci number recursively by using its definition
//Input: A nonnegative integer  $n$ 
//Output: The  $n$ th Fibonacci number
if  $n \leq 1$  return  $n$ 
else return  $F(n - 1) + F(n - 2)$ 
```

Before embarking on its formal analysis, can you tell whether this is an efficient algorithm? Well, we need to do a formal analysis anyway. The algorithm's basic operation is clearly addition, so let $A(n)$ be the number of additions performed by the algorithm in computing $F(n)$. Then the numbers of additions needed for computing $F(n - 1)$ and $F(n - 2)$ are $A(n - 1)$ and $A(n - 2)$, respectively, and the algorithm needs one more addition to compute their sum. Thus, we get the following recurrence for $A(n)$:

$$\begin{aligned}A(n) &= A(n - 1) + A(n - 2) + 1 && \text{for } n > 1, \\A(0) &= 0, A(1) = 0.\end{aligned}\tag{2.13}$$

The recurrence $A(n) - A(n - 1) - A(n - 2) = 1$ is quite similar to recurrence (2.10), but its right-hand side is not equal to zero. Such recurrences are called *inhomogeneous recurrences*. There are general techniques for solving inhomogeneous recurrences (see Appendix B or any textbook on discrete mathematics), but for this particular recurrence, a special trick leads to a faster solution. We can reduce our inhomogeneous recurrence to a homogeneous one by rewriting it as

$$[A(n) + 1] - [A(n - 1) + 1] - [A(n - 2) + 1] = 0$$

and substituting $B(n) = A(n) + 1$:

$$\begin{aligned}B(n) - B(n - 1) - B(n - 2) &= 0 \\B(0) = 1, B(1) = 1.\end{aligned}$$

This homogeneous recurrence can be solved exactly in the same manner as recurrence (2.10) was solved to find an explicit formula for $F(n)$. But it can actually be avoided by noting that $B(n)$ is, in fact, the same recurrence as $F(n)$ except that it starts with two ones and thus runs one step ahead of $F(n)$. So $B(n) = F(n + 1)$, and

$$A(n) = B(n) - 1 = F(n + 1) - 1 = \frac{1}{\sqrt{5}}(\phi^{n+1} - \hat{\phi}^{n+1}) - 1.$$

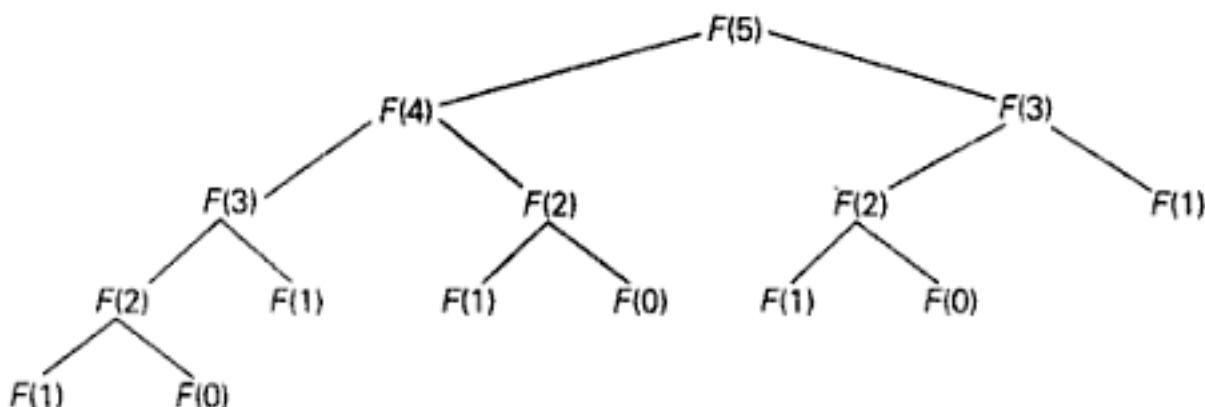


FIGURE 2.6 Tree of recursive calls for computing the 5th Fibonacci number by the definition-based algorithm

Hence, $A(n) \in \Theta(\phi^n)$ and, if we measure the size of n by the number of bits $b = \lfloor \log_2 n \rfloor + 1$ in its binary representation, the efficiency class will be even worse, namely doubly exponential.

The poor efficiency class of the algorithm could be anticipated by the nature of recurrence (2.13). Indeed, it contains two recursive calls with the sizes of smaller instances only slightly smaller than size n . (Have you encountered such a situation before?) We can also see the reason behind the algorithm's inefficiency by looking at a recursive tree of calls tracing the algorithm's execution. An example of such a tree for $n = 5$ is given in Figure 2.6. Note that the same values of the function are being evaluated again and again, which is clearly extremely inefficient.

We can obtain a much faster algorithm by simply computing the successive elements of the Fibonacci sequence iteratively, as is done in the following algorithm.

ALGORITHM *Fib(n)*

```

//Computes the nth Fibonacci number iteratively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
F[0]  $\leftarrow$  0; F[1]  $\leftarrow$  1
for i  $\leftarrow$  2 to n do
  F[i]  $\leftarrow$  F[i - 1] + F[i - 2]
return F[n]
  
```

This algorithm clearly makes $n - 1$ additions. Hence, it is linear as a function of n and “only” exponential as a function of the number of bits b in n 's binary representation. Note that using an extra array for storing all the preceding elements of the Fibonacci sequence can be avoided: storing just two values is necessary to accomplish the task (see Problem 6 in Exercises 2.5).

The third alternative for computing the n th Fibonacci number lies in using formula (2.12). The efficiency of the algorithm will obviously be determined by the efficiency of an exponentiation algorithm used for computing ϕ^n . If it is done by simply multiplying ϕ by itself $n - 1$ times, the algorithm will be in $\Theta(n) = \Theta(2^b)$. There are faster algorithms for the exponentiation problem. For example,

we discuss $\Theta(\log n) = \Theta(b)$ algorithms for this problem in Chapters 5 and 6. Note also that special care should be exercised in implementing this approach to computing the n th Fibonacci number. Since all its intermediate results are irrational numbers, we would have to make sure that their approximations in the computer are accurate enough so that the final round-off yields a correct result.

Finally, there exists a $\Theta(\log n)$ algorithm for computing the n th Fibonacci number that manipulates only integers. It is based on the equality

$$\begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \text{ for } n \geq 1$$

and an efficient way of computing matrix powers.

Exercises 2.5

1. Write a program to determine the Fibonacci series up to an integer n in the C and C++ languages.
2. Derive a thermo that describes the solution to a homogeneous second order linear recurrence with constant coefficients.
3. The maximum values of the Java primitive types int and long are $2^{31} - 1$ and $2^{63} - 1$, respectively. Find the smallest n for which the n th Fibonacci number is not going to fit in a memory allocated for
 - a. the type int.
 - b. the type long.
4.  Climbing stairs Find the number of different ways to climb an n -stair staircase if each step is either one or two stairs. (For example, a 3-stair staircase can be climbed three ways: 1-1-1, 1-2, and 2-1.) [Tuc80], p. 112.
5. Consider the recursive definition-based algorithm for computing the n th Fibonacci number $F(n)$. Let $C(n)$ and $Z(n)$ be the number of times $F(1)$ and $F(0)$, respectively, are computed. Prove that
 - a. $C(n) = F(n)$
 - b. $Z(n) = F(n-1)$.
6. Using the definition, design an algorithm that computes the n^{th} Fibonacci number recursively.
7. Prove the equality

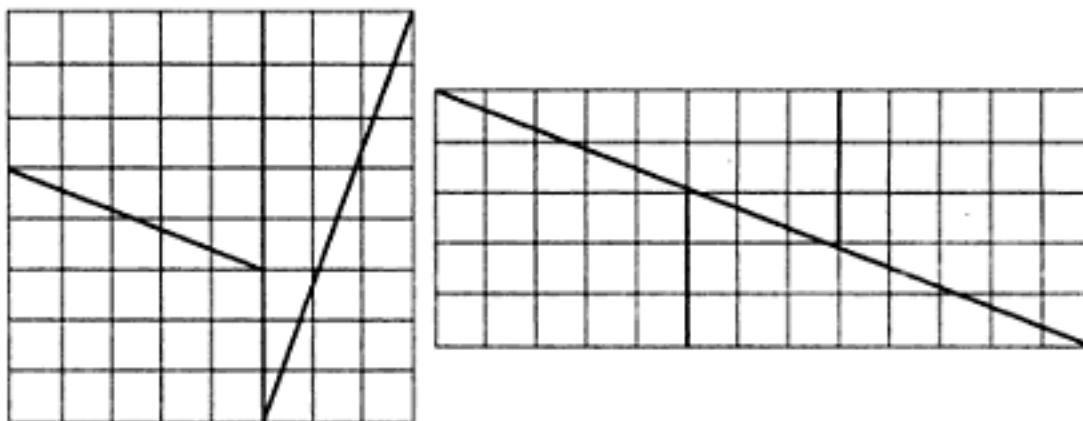
$$\begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \text{ for } n \geq 1.$$

8. How many modulo divisions are made by Euclid's algorithm on two consecutive Fibonacci numbers $F(n)$ and $F(n-1)$ as the algorithm's input?
9. a. Prove *Cassini's identity*:

$$F(n+1)F(n-1) - [F(n)]^2 = (-1)^n \quad \text{for } n \geq 1.$$



- b. Disappearing square** Consider the following paradox, which is based on Cassini's identity. Take an 8-by-8 chessboard (more generally, any $F(n)$ -by- $F(n)$ board divided into $[F(n)]^2$ squares). Cut it into two trapezoids and two triangles as shown in the left portion of the figure below. Then reassemble it as shown in the right portion of the figure. The area of the left rectangle is $8 \times 8 = 64$ squares, while the area of the right rectangle is $13 \times 5 = 65$ squares. Explain the paradox.



- 10.** In the language of your choice, implement two algorithms for computing the last five digits of the n th Fibonacci number that are based on (a) the recursive definition-based algorithm $F(n)$; (b) the iterative definition-based algorithm $Fib(n)$. Perform an experiment to find the largest value of n for which your programs run under 1 minute on your computer.

2.6 Empirical Analysis of Algorithms

In Sections 2.3 and 2.4, we saw how algorithms, both nonrecursive and recursive, can be analyzed mathematically. Though these techniques can be applied successfully to many simple algorithms, the power of mathematics, even when enhanced with more advanced techniques (see [Sed96], [Pur85], [Gra94], and [Gre82]), is far from limitless. In fact, even some seemingly simple algorithms have proved to be very difficult to analyze with mathematical precision and certainty. As we pointed out in Section 2.2, this is especially true for average-case analysis.

The principal alternative to the mathematical analysis of an algorithm's efficiency is its empirical analysis. This approach implies the steps spelled out in the following plan.

General Plan for Empirical Analysis of Algorithm Time Efficiency

1. Understand the experiment's purpose.
2. Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).
4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.

5. Generate a sample of inputs.
6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.
7. Analyze the data obtained.

Let us discuss these steps one at a time. There are several different goals one can pursue in analyzing algorithms empirically. They include checking the accuracy of a theoretical assertion about the algorithm's efficiency, comparing the efficiency of several algorithms for solving the same problem or different implementations of the same algorithm, developing a hypothesis about the algorithm's efficiency class, and ascertaining the efficiency of the program implementing the algorithm on a particular machine. Obviously, an experiment's design should depend on the question the experimenter seeks to answer.

In particular, the experiment's goal should influence, if not dictate, how the algorithm's efficiency is to be measured. The first alternative is to insert a counter (or counters) into a program implementing the algorithm to count the number of times the algorithm's basic operation is executed. This is usually a straightforward operation; you should only be mindful of the possibility that the basic operation is located in several places in the program and that all its executions need to be accounted for. As straightforward as this task usually is, you should always test the modified program to ensure that it works correctly, in terms of both the problem it solves and the counts it yields.

The second alternative is to time the program implementing the algorithm in question. The easiest way to do this is to use a system's command, such as the `time` command in UNIX. Alternatively, we can measure the running time of a code fragment by asking for the system time right before the fragment's start (t_{start}) and just after its completion (t_{finish}), and then computing the difference between the two ($t_{finish} - t_{start}$).⁷ In C and C++, you can use the function `clock` for this purpose; in Java, the method `currentTimeMillis()` in the `System` class is available.

It is important to keep several facts in mind, however. First, a system's time is typically not very accurate, and you might get somewhat different results on repeated runs of the same program on the same inputs. An obvious remedy is to make several such measurements and then take their average (or the median) as the sample's observation point. Second, given the high speed of modern computers, the running time may fail to register at all and be reported as zero. The standard trick to overcome this obstacle is to run the program in an extra loop many times, measure the total running time, and then divide it by the number of the loop's repetitions. Third, on a computer running under a time-sharing system (such as UNIX), the reported time may include the time spent by the CPU on other programs, which obviously defeats the purpose of the experiment. Therefore, you should take care to ask the system for the time devoted specifically to execution of your program. (In UNIX, this time is called the "user time," and it is automatically provided by the `time` command.)

7. If the system time is given in units called "ticks," the difference should be divided by a constant indicating the number of ticks per time unit.

Thus, measuring the physical running time has several disadvantages, both fundamental (dependence on a particular machine being the most important of them) and technical, not shared by counting the executions of a basic operation. On the other hand, the physical running time provides very specific information about an algorithm's performance in a particular computing environment, which can be of more importance to the experimenter than, say, the algorithm's asymptotic efficiency class. In addition, measuring time spent on different segments of a program can pinpoint a bottleneck in the program's performance that can be missed by an abstract deliberation about the algorithm's basic operation. Getting such data—called *profiling*—is an important resource in the empirical analysis of an algorithm's running time; the data in question can usually be obtained from the system tools available in most computing environments.

Whether you decide to measure the efficiency by basic operation counting or by time clocking, you will need to decide on a sample of inputs for the experiment. Often, the goal is to use a sample representing a “typical” input; so the challenge is to understand what a “typical” input is. For some classes of algorithms—e.g., algorithms for the traveling salesman problem discussed later in the book—researchers have developed a set of instances they use for benchmarking. But much more often than not, an input sample has to be developed by the experimenter. Typically, you will have to make decisions about the sample size (it is sensible to start with a relatively small sample and increase it later if necessary), the range of input sizes in your sample (typically neither trivially small nor excessively large), and a procedure for generating inputs in the range chosen. The instance sizes can either adhere to some pattern (e.g., 1000, 2000, 3000, . . . , 10,000 or 500, 1000, 2000, 4000, . . . , 128000) or be generated randomly within the range chosen.

The principal advantage of size changing according to a pattern is that its impact is easier to analyze. For example, if a sample's sizes are generated by doubling, we can compute the ratios $M(2n)/M(n)$ of the observed metric M (the count or the time) and see whether the ratios exhibit a behavior typical of algorithms in one of the basic efficiency classes (see Section 2.2). The major disadvantage of nonrandom sizes is the possibility that the algorithm under investigation exhibits atypical behavior on the sample chosen. For example, if all the sizes in a sample are even and an algorithm under investigation runs much more slowly on odd-size inputs, the empirical results will be quite misleading.

Another important issue concerning sizes in an experiment's sample is whether several instances of the same size should be included. If you expect the observed metric to vary considerably on instances of the same size, it is probably wise to include several instances for every size in the sample. (There are well-developed methods in statistics to help the experimenter make such decisions; you will find no shortage of books on this subject.) Of course, if several instances of the same size are included in the sample, the averages or medians of the observed values for each size should be computed and investigated instead of or in addition to individual sample points.

Much more often than not, an empirical analysis of an algorithm's efficiency requires generating random numbers. Even if we decide to use a pattern for input sizes, we typically want instances themselves generated randomly. Generating

random numbers on a digital computer is known to present a difficult problem because, in principle, the problem can be solved only approximately. This is the reason computer scientists prefer to call such numbers *pseudorandom*. As a practical matter, the easiest and most natural way of getting such numbers is to take advantage of a random number generator available in computer language libraries. Typically, its output will be a value of a (pseudo)random variable uniformly distributed in the interval between 0 and 1. If a different (pseudo)random variable is desired, an appropriate transformation needs to be made. For example, if x is a continuous random variable uniformly distributed on the interval $0 \leq x < 1$, the variable $y = l + \lfloor x(r - l) \rfloor$ will be uniformly distributed among the integer values between integers l and $r - 1$ ($l < r$).

Alternatively, you can implement one of several known algorithms for generating (pseudo)random numbers. The most widely used and thoroughly studied of such algorithms is the *linear congruential method*.

ALGORITHM *Random*(n, m, seed, a, b)

```
//Generates a sequence of  $n$  pseudorandom numbers according to the linear
//congruential method
//Input: A positive integer  $n$  and positive integer parameters  $m, \text{seed}, a, b$ 
//Output: A sequence  $r_1, \dots, r_n$  of  $n$  pseudorandom integers uniformly
//         distributed among integer values between 0 and  $m - 1$ 
//Note: Pseudorandom numbers between 0 and 1 can be obtained
//       by treating the integers generated as digits after the decimal point
 $r_0 \leftarrow \text{seed}$ 
for  $i \leftarrow 1$  to  $n$  do
     $r_i \leftarrow (a * r_{i-1} + b) \bmod m$ 
```

The simplicity of the algorithm's pseudocode is misleading because the devil lies in the details of choosing the algorithm's parameters. Here is a partial list of recommendations based on the results of a sophisticated mathematical analysis (see [Knill], pp. 184–185, for details): seed may be chosen arbitrarily and is often set to the current date and time; m should be large and may be conveniently taken as 2^w , where w is the computer's word size; a should be selected as an integer between $0.01m$ and $0.99m$ with no particular pattern in its digits but such that $a \bmod 8 = 5$; the value of b can be chosen as 1.

The empirical data obtained as the result of an experiment need to be recorded and then presented for an analysis. Data can be presented numerically in a table or graphically in a *scatterplot*, that is by points in a Cartesian coordinate system. It is a good idea to use both these options whenever it is feasible because both methods have their unique strengths and weaknesses.

The principal advantage of tabulated data lies in the opportunity to manipulate it easily. For example, we can compute the ratios $M(n)/g(n)$ where $g(n)$ is a candidate to represent the efficiency class of the algorithm in question. If the algorithm is indeed in $\Theta(g(n))$, most likely these ratios will converge to some positive constant as n gets large. (Note that careless novices sometimes assume that this constant must be 1, which is, of course, incorrect according to the definition

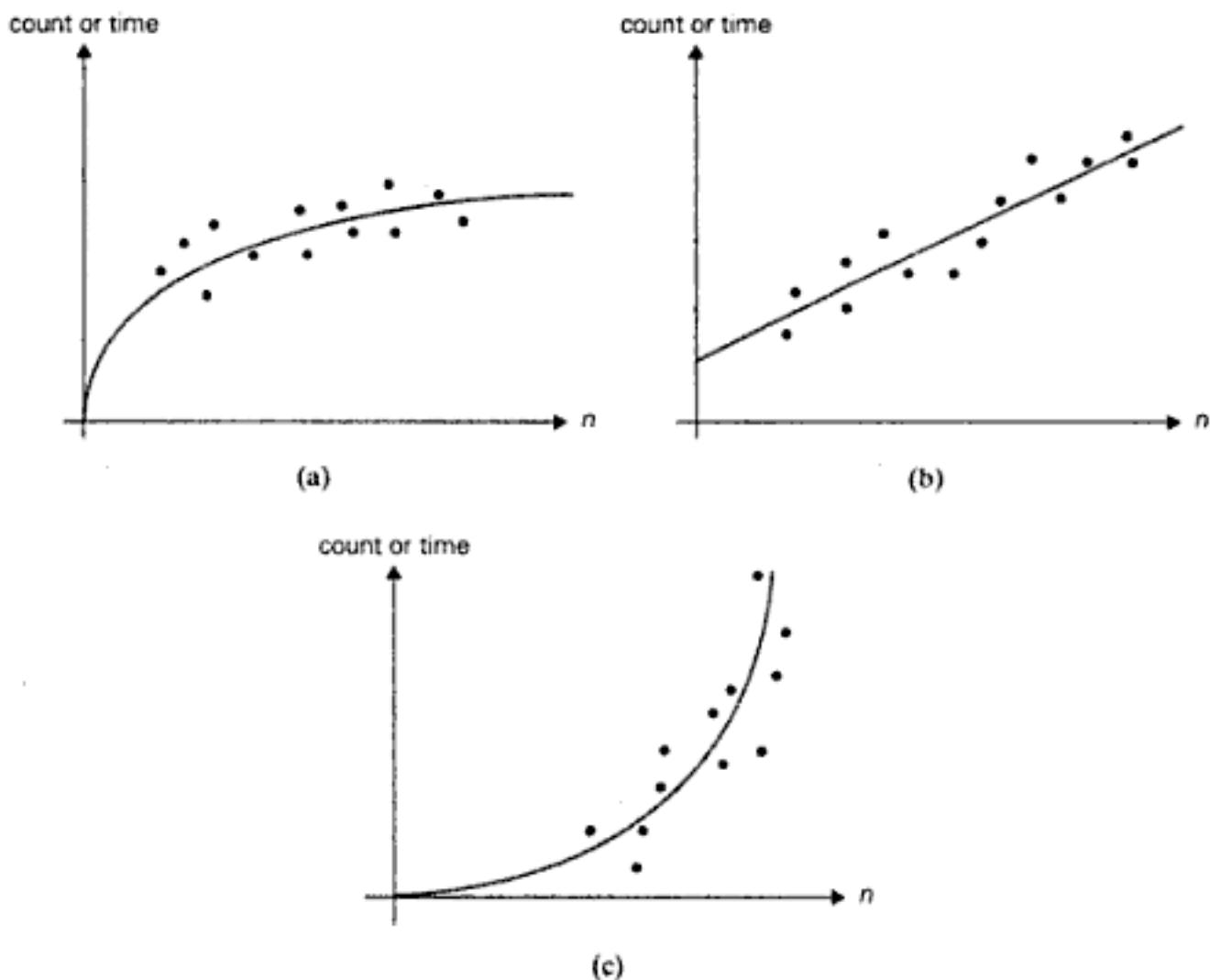


FIGURE 2.7 Typical scatterplots: (a) logarithmic; (b) linear; (c) one of the convex functions

of $\Theta(g(n))$.) Or we can compute the ratios $M(2n)/M(n)$ and see how the running time reacts to doubling of its input size. As we discussed in Section 2.2, such ratios should change only slightly for logarithmic algorithms and most likely converge to 2, 4, and 8 for linear, quadratic, and cubic algorithms, respectively—to name the most obvious and convenient cases.

On the other hand, the form of a scatterplot may also help in ascertaining the algorithm's probable efficiency class. For a logarithmic algorithm, the scatterplot will have a concave shape (Figure 2.7a); this fact distinguishes it from all the other basic efficiency classes. For a linear algorithm, the points will tend to aggregate around a straight line or, more generally, to be contained between two straight lines (Figure 2.7b). Scatterplots of functions in $\Theta(n \lg n)$ and $\Theta(n^2)$ will have a convex shape (Figure 2.7c), making them difficult to differentiate. A scatterplot of a cubic algorithm will also have a convex shape, but it will show a much more rapid increase in the metric's values. An exponential algorithm will most probably require a logarithmic scale for the vertical axis, in which the values of $\log_a M(n)$ rather than those of $M(n)$ are plotted. (The commonly used logarithm base is 2 or 10.) In such a coordinate system, a scatterplot of a truly exponential algorithm should resemble a linear function because $M(n) \approx c a^n$ implies $\log_b M(n) \approx \log_b c + n \log_b a$ and vice versa.

One of the possible applications of the empirical analysis is to predict the algorithm's performance on an instance not included in the experiment sample. For example, if we observe that the ratios $M(n)/g(n)$ are close to some constant c for the sample instances, we can approximate $M(n)$ by the product $cg(n)$ for other instances, too. Though this approach is sensible, it should be used with caution, especially for values of n outside the sample range. (Mathematicians call such predictions *extrapolation*, as opposed to *interpolation*, which deals with values within the sample range.) Of course, we can also try unleashing the standard techniques of statistical data analysis and prediction. Note, however, that the majority of such techniques are based on specific probabilistic assumptions that may or may not be valid for the experimental data in question.

It seems appropriate to end this section by pointing out the basic differences between mathematical and empirical analyses of algorithms. The principal strength of the mathematical analysis is its independence of specific inputs; its principal weakness is its limited applicability, especially for investigating the average-case efficiency. The principal strength of the empirical analysis lies in its applicability to any algorithm, but its results can depend on the particular sample of instances and the computer used in the experiment.

Exercises 2.6

1. Consider the following well-known sorting algorithm (we study it more closely later in the book) with a counter inserted to count the number of key comparisons.

ALGORITHM *SortAnalysis*($A[0..n - 1]$)

```
//Input: An array  $A[0..n - 1]$  of  $n$  orderable elements
//Output: The total number of key comparisons made
count ← 0
for  $i \leftarrow 1$  to  $n - 1$  do
     $v \leftarrow A[i]$ 
     $j \leftarrow i - 1$ 
    while  $j \geq 0$  and  $A[j] > v$  do
        count ← count + 1
         $A[j + 1] \leftarrow A[j]$ 
         $j \leftarrow j - 1$ 
     $A[j + 1] \leftarrow v$ 
return count
```

Is the comparison counter inserted in the right place? If you believe it is, prove it; if you believe it is not, make an appropriate correction.

2. a. Run the program of Problem 1, with a properly inserted counter (or counters) for the number of key comparisons, on 20 random arrays of sizes 1000, 1500, 2000, 2500, ..., 9000, 9500.

- b. Analyze the data obtained to form a hypothesis about the algorithm's average-case efficiency.
 - c. Estimate the number of key comparisons one should expect for a randomly generated array of size 10,000 sorted by the same algorithm.
3. Why are the random numbers generated by digital computers called as "Pseudorandom"?
4. Hypothesize a likely efficiency class of an algorithm based on the following empirical observations of its basic operation's count:

size	1000	1500	2000	2500	3000	3500	4000	4500
count	10,400	18,100	25,600	32,900	39,880	46,450	53,112	61,200

5. What scale transformation will make a logarithmic scatterplot look like a linear one?
6. How can we distinguish a scatterplot for an algorithm in $\Theta(\lg \lg n)$ from a scatterplot for an algorithm in $\Theta(\lg n)$?
7. a. Find empirically the largest number of divisions made by Euclid's algorithm for computing $\gcd(m, n)$ for $1 \leq n \leq m \leq 100$.
 b. For each positive integer k , find empirically the smallest pair of integers $1 \leq n \leq m \leq 100$ for which Euclid's algorithm needs to make k divisions in order to find $\gcd(m, n)$.
8. The average-case efficiency of Euclid's algorithm on inputs of size n can be measured by the average number of divisions $D_{avg}(n)$ made by the algorithm in computing $\gcd(n, 1), \gcd(n, 2), \dots, \gcd(n, n)$. For example,

$$D_{avg}(5) = \frac{1}{5}(1 + 2 + 3 + 2 + 1) = 1.8.$$

Produce a scatterplot of $D_{avg}(n)$ and indicate the algorithm's likely average-case efficiency class.

9. Run an experiment to ascertain the efficiency class of the sieve of Eratosthenes (see Section 1.1).
10. a. Explain the typical scatterplot forms and bring out their uses.
 b. What are the circumstances in which extrapolation may be needed during empirical analysis?

2.7 Algorithm Visualization

In addition to the mathematical and empirical analyses of algorithms, there is yet a third way to study algorithms. It is called **algorithm visualization** and can be defined as the use of images to convey some useful information about algorithms. That information can be a visual illustration of an algorithm's operation, of its performance on different kinds of inputs, or of its execution speed versus that of other algorithms for the same problem. To accomplish this goal,

an algorithm visualization uses graphic elements (points, line segments, two- or three-dimensional bars, and so on) to represent some “interesting events” in the algorithm’s operation.

There are two principal variations of algorithm visualization:

- static algorithm visualization
- dynamic algorithm visualization, also called ***algorithm animation***.

Static algorithm visualization shows an algorithm’s progress through a series of still images. Algorithm animation, on the other hand, shows a continuous, movie-like presentation of an algorithm’s operations. Animation is an arguably more sophisticated option, and it is, of course, much more difficult to implement.

Early efforts in the area of algorithm visualization go back to the 1970s. The watershed event happened in 1981 with the appearance of a 30-minute color sound film titled *Sorting Out Sorting*. The algorithm visualization classic was produced at the University of Toronto by Ronald Baecker with the assistance of D. Sherman [Bae81, Bae98]. It contained visualizations of nine well-known sorting algorithms (more than half of them are discussed later in the book) and provided quite a convincing demonstration of their relative speeds.

The success of *Sorting Out Sorting* made sorting algorithms a perennial favorite for algorithm animation. Indeed, the sorting problem lends itself quite naturally to visual presentation via vertical or horizontal bars or sticks of different heights or lengths, which are rearranged according to their sizes (Figure 2.8). This presentation is convenient, however, only for illustrating actions of a typical sorting algorithm on small inputs. For larger files, *Sorting Out Sorting* used the ingenious idea of presenting data by a scatterplot of points on a coordinate plane, with the first coordinate representing an item’s position in the file and the second one representing the item’s value; with such a representation, the process of sorting looks like a transformation of a “random” scatterplot of points into the points along a frame’s diagonal (Figure 2.9). In addition, most sorting algorithms work by comparing and exchanging two given items at a time—an event that can be animated relatively easily.

Since the appearance of *Sorting Out Sorting*, a great number of algorithm animations have been created. They range in scope from one particular algorithm to a group of algorithms for the same problem (e.g., sorting) or the same application area (e.g., geometric algorithms) to general-purpose animation systems. The most widely known of the general-purpose systems include BALSA [Bro84], TANGO [Sta90], and ZEUS [Bro91]; a comparative review of their features, along with those of nine other packages, can be found in [Pri93]. A good general-purpose animation system should allow a user to not only watch and interact with existing animations of a wide variety of algorithms; it should also provide facilities for creating new animations. Experience has shown that creating such systems is a difficult but not impossible task.

The appearance of Java and the World Wide Web has given a new impetus to algorithm animation. You are advised to start an exploration with an up-to-date site containing a collection of links to sites devoted to algorithm animation. Since the Web world is notorious for its instability, no specific Web addresses appear here; a search for the phrase “algorithm animation” or “algorithm visualization” with a good search engine should do the trick. While you peruse and

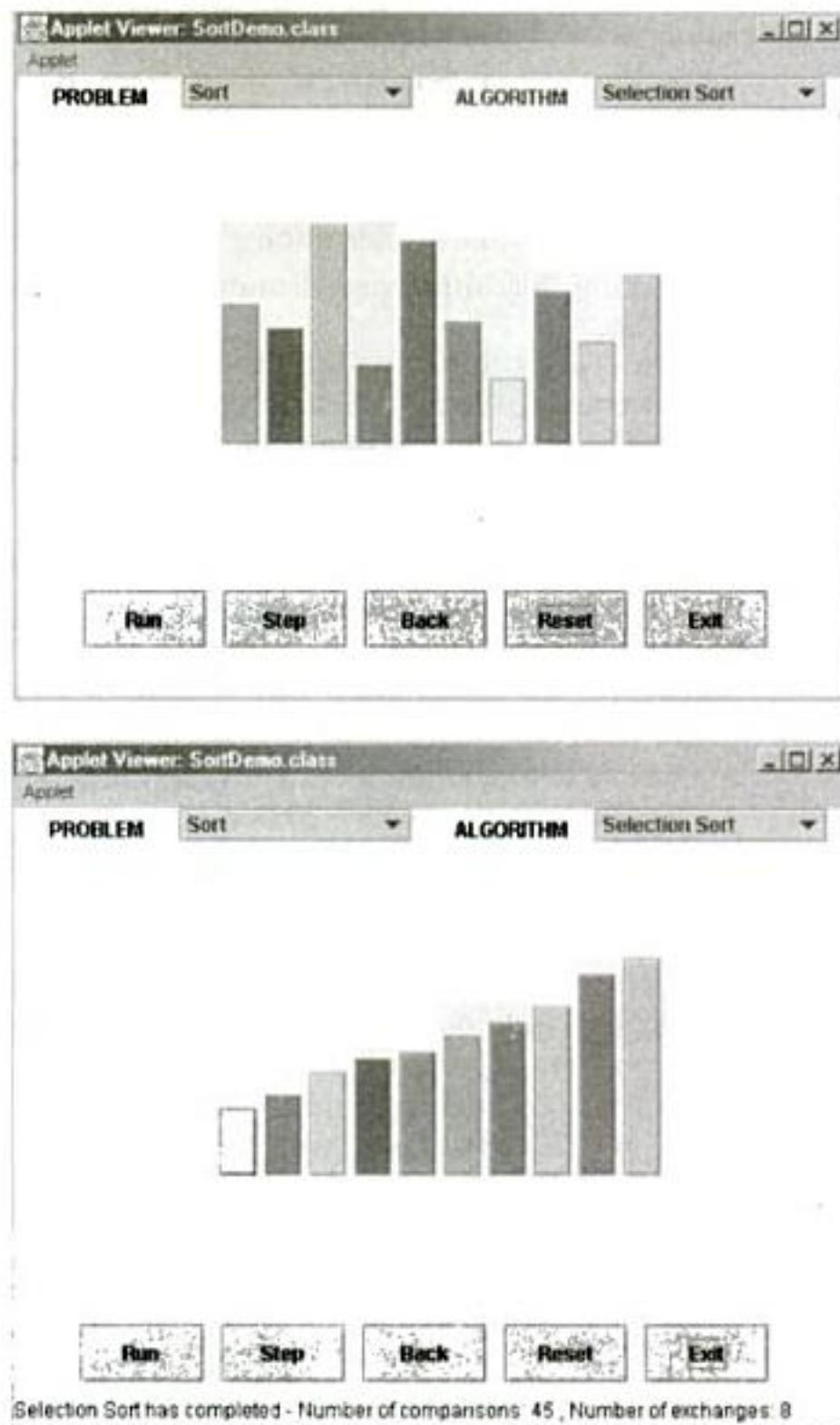


FIGURE 2.8 Initial and final screens of a typical visualization of a sorting algorithm using the bar representation

evaluate different algorithm animations, you may want to keep in mind the “ten commandments of algorithm animations.” This list of desirable features of an animation’s user interface was suggested by Peter Gloor [Glo98], who was a principal developer of *Animated Algorithms*, another well-known algorithm visualization system:

1. Be consistent.
2. Be interactive.

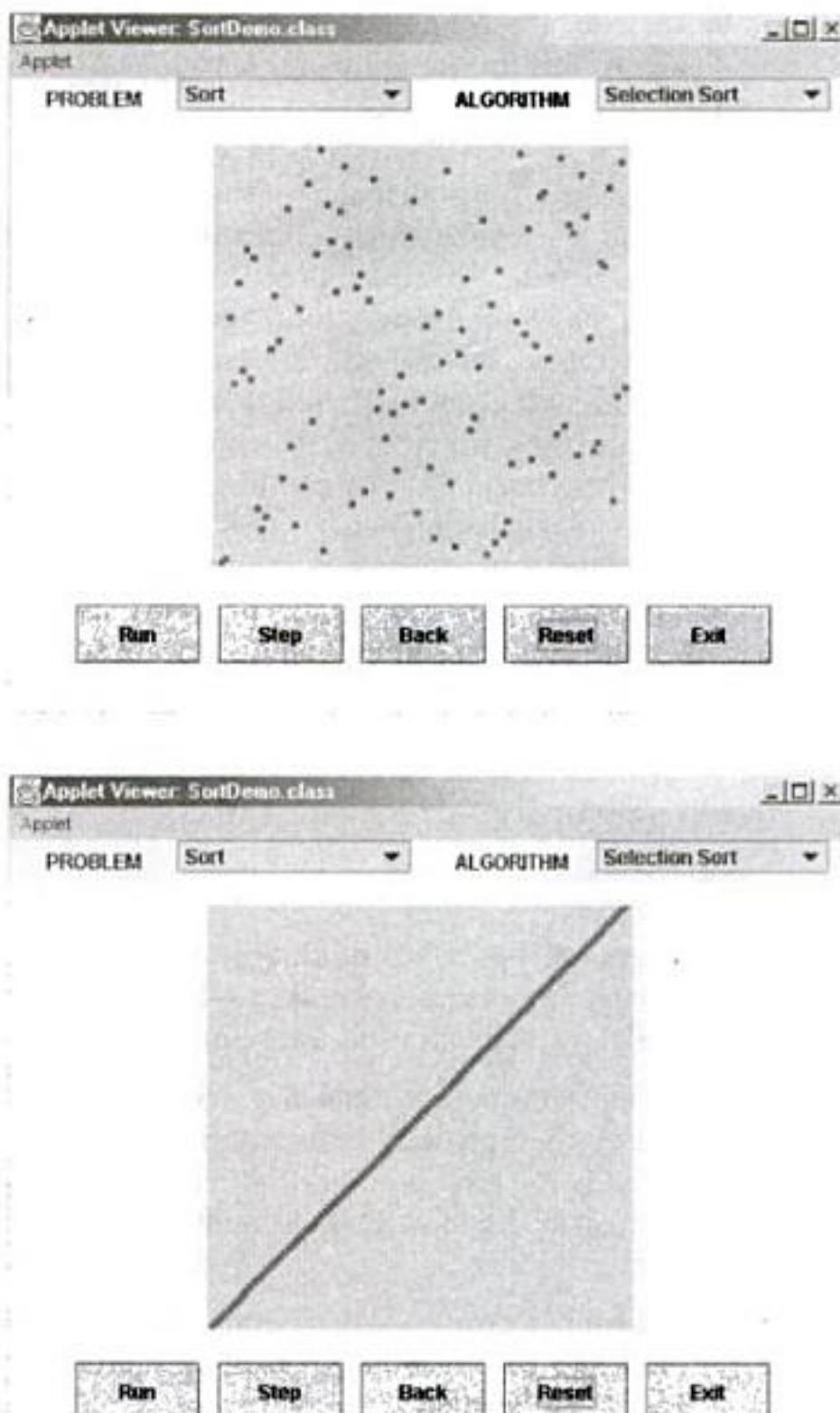


FIGURE 2.9 Initial and final screens of a typical visualization of a sorting algorithm using the scatterplot representation

3. Be clear and concise.
4. Be forgiving to the user.
5. Adapt to the knowledge level of the user.
6. Emphasize the visual component.
7. Keep the user interested.
8. Incorporate both symbolic and iconic representations.

9. Include algorithm's analysis (run statistics) and comparisons with other algorithms for the same problem.
10. Include execution history.

There are two principal applications of algorithm visualization: research and education. The application to education seeks to help students learning algorithms. Potential benefits for researchers are based on expectations that algorithm visualization may help uncover some unknown features of algorithms. For example, one researcher used a visualization of the recursive Tower of Hanoi algorithm in which odd- and even-numbered disks were two different colors. He noticed that two disks of the same color never came in direct contact during the algorithm's execution. This observation helped him in developing a better nonrecursive version of the classic algorithm.

Although some successes in both education and research applications have been reported, they are not as impressive as one might expect. Experience has shown that creating sophisticated software systems is not going to be enough. A deeper understanding of human perception of images will be required before the true potential of algorithm visualization is fulfilled.

SUMMARY

- There are two kinds of algorithm efficiency: time efficiency and space efficiency. *Time efficiency* indicates how fast the algorithm runs; *space efficiency* deals with the extra space it requires.
- An algorithm's time efficiency is principally measured as a function of its input size by counting the number of times its basic operation is executed. A *basic operation* is the operation that contributes most toward running time. Typically, it is the most time-consuming operation in the algorithm's innermost loop.
- For some algorithms, the running time may differ considerably for inputs of the same size, leading to *worst-case* efficiency, *average-case* efficiency, and *best-case* efficiency.
- The established framework for analyzing an algorithm's time efficiency is primarily grounded in the order of growth of the algorithm's running time as its input size goes to infinity.
- The notations O , Ω , and Θ are used to indicate and compare the asymptotic orders of growth of functions expressing algorithm efficiencies.
- The efficiencies of a large number of algorithms fall into the following few classes: *constant*, *logarithmic*, *linear*, "*n-log-n*," *quadratic*, *cubic*, and *exponential*.
- The main tool for analyzing the time efficiency of a nonrecursive algorithm is to set up a sum expressing the number of executions of its basic operation and ascertain the sum's order of growth.

- The main tool for analyzing the time efficiency of a recursive algorithm is to set up a recurrence relation expressing the number of executions of its basic operation and ascertain the solution's order of growth.
- Succinctness of a recursive algorithm may mask its inefficiency.
- The *Fibonacci numbers* are an important sequence of integers in which every element is equal to the sum of its two immediate predecessors. There are several algorithms for computing the Fibonacci numbers with drastically different efficiencies.
- Empirical analysis of an algorithm is performed by running a program implementing the algorithm on a sample of inputs and analyzing the data observed (the basic operation's count or physical running time). This often involves generating pseudorandom numbers. The applicability to any algorithm is the principal strength of this approach; the dependence of results on the particular computer and instance sample is its main weakness.
- *Algorithm visualization* is the use of images to convey useful information about algorithms. The two principal variations of algorithm visualization are static algorithm visualization and dynamic algorithm visualization (also called *algorithm animation*).

3

Brute Force

Science is as far removed from brute force as this sword from a crowbar.

—Edward Lytton (1803–1873), *Leila*, Book II, Chapter I

Doing a thing well is often a waste of time.

—Robert Byrne, master pool and billiards player and writer

After introducing the framework and methods for algorithm analysis in the preceding chapter, we are ready to embark on a discussion of algorithm design techniques. Each of the next eight chapters is devoted to a particular design strategy. The subject of this chapter is brute force—the simplest of the design strategies. It can be described as follows:

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

The “force” implied by the strategy’s definition is that of a computer and not that of one’s intellect. “Just do it!” would be another way to describe the prescription of the brute-force approach. And often, the brute-force strategy is indeed the one that is easiest to apply.

As an example, consider the exponentiation problem: compute a^n for a given number a and a nonnegative integer n . Though this problem might seem trivial, it provides a useful vehicle for illustrating several algorithm design techniques, including the brute-force approach. (Also note that computing $a^n \bmod m$ for some large integers is a principal component of a leading encryption algorithm.) By the definition of exponentiation,

$$a^n = \underbrace{a * \cdots * a}_{n \text{ times}}.$$

This suggests simply computing a^n by multiplying 1 by a n times.

We have already encountered at least two brute-force algorithms in the book: the consecutive integer checking algorithm for computing $\text{gcd}(m, n)$ (Section 1.1) and the definition-based algorithm for matrix multiplication (Section 2.3). Many other examples are given later in this chapter. (Can you identify a few algorithms you already know as being based on the brute-force approach?)

Though rarely a source of clever or efficient algorithms, the brute-force approach should not be overlooked as an important algorithm design strategy. First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems. (In fact, it seems to be the only general approach for which it is more difficult to point out problems it *cannot* tackle.) Second, for some important problems (e.g., sorting, searching, matrix multiplication, string matching), the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size. Third, the expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with acceptable speed. Fourth, even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem. Finally, a brute-force algorithm can serve an important theoretical or educational purpose, e.g., as a yardstick with which to judge more efficient alternatives for solving a problem.

3.1 Selection Sort and Bubble Sort

In this section, we consider the application of the brute-force approach to the problem of sorting: given a list of n orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in nondecreasing order. As we mentioned in Section 1.3, dozens of algorithms have been developed for solving this very important problem. You might have learned several of them in the past. If you have, try to forget them for the time being and look at the problem afresh.

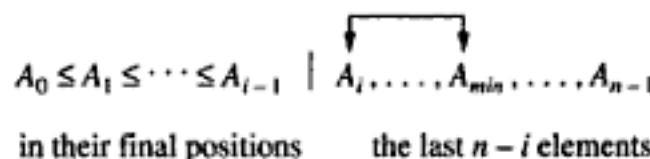


Now, after your mind is unburdened of previous knowledge of sorting algorithms, ask yourself a question: "What would be the most straightforward method for solving the sorting problem?" Reasonable people may disagree on the answer to this question. The two algorithms discussed here—selection sort and bubble sort—seem to be the two prime candidates. The first of these two algorithms is a better algorithm overall, and it implements the brute-force approach more clearly.

Selection Sort

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last $n - 1$ elements and exchange it with the second element, putting the second smallest element in its final position. Generally, on the i th pass through the list, which we number from 0 to $n - 2$, the

algorithm searches for the smallest item among the last $n - i$ elements and swaps it with A_i :



After $n - 1$ passes, the list is sorted.

Here is a pseudocode of this algorithm, which, for simplicity, assumes that the list is implemented as an array.

ALGORITHM *SelectionSort(A[0..n - 1])*

```
//Sorts a given array by selection sort
//Input: An array A[0..n - 1] of orderable elements
//Output: Array A[0..n - 1] sorted in ascending order
for i ← 0 to n - 2 do
    min ← i
    for j ← i + 1 to n - 1 do
        if A[j] < A[min]  min ← j
    swap A[i] and A[min]
```

As an example, the action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated in Figure 3.1.

The analysis of selection sort is straightforward. The input's size is given by the number of elements n ; the algorithm's basic operation is the key comparison $A[j] < A[min]$. The number of times it is executed depends only on the array's size and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

I	89	45	68	90	29	34	17
17	I	45	68	90	29	34	89
17	29	I	68	90	45	34	89
17	29	34	I	90	45	68	89
17	29	34	45	I	90	68	89
17	29	34	45	68	I	90	89
17	29	34	45	68	89	I	90

FIGURE 3.1 Example of sorting with selection sort. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

We have already encountered the last sum in analyzing the algorithm of Example 2 in Section 2.3 (so you should be able to compute it now on your own). Whether you compute this sum by distributing the summation symbol or by immediately getting the sum of decreasing integers, the answer, of course, must be the same:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}.$$

Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs. Note, however, that the number of key swaps is only $\Theta(n)$ or, more precisely, $n-1$ (one for each repetition of the i loop). This property distinguishes selection sort positively from many other sorting algorithms.

Bubble Sort

Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up “bubbling up” the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on until, after $n-1$ passes, the list is sorted. Pass i ($0 \leq i \leq n-2$) of bubble sort can be represented by the following diagram:

$$A_0, \dots, A_j \xrightarrow{?} A_{j+1}, \dots, A_{n-i-1} \mid A_{n-i} \leq \dots \leq A_{n-1}$$

in their final positions

Here is a pseudocode of this algorithm.

ALGORITHM *BubbleSort(A[0..n - 1])*

```

//Sorts a given array by bubble sort
//Input: An array A[0..n - 1] of orderable elements
//Output: Array A[0..n - 1] sorted in ascending order
for i ← 0 to n - 2 do
    for j ← 0 to n - 2 - i do
        if A[j + 1] < A[j] swap A[j] and A[j + 1]

```

The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated as an example in Figure 3.2.

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n ; it is obtained by a sum that is almost identical to the sum for selection sort:

$$\begin{aligned}
C(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] \\
&= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).
\end{aligned}$$

89	?	45	68	90	29	34	17
45	89	?	68	90	29	34	17
45	68	89	?	90	29	34	17
45	68	89	29	90	?	34	17
45	68	89	29	34	90	?	17
45	68	89	29	34	17	1	90
45	?	68	?	89	?	29	34
45	68	29	89	?	34	17	1
45	68	29	34	89	?	17	1
45	68	29	34	17	1	89	90

etc.

FIGURE 3.2 First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

The number of key swaps, however, depends on the input. For the worst case of decreasing arrays, it is the same as the number of key comparisons:

$$S_{\text{worst}}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

As is often the case with an application of the brute-force strategy, the first version of an algorithm obtained can often be improved with a modest amount of effort. Specifically, we can improve the crude version of bubble sort given above by exploiting the following observation: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm (Problem 9a in Exercises 3.1). Though the new version runs faster on some inputs, it is still in $\Theta(n^2)$ in the worst and average cases. In fact, even among elementary sorting methods, bubble sort is an inferior choice, and, if it were not for its catchy name, you would probably have never heard of it. However, the general lesson you just learned is important and worth repeating:

A first application of the brute-force approach often results in an algorithm that can be improved with a modest amount of effort.

Exercises 3.1

1. Describe brute-force approach. What are the advantages and disadvantages of this approach?
2. a. What is the efficiency of the brute-force algorithm for computing a^n as a function of n ? As a function of the number of bits in the binary representation of n ?

- b. If you are to compute $a^n \bmod m$ where $a > 1$ and n is a large positive integer, how would you circumvent the problem of a very large magnitude of a^n ?
- 3. For each of the algorithms in Problems 4, 5, and 6 of Exercises 2.3, tell whether or not the algorithm is based on the brute-force approach.
- 4. a. Design a brute-force algorithm for computing the value of a polynomial

$$p(x) = a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0$$
 at a given point x_0 and determine its worst-case efficiency class.
 - b. If the algorithm you designed is in $\Theta(n^2)$, design a linear algorithm for this problem.
 - c. Is it possible to design an algorithm with a better than linear efficiency for this problem?
- 5. Sort the list {do, while, for, int, if} in lexical order by selection sort.
- 6. Is selection sort stable? (The definition of a stable sorting algorithm was given in Section 1.3.)
- 7. Is it possible to implement selection sort for linked lists with the same $\Theta(n^2)$ efficiency as the array version?
- 8. Sort the list E, X, A, M, P, L, E in alphabetical order by bubble sort.
- 9. a. Prove that if bubble sort makes no exchanges on its pass through a list, the list is sorted and the algorithm can be stopped.
 - b. Write a pseudocode of the method that incorporates this improvement.
 - c. Prove that the worst-case efficiency of the improved version is quadratic.
- 10. Is bubble sort stable?
- 11. You have a row of binary digits arranged randomly. Arrange them in such an order that all 0's precede all 1s or vice-versa. The only constraint in arranging them is that you are allowed to interchange the positions of binary digits if they are not similar.

1 1 0 1 0 1 0 0 → 0 0 0 0 1 1 1 1

Design an algorithm for solving this problem.

3.2 Sequential Search and Brute-Force String Matching

We saw in the previous section two applications of the brute-force approach to the sorting problem. Here we discuss two applications of this strategy to the problem of searching. The first deals with the canonical problem of searching for an item of a given value in a given list. The second is different in that it deals with the string-matching problem.

Sequential Search

We have already encountered a brute-force algorithm for the general searching problem: it is called sequential search (see Section 2.1). To repeat, the algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search). A simple extra trick is often employed in implementing sequential search: if we append the search key to the end of the list, the search for the key will have to be successful, and therefore we can eliminate a check for the list's end on each iteration of the algorithm. Here is a pseudocode for this enhanced version, with its input implemented as an array.

ALGORITHM *SequentialSearch2(A[0..n], K)*

```

//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n - 1] whose value is
//         equal to K or -1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
    i ← i + 1
if i < n return i
else return -1

```

Another straightforward improvement can be incorporated in sequential search if a given list is known to be sorted: searching in such a list can be stopped as soon as an element greater than or equal to the search key is encountered.

Sequential search provides an excellent illustration of the brute-force approach, with its characteristic strength (simplicity) and weakness (inferior efficiency). The efficiency results obtained in Section 2.1 for the standard version of sequential search change for the enhanced version only very slightly, so that the algorithm remains linear in both worst and average cases. We discuss later in the book several searching algorithms with a better time efficiency.

Brute-Force String Matching

Recall the string-matching problem introduced in Section 1.3: given a string of *n* characters called the *text* and a string of *m* characters ($m \leq n$) called the *pattern*, find a substring of the text that matches the pattern. To put it more precisely, we want to find *i*—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \dots, t_{i+j} = p_j, \dots, t_{i+m-1} = p_{m-1}$:

t_0	...	t_i	...	t_{i+j}	...	t_{i+m-1}	...	t_{n-1}	text <i>T</i>
↓		↓		↓					
p_0	...	p_j	...	p_{m-1}	pattern <i>P</i>				

If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

A brute-force algorithm for the string-matching problem is quite obvious: align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered. In the latter case, shift the pattern one position to the right and resume character comparisons, starting again with the first character of the pattern and its counterpart in the text. Note that the last position in the text which can still be a beginning of a matching substring is $n - m$ (provided the text's positions are indexed from 0 to $n - 1$). Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

ALGORITHM *BruteForceStringMatch($T[0..n - 1]$, $P[0..m - 1]$)*

```
//Implements brute-force string matching
//Input: An array  $T[0..n - 1]$  of  $n$  characters representing a text and
//       an array  $P[0..m - 1]$  of  $m$  characters representing a pattern
//Output: The index of the first character in the text that starts a
//       matching substring or  $-1$  if the search is unsuccessful
for  $i \leftarrow 0$  to  $n - m$  do
     $j \leftarrow 0$ 
    while  $j < m$  and  $P[j] = T[i + j]$  do
         $j \leftarrow j + 1$ 
        if  $j = m$  return  $i$ 
return  $-1$ 
```

An operation of the algorithm is illustrated in Figure 3.3.

Note that for this example, the algorithm shifts the pattern almost always after a single character comparison. However, the worst case is much worse: the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the $n - m + 1$ tries. (Problem 6 asks you to give a specific example of such a situation.) Thus, in the worst case, the algorithm is in $\Theta(nm)$. For a typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons (check

N	O	B	O	D	Y	_	N	O	T	I	C	E	D	_	H	I	M
N	O	T															
N	O	T															
N	O	T															
N	O	T															
N	O	T															
N	O	T															
N	O	T															
N	O	T															
N	O	T															
N	O	T															

FIGURE 3.3 Example of brute-force string matching. (The pattern's characters that are compared with their text counterparts are in bold type.)

the example again). Therefore, the average-case efficiency should be considerably better than the worst-case efficiency. Indeed it is: for searching in random texts, it has been shown to be linear, i.e., $\Theta(n + m) = \Theta(n)$. There are several more sophisticated and more efficient algorithms for string searching. The most widely known of them—by R. Boyer and J. Moore—is outlined in Section 7.2 along with its simplification suggested by R. Horspool.

Exercises 3.2

1. Find the number of comparisons required to search for ‘6’ in the given sequence of numbers: 10, 19, 7, 9, 6, 15.
2. As shown in Section 2.1, the average number of key comparisons made by sequential search (without a sentinel, under standard assumptions about its inputs) is given by the formula

$$C_{\text{avg}}(n) = \frac{p(n+1)}{2} + n(1-p),$$

where p is the probability of a successful search. Determine, for a fixed n , the values of p ($0 \leq p \leq 1$) for which this formula yields the largest value of $C_{\text{avg}}(n)$ and the smallest value of $C_{\text{avg}}(n)$.



3. *Gadget testing* A firm wants to determine the highest floor of its n -story headquarters from which a gadget can fall with no impact on the gadget’s functionality. The firm has two identical gadgets to experiment with. Design an algorithm in the best efficiency class you can to solve this problem.
4. Find the number of character comparisons that will be made by ‘straight forward string matching’ for the pattern ABABC in the following text:

BAABABABCCA

5. Write a brute-force algorithm for counting the number of vowels in a given text.
6. Give an example of a text of length n and a pattern of length m that constitutes a worst-case input for the brute-force string-matching algorithm. Exactly how many character comparisons will be made for such input?
7. Write a visualization program for the brute-force string-matching algorithm.
8. In solving the string-matching problem, would there be any advantage in comparing pattern and text characters right-to-left instead of left-to-right?
9. Consider the problem of counting, in a given text, the number of substrings that start with an A and end with a B. (For example, there are four such substrings in CABAAAXBYA.)
 - a. Design a brute-force algorithm for this problem and determine its efficiency class.
 - b. Design a more efficient algorithm for this problem [Gin04].



10. Word find A popular diversion in the United States, “word find,” asks the player to find each of a given set of words in a square table filled with single letters. A word can read horizontally (left or right), vertically (up or down), or along a 45 degree diagonal (in any of the four directions), formed by consecutively adjacent cells of the table; it may wrap around the table’s boundaries, but it must read in the same direction with no zigzagging. The same cell of the table may be used in different words, but, in a given word, the same cell may be used no more than once. Write a computer program for solving this puzzle.



11. Battleship game Write a program based on a version of brute-force pattern matching for playing Battleship (a classic strategy game) on the computer. The rules of the game are as follows. There are two opponents in the game (in this case, a human player and the computer). The game is played on two identical boards (10-by-10 tables of squares) on which each opponent places his or her ships, not seen by the opponent. Each player has five ships, each of which occupies a certain number of squares on the board: a destroyer (2 squares), a submarine (3 squares), a cruiser (3 squares), a battleship (4 squares), and an aircraft carrier (5 squares). Each ship is placed either horizontally or vertically, with no two ships touching each other. The game is played by the opponents taking turns “shooting” at each other’s ships. The result of every shot is displayed as either a hit or a miss. In case of a hit, the player gets to go again and keeps playing until missing. The goal is to sink all the opponent’s ships before the opponent succeeds in doing it first. (To sink a ship, all squares occupied by the ship must be hit.)

3.3 Closest-Pair and Convex-Hull Problems by Brute Force

In this section, we consider a straightforward approach to two well-known problems dealing with a finite set of points in the plane. These problems, aside from their theoretical interest, arise in two important applied areas: computational geometry and operations research.

Closest-Pair Problem

The closest-pair problem calls for finding two closest points in a set of n points. For simplicity, we consider the two-dimensional case, although the problem can be posed for points in higher-dimensional spaces as well. We assume that the points in question are specified in a standard fashion by their (x, y) Cartesian coordinates and that the distance between two points $P_i = (x_i, y_i)$ and $P_j = (x_j, y_j)$ is the standard Euclidean distance

$$d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

The brute-force approach to solving this problem leads to the following obvious algorithm: compute the distance between each pair of distinct points and find a pair with the smallest distance. Of course, we do not want to compute the distance between the same pair of points twice. To avoid doing so, we consider only the pairs of points (P_i, P_j) for which $i < j$.

ALGORITHM *BruteForceClosestPoints(P)*

```

//Finds two closest points in the plane by brute force
//Input: A list  $P$  of  $n$  ( $n \geq 2$ ) points  $P_1 = (x_1, y_1), \dots, P_n = (x_n, y_n)$ 
//Output: Indices index1 and index2 of the closest pair of points
dmin  $\leftarrow \infty$ .
for  $i \leftarrow 1$  to  $n - 1$  do
    for  $j \leftarrow i + 1$  to  $n$  do
         $d \leftarrow \text{sqrt}((x_i - x_j)^2 + (y_i - y_j)^2)$  //sqrt is the square root function
        if  $d < dmin$ 
             $dmin \leftarrow d$ ; index1  $\leftarrow i$ ; index2  $\leftarrow j$ 
return index1, index2
```

The basic operation of the algorithm is computing the Euclidean distance between two points. In the age of electronic calculators with the square-root button, one might be led to believe that computing the square root is as simple an operation as, say, addition or multiplication. Actually, it is not. For starters, even for most integers, square roots are irrational numbers that therefore can be found only approximately. Moreover, computing such approximations is not a trivial matter. But, in fact, computing square roots can be avoided! (Can you think how?) The trick is to realize that we can simply ignore the square root function and compare the values $(x_i - x_j)^2 + (y_i - y_j)^2$ themselves. We can do this because the smaller a number of which we take the square root, the smaller its square root, or, as mathematicians say, the square root function is strictly increasing.

So, if we replace $d \leftarrow \text{sqrt}((x_i - x_j)^2 + (y_i - y_j)^2)$ by $dsqr \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2$, the basic operation of the algorithm will be squaring a number. The number of times it will be executed can be computed as follows:

$$\begin{aligned}
 C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n 2 = 2 \sum_{i=1}^{n-1} (n-i) \\
 &= 2[(n-1) + (n-2) + \cdots + 1] = (n-1)n \in \Theta(n^2).
 \end{aligned}$$

In Chapter 4, we discuss an $n \log n$ algorithm for this problem.

Convex-Hull Problem

On to the other problem—computing the convex hull. We start with a definition of a convex set.

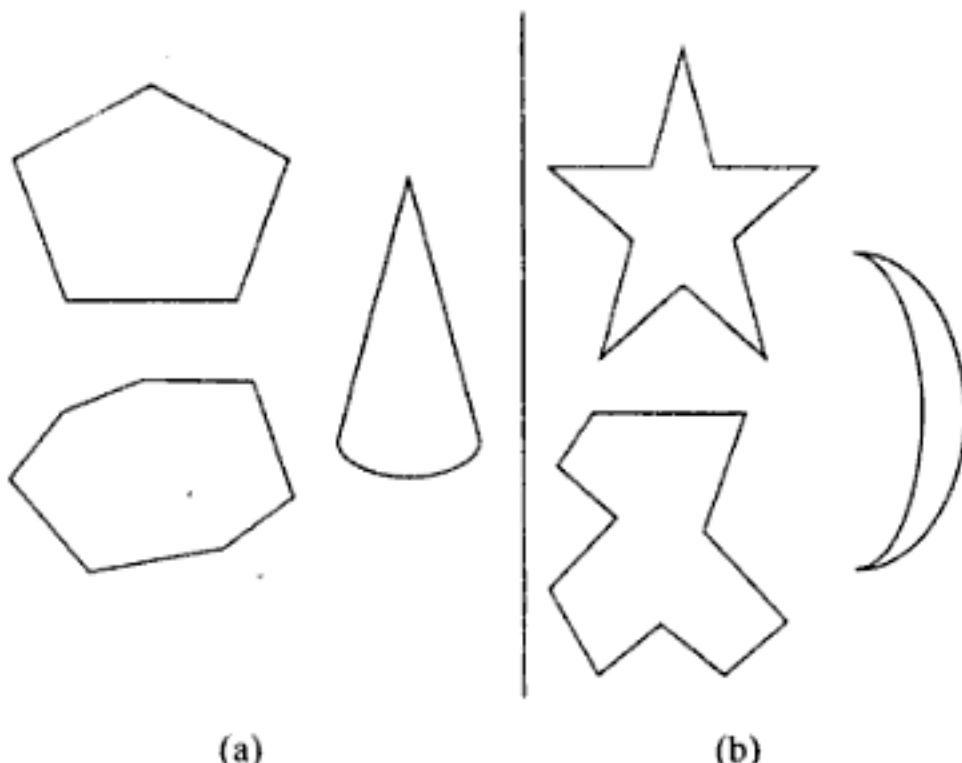


FIGURE 3.4 (a) Convex sets. (b) Sets that are not convex.

DEFINITION A set of points (finite or infinite) in the plane is called **convex** if for any two points P and Q in the set, the entire line segment with the endpoints at P and Q belongs to the set.

All the sets depicted in Figure 3.4a are convex, and so are a straight line, a triangle, a rectangle, and, more generally, any convex polygon,¹ a circle, and the entire plane. On the other hand, the sets depicted in Figure 3.4b, any finite set of two or more distinct points, the boundary of any convex polygon, and a circumference are examples of sets that are not convex.

Now we are ready for the notion of the convex hull. Intuitively, the convex hull of a set of n points in the plane is the smallest convex polygon that contains all of them (either inside or on its boundary). If this formulation does not quite fire up your enthusiasm, consider the problem as one of barricading n sleeping tigers by a fence of the shortest length. This interpretation is due to D. Harel [Har92]; it is somewhat lively, however, because the fence posts have to be erected right at the spots where some of the tigers sleep! There is another, much tamer interpretation of this notion. Imagine that the points in question are represented by nails driven into a large sheet of plywood representing the plane. Take a rubber band and stretch it to include all the nails, then let it snap into place. The convex hull is the area bounded by the snapped rubber band (Figure 3.5).

A formal definition of the convex hull that is applicable to arbitrary sets, including sets of points that happen to lie on the same line, follows.

1. By a triangle, rectangle, and, more generally, a convex polygon, we mean here a region, i.e., the set of points both inside and on the boundary of the shape in question.

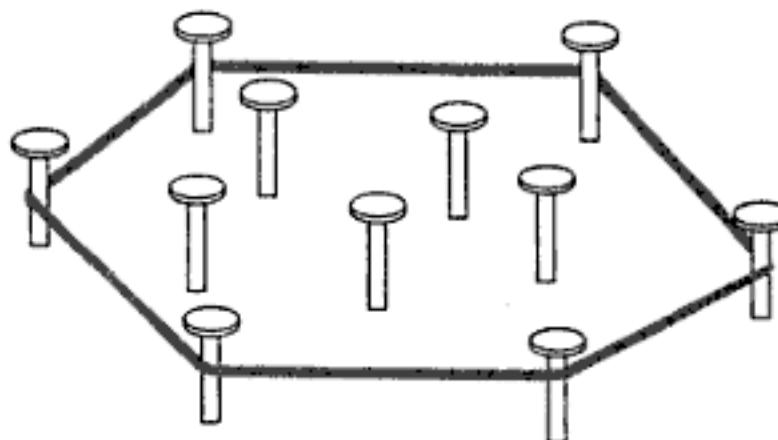


FIGURE 3.5 Rubber-band interpretation of the convex hull

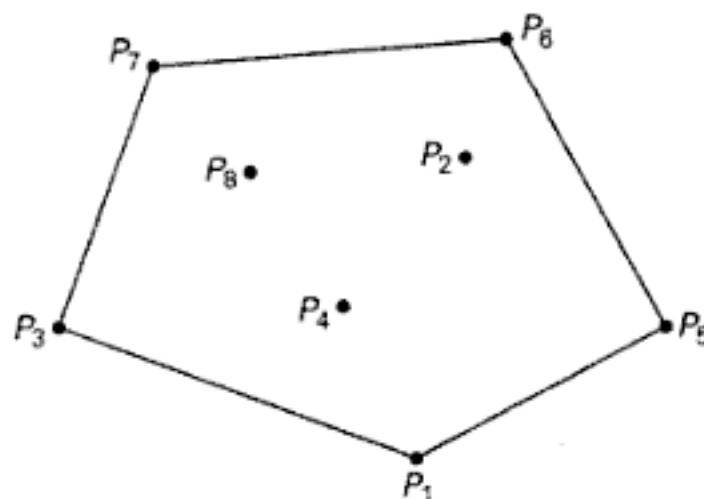


FIGURE 3.6 The convex hull for this set of eight points is the convex polygon with vertices at P_1 , P_5 , P_6 , P_7 , and P_3 .

DEFINITION The *convex hull* of a set S of points is the smallest convex set containing S . (The “smallest” requirement means that the convex hull of S must be a subset of any convex set containing S .)

If S is convex, its convex hull is obviously S itself. If S is a set of two points, its convex hull is the line segment connecting these points. If S is a set of three points not on the same line, its convex hull is the triangle with the vertices at the three points given; if the three points do lie on the same line, the convex hull is the line segment with its endpoints at the two points that are farthest apart. For an example of the convex hull for a larger set, see Figure 3.6.

A study of the examples makes the following theorem an expected result.

THEOREM The convex hull of any set S of $n > 2$ points (not all on the same line) is a convex polygon with the vertices at some of the points of S . (If all the points do lie on the same line, the polygon degenerates to a line segment but still with the endpoints at two points of S .)

The *convex-hull problem* is the problem of constructing the convex hull for a given set S of n points. To solve it, we need to find the points that will serve as the vertices of the polygon in question. Mathematicians call vertices of such a polygon

“extreme points.” By definition, an *extreme point* of a convex set is a point of this set that is not a middle point of any line segment with endpoints in the set. For example, the extreme points of a triangle are its three vertices, the extreme points of a circle are all the points of its circumference, and the extreme points of the convex hull of the set of eight points in Figure 3.6 are P_1, P_5, P_6, P_7 , and P_3 .

Extreme points have several special properties other points of a convex set do not have. One of them is exploited by a very important algorithm called the *simplex method* (Section 10.1). This algorithm solves *linear programming* problems, problems of finding a minimum or a maximum of a linear function of n variables subject to linear constraints (see Problem 10 in Exercises 3.3 for an example and Sections 6.6 and 10.1 for a general discussion). Here, however, we are interested in extreme points because their identification solves the convex-hull problem. Actually, to solve this problem completely, we need to know a bit more than just which of n points of a given set are extreme points of the set’s convex hull: we need to know which pairs of points need to be connected to form the boundary of the convex hull. Note that this issue can also be addressed by listing the extreme points in a clockwise or a counterclockwise order.

So how can we solve the convex-hull problem in a brute-force manner? If you do not see an immediate plan for a frontal attack, do not be dismayed: the convex-hull problem is one with no obvious algorithmic solution. Nevertheless, there is a simple but inefficient algorithm that is based on the following observation about line segments making up a boundary of the convex hull: a line segment connecting two points P_i and P_j of a set of n points is a part of its convex hull’s boundary if and only if all the other points of the set lie on the same side of the straight line through these two points.² (Verify this property for the set in Figure 3.6.) Repeating this test for every pair of points yields a list of line segments that make up the convex hull’s boundary.

A few elementary facts from analytical geometry are needed to implement this algorithm. First, the straight line through two points $(x_1, y_1), (x_2, y_2)$ in the coordinate plane can be defined by the equation

$$ax + by = c,$$

where $a = y_2 - y_1, b = x_1 - x_2, c = x_1y_2 - y_1x_2$.

Second, such a line divides the plane into two half-planes: for all the points in one of them, $ax + by > c$, while for all the points in the other, $ax + by < c$. (For the points on the line itself, of course, $ax + by = c$.) Thus, to check whether certain points lie on the same side of the line, we can simply check whether the expression $ax + by - c$ has the same sign at each of these points. We leave the implementation details as an exercise.

What is the time efficiency of this algorithm? It is in $O(n^3)$: for each of $n(n - 1)/2$ pairs of distinct points, we may need to find the sign of $ax + by - c$ for each of the other $n - 2$ points. There are much more efficient algorithms for this important problem, and we discuss one of them later in the book.

2. For the sake of simplicity, we assume here that no three points of a given set lie on the same line. A modification needed for the general case is left for the exercises.

Exercises 3.3

1. Can you design a faster algorithm than the one based on the brute-force strategy to solve the closest-pair problem for n points x_1, \dots, x_n on the real line?
2. Let $x_1 < x_2 < \dots < x_n$ be real numbers representing coordinates of n villages located along a straight road. A post office needs to be built in one of these villages.
 - a. Design an efficient algorithm to find the post office location minimizing the average distance between the villages and the post office.
 - b. Design an efficient algorithm to find the post office location minimizing the maximum distance from a village to the post office.
3. a. There are several alternative ways to define a distance between two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ in the Cartesian plane. In particular, the so-called ***Manhattan distance*** is defined as

$$d_M(P_1, P_2) = |x_1 - x_2| + |y_1 - y_2|.$$

Prove that d_M satisfies the following axioms which every distance function must satisfy:

- i. $d_M(P_1, P_2) \geq 0$ for any two points P_1 and P_2 , and $d_M(P_1, P_2) = 0$ if and only if $P_1 = P_2$;
 - ii. $d_M(P_1, P_2) = d_M(P_2, P_1)$;
 - iii. $d_M(P_1, P_2) \leq d_M(P_1, P_3) + d_M(P_3, P_2)$ for any P_1, P_2 , and P_3 .
 - b. Sketch all the points in the x, y coordinate plane whose Manhattan distance to the origin $(0,0)$ is equal to 1. Do the same for the Euclidean distance.
 - c. True or false: A solution to the closest-pair problem does not depend on which of the two metrics— d_E (Euclidean) or d_M (Manhattan)—is used?
-  4. *Odd pie fight* There are $n \geq 3$ people positioned on a field (Euclidean plane) so that each has a unique nearest neighbor. Each person has a cream pie. At a signal, everybody hurles his or her pie at the nearest neighbor. Assuming that n is odd and that nobody can miss his or her target, true or false: There always remains at least one person not hit by a pie [Car79]?
5. The closest-pair problem can be posed on the k -dimensional space in which the Euclidean distance between two points $P' = (x'_1, \dots, x'_k)$ and $P'' = (x''_1, \dots, x''_k)$ is defined as

$$d(P', P'') = \sqrt{\sum_{s=1}^k (x'_s - x''_s)^2}.$$

What will be the efficiency class of the brute-force algorithm for the k -dimensional closest-pair problem?

6. Determine the sets which are convex.
 - a. star
 - b. cone
 - c. pentagon
 - d. semi circle
 - e. arc
7. Design a linear-time algorithm to determine the largest co-ordinate of the convex hull of a set of $n > 1$ points in the plane.
8. What modification needs to be made in the brute-force algorithm for the convex-hull problem to handle more than two points on the same straight line?
9. What is the time complexity of the algorithm which can solve the convex-hull problem using the brute-force method?
10. Consider the following small instance of the linear programming problem:

$$\begin{aligned} \text{maximize } & 3x + 5y \\ \text{subject to } & x + y \leq 4 \\ & x + 3y \leq 6 \\ & x \geq 0, y \geq 0. \end{aligned}$$

- a. Sketch, in the Cartesian plane, the problem's **feasible region**, defined as the set of points satisfying all the problem's constraints.
- b. Identify the region's extreme points.
- c. Solve the optimization problem given by using the following theorem: a linear programming problem with a nonempty bounded feasible region always has a solution, which can be found at one of the extreme points of its feasible region.

3.4 Exhaustive Search

Many important problems require finding an element with a special property in a domain that grows exponentially (or faster) with an instance size. Typically, such problems arise in situations that involve—explicitly or implicitly—combinatorial objects such as permutations, combinations, and subsets of a given set. Many such problems are optimization problems: they ask to find an element that maximizes or minimizes some desired characteristic such as a path's length or an assignment's cost.

Exhaustive search is simply a brute-force approach to combinatorial problems. It suggests generating each and every element of the problem's domain, selecting those of them that satisfy all the constraints, and then finding a desired element (e.g., the one that optimizes some objective function). Note that though the idea of exhaustive search is quite straightforward, its implementation typically requires an algorithm for generating certain combinatorial objects. We delay a discussion of such algorithms until Chapter 5 and assume here that they exist. We illustrate exhaustive search by applying it to three important problems: the traveling salesman problem, the knapsack problem, and the assignment problem.

Traveling Salesman Problem

The ***traveling salesman problem (TSP)*** has been intriguing researchers for the last 150 years by its seemingly simple formulation, important applications, and interesting connections to other combinatorial problems. In layman's terms, the problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started. The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once. It is named after the Irish mathematician Sir William Rowan Hamilton (1805–1865), who became interested in such cycles as an application of his algebraic discoveries.)

It is easy to see that a Hamiltonian circuit can be also defined as a sequence of $n + 1$ adjacent vertices $v_{i_0}, v_{i_1}, \dots, v_{i_{n-1}}, v_{i_0}$, where the first vertex of the sequence is the same as the last one while all the other $n - 1$ vertices are distinct. Further, we can assume, with no loss of generality, that all circuits start and end at one particular vertex (they are cycles after all, are they not?). Thus, we can get all the tours by generating all the permutations of $n - 1$ intermediate cities, compute the tour lengths, and find the shortest among them. Figure 3.7 presents a small instance of the problem and its solution by this method.

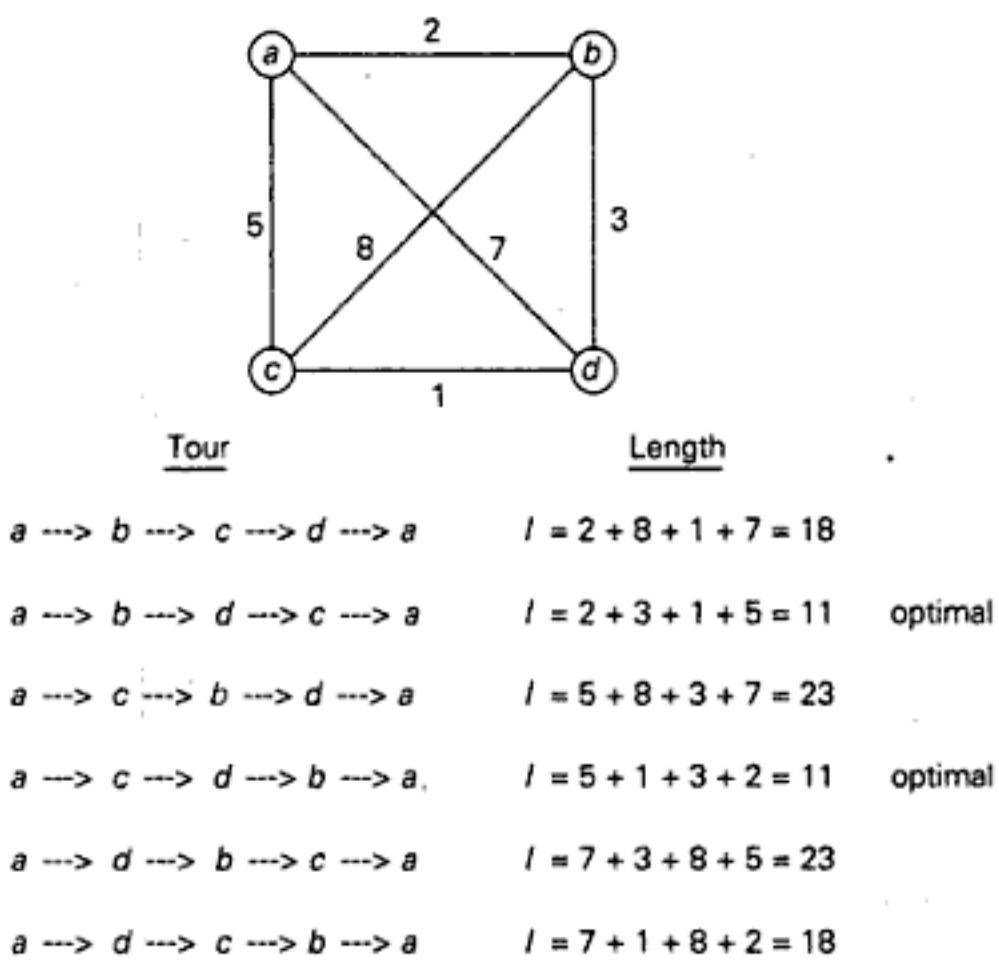


FIGURE 3.7 Solution to a small instance of the traveling salesman problem by exhaustive search

An inspection of Figure 3.7 reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by half. We could, for example, choose any two intermediate vertices, say, B and C , and then consider only permutations in which B precedes C . (This trick implicitly defines a tour's direction.)

This improvement cannot brighten the efficiency picture much, however. The total number of permutations needed will still be $(n - 1)!/2$, which makes the exhaustive-search approach impractical for all but very small values of n . On the other hand, if you always see your glass as half-full, you can claim that cutting the work by half is nothing to sneeze at, even if you solve a small instance of the problem, especially by hand. Also note that had we not limited our investigation to the circuits starting at the same vertex, the number of permutations would have been even larger, by a factor of n .

Knapsack Problem

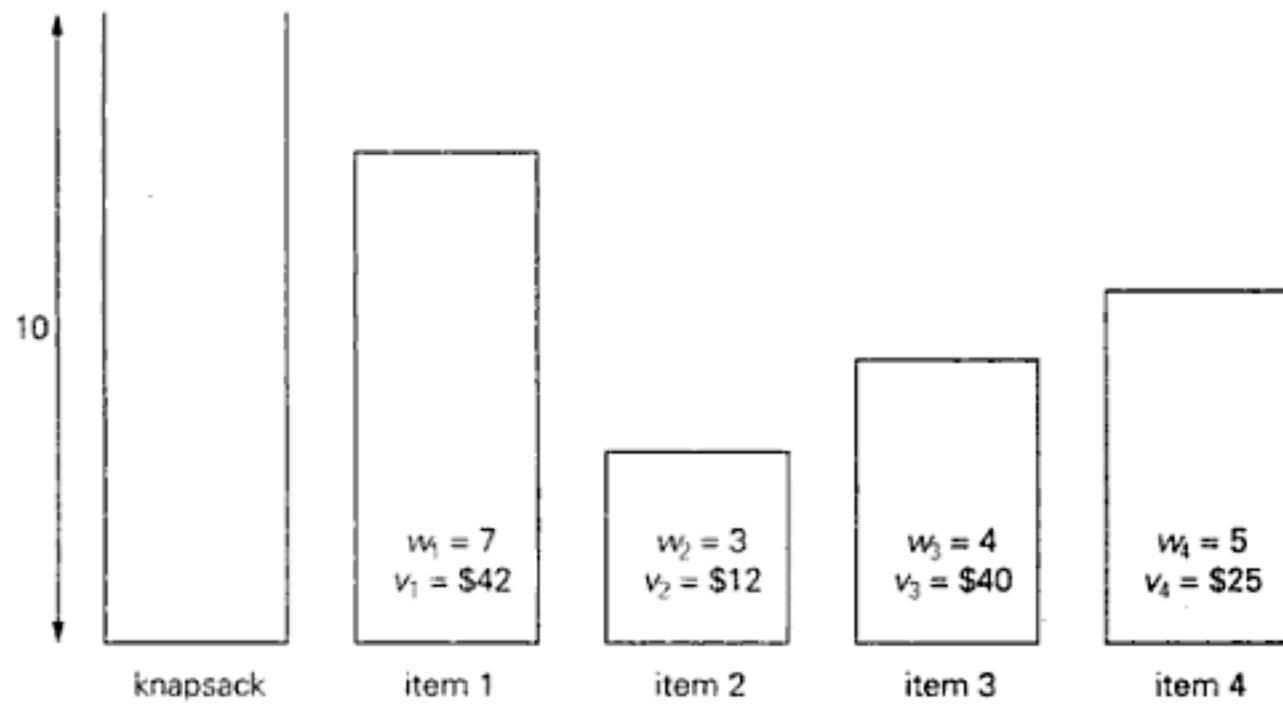
Here is another well-known problem in algorithmics. Given n items of known weights w_1, \dots, w_n and values v_1, \dots, v_n and a knapsack of capacity W , find the most valuable subset of the items that fit into the knapsack. If you do not like the idea of putting yourself in the shoes of a thief who wants to steal the most valuable loot that fits into his knapsack, think about a transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity. Figure 3.8a presents a small instance of the knapsack problem.

The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack's capacity), and finding a subset of the largest value among them. As an example, the solution to the instance of Figure 3.8a is given in Figure 3.8b. Since the number of subsets of an n -element set is 2^n , the exhaustive search leads to a $\Omega(2^n)$ algorithm no matter how efficiently individual subsets are generated.

Thus, for both the traveling salesman and knapsack problems, exhaustive search leads to algorithms that are extremely inefficient on every input. In fact, these two problems are the best-known examples of so-called ***NP-hard problems***. No polynomial-time algorithm is known for any *NP-hard* problem. Moreover, most computer scientists believe that such algorithms do not exist, although this very important conjecture has never been proven. More sophisticated approaches—backtracking and branch-and-bound (see Sections 12.1 and 12.2)—enable us to solve some but not all instances of these (and similar) problems in less than exponential time. Alternatively, we can use one of many approximation algorithms, such as those described in Section 12.3.

Assignment Problem

In our third example of a problem that can be solved by exhaustive search, there are n people who need to be assigned to execute n jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the i th person is assigned to the j th job



(a)

Subset	Total weight	Total value
\emptyset	0	\$ 0
{1}	7	\$42
{2}	3	\$12
{3}	4	\$40
{4}	5	\$25
{1, 2}	10	\$36
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	\$52
{2, 4}	8	\$37
{3, 4}	9	\$65
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

FIGURE 3.8 (a) Instance of the knapsack problem. (b) Its solution by exhaustive search. (The information about the optimal selection is in bold.)

is a known quantity $C[i, j]$ for each pair $i, j = 1, 2, \dots, n$. The problem is to find an assignment with the minimum total cost.

A small instance of this problem follows, with the table entries representing the assignment costs $C[i, j]$:

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

It is easy to see that an instance of the assignment problem is completely specified by its cost matrix C . In terms of this matrix, the problem calls for a selection of one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible. Note that no obvious strategy for finding a solution works here. For example, we cannot select the smallest element in each row because the smallest elements may happen to be in the same column. In fact, the smallest element in the entire matrix need not be a component of an optimal solution. Thus, opting for the exhaustive search may appear as an unavoidable evil.

We can describe feasible solutions to the assignment problem as n -tuples $\langle j_1, \dots, j_n \rangle$ in which the i th component, $i = 1, \dots, n$, indicates the column of the element selected in the i th row (i.e., the job number assigned to the i th person). For example, for the cost matrix above, $\langle 2, 3, 4, 1 \rangle$ indicates a feasible assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, and Person 4 to Job 1. The requirements of the assignment problem imply that there is a one-to-one correspondence between feasible assignments and permutations of the first n integers. Therefore, the exhaustive-search approach to the assignment problem would require generating all the permutations of integers $1, 2, \dots, n$, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum. A few first iterations of applying this algorithm to the instance given above are shown in Figure 3.9; you are asked to complete it in the exercises.

Since the number of permutations to be considered for the general case of the assignment problem is $n!$, exhaustive search is impractical for all but very small instances of the problem. Fortunately, there is a much more efficient algorithm for

$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$	$\langle 1, 2, 3, 4 \rangle$	$\text{cost} = 9 + 4 + 1 + 4 = 18$
	$\langle 1, 2, 4, 3 \rangle$	$\text{cost} = 9 + 4 + 8 + 9 = 30$
	$\langle 1, 3, 2, 4 \rangle$	$\text{cost} = 9 + 3 + 8 + 4 = 24$
	$\langle 1, 3, 4, 2 \rangle$	$\text{cost} = 9 + 3 + 8 + 6 = 26$
	$\langle 1, 4, 2, 3 \rangle$	$\text{cost} = 9 + 7 + 8 + 9 = 33$
	$\langle 1, 4, 3, 2 \rangle$	$\text{cost} = 9 + 7 + 1 + 6 = 23$
		etc.

FIGURE 3.9 First few iterations of solving a small instance of the assignment problem by exhaustive search

this problem called the **Hungarian method** after the Hungarian mathematicians König and Egerváry whose work underlies the method (see, e.g., [Kol95]).

This is good news: the fact that a problem's domain grows exponentially (or faster) does not necessarily imply that there can be no efficient algorithm for solving it. In fact, we present several other examples of such problems later in the book. However, such examples are more of an exception from the rule. More often than not, there are no known polynomial-time algorithms for problems whose domain grows exponentially with instance size (provided we want to solve them exactly). And, as mentioned, such algorithms quite possibly do not exist.

Exercises 3.4

1. a. Assuming that each tour can be generated in constant time, what will be the efficiency class of the exhaustive-search algorithm outlined in the text for the traveling salesman problem?
 - i. one hour.
 - ii. 24-hours.
 - iii. one year.
 - iv. one century.
2. Write an efficient and exhaustive search algorithm for the traveling salesmen problem.
3. Outline an algorithm to determine whether a connected graph represented by its adjacency matrix has a Eulerian circuit. What is the efficiency class of your algorithm?
4. Find the optimal solution for the assignment problem given below.

	Job 1	Job 2	Job 3	Job 4
Person 1	4	3	8	6
Person 2	5	7	2	4
Person 3	16	9	3	1
Person 4	2	5	3	7

5. Give examples of assignment problems where
 - a. The largest element of cost matrix is not included in the optimal solution.
 - b. The largest element of cost matrix is not feasible for optimal solution.
6. Consider the **partition problem**: given n positive integers, partition them into two disjoint subsets with the same sum of their elements. (Of course, the problem does not always have a solution.) Design an exhaustive-search algorithm

for this problem. Try to minimize the number of subsets the algorithm needs to generate.

7. Consider the *clique problem*: given a graph G and a positive integer k , determine whether the graph contains a *clique* of size k , i.e., a complete subgraph of k vertices. Design an exhaustive-search algorithm for this problem.
8. Explain how exhaustive search can be applied to the sorting problem and determine the efficiency class of such an algorithm.
9.  *Magic squares* A magic square of order n is an arrangement of the numbers from 1 to n^2 in an n -by- n matrix, with each number occurring exactly once, so that each row, each column, and each main diagonal has the same sum.
 - a. Prove that if a magic square of order n exists, the sum in question must be equal to $n(n^2 + 1)/2$.
 - b. Design an exhaustive-search algorithm for generating all magic squares of order n .
 - c. Go to the Internet or your library and find a better algorithm for generating magic squares.
 - d. Implement the two algorithms—the exhaustive search and the one you have found—and run an experiment to determine the largest value of n for which each of the algorithms is able to find a magic square of order n in less than one minute of your computer's time.
10.  *Famous alphametic* A puzzle in which the digits in a correct mathematical expression, such as a sum, are replaced by letters is called a *cryptarithm*; if, in addition, the puzzle's words make sense, it is said to be an *alphametic*. The most well-known alphametic was published by renowned British puzzlist H. E. Dudeney (1857–1930):

$$\begin{array}{r}
 & S & E & N & D \\
 + & M & O & R & E \\
 \hline
 M & O & N & E & Y
 \end{array}$$

Two conditions are assumed: first, the correspondence between letters and decimal digits is one-to-one, i.e., each letter represents one digit only and different letters represent different digits; second, the digit zero does not appear as the left-most digit in any of the numbers. To solve an alphametic means to find which digit each letter represents. Note that a solution's uniqueness cannot be assumed and has to be verified by the solver.

- a. Write a program for solving cryptarithms by exhaustive search. Assume that a given cryptarithm is a sum of two words.
- b. Solve Dudeney's puzzle the way it was expected to be solved when it was first published in 1924.

SUMMARY

- *Brute force* is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.
- The principal strengths of the brute-force approach are wide applicability and simplicity; its principal weakness is the subpar efficiency of most brute-force algorithms.
- A first application of the brute-force approach often results in an algorithm that can be improved with a modest amount of effort.
- The following noted algorithms can be considered as examples of the brute-force approach:
 - definition-based algorithm for matrix multiplication;
 - *selection sort*;
 - *sequential search*;
 - straightforward string-matching algorithm.
- *Exhaustive search* is a brute-force approach to combinatorial problems. It suggests generating each and every combinatorial object of the problem, selecting those of them that satisfy all the constraints, and then finding a desired object.
- The *traveling salesman problem*, the *knapsack problem*, and the *assignment problem* are typical examples of problems that can be solved, at least theoretically, by exhaustive-search algorithms.
- Exhaustive search is impractical for all but very small instances of problems it can be applied to.

4

Divide-and-Conquer

Whatever man prays for, he prays for a miracle. Every prayer reduces itself to this—Great God, grant that twice two be not four.

—Ivan Turgenev (1818–1883), Russian novelist and short-story writer

Divide-and-conquer is probably the best-known general algorithm design technique. Though its fame may have something to do with its catchy name, it is well deserved: quite a few very efficient algorithms are specific implementations of this general strategy. Divide-and-conquer algorithms work according to the following general plan:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved (typically recursively, though sometimes a different algorithm is employed when instances become small enough).
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

The divide-and-conquer technique is diagrammed in Figure 4.1, which depicts the case of dividing a problem into two smaller subproblems, by far the most widely occurring case (at least for divide-and-conquer algorithms designed to be executed on a single-processor computer).

As an example, let us consider the problem of computing the sum of n numbers a_0, \dots, a_{n-1} . If $n > 1$, we can divide the problem into two instances of the same problem: to compute the sum of the first $\lfloor n/2 \rfloor$ numbers and to compute the sum of the remaining $\lceil n/2 \rceil$ numbers. (Of course, if $n = 1$, we simply return a_0 as the answer.) Once each of these two sums is computed (by applying the same method, i.e., recursively), we can add their values to get the sum in question:

$$a_0 + \dots + a_{n-1} = (a_0 + \dots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \dots + a_{n-1}).$$

Is this an efficient way to compute the sum of n numbers? A moment of reflection (why could it be more efficient than the brute-force summation?), a small example of summing, say, four numbers by this algorithm, a formal analysis

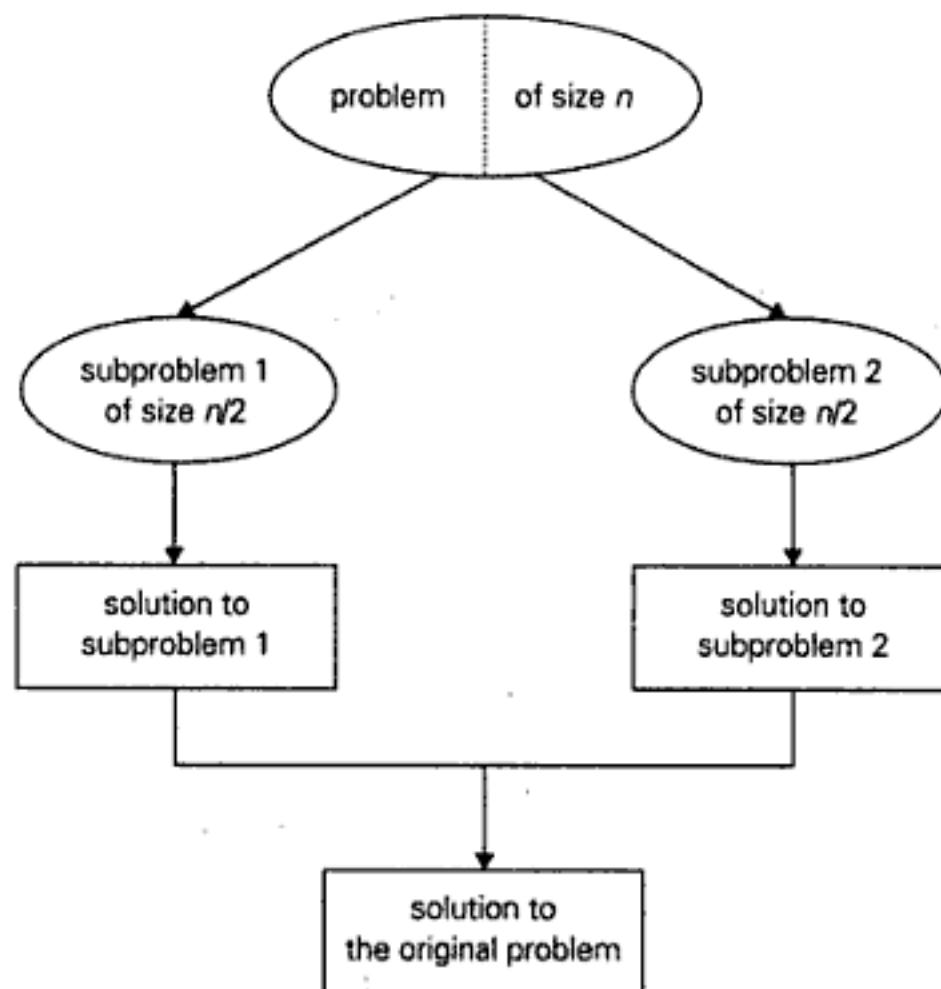


FIGURE 4.1 Divide-and-conquer technique (typical case)

(which follows), and common sense (we do not compute sums this way, do we?) all lead to a negative answer to this question.

Thus, not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution. But often our prayers to the Goddess of Algorithmics—see the chapter’s epigraph—are answered, and the time spent on executing the divide-and-conquer plan turns out to be smaller than solving a problem by a different method. In fact, the divide-and-conquer approach yields some of the most important and efficient algorithms in computer science. We discuss a few classic examples of such algorithms in this chapter. Though we consider only sequential algorithms here, it is worth keeping in mind that the divide-and-conquer technique is ideally suited for parallel computations, in which each subproblem can be solved simultaneously by its own processor.

As mentioned above, in the most typical case of divide-and-conquer, a problem’s instance of size n is divided into two instances of size $n/2$. More generally, an instance of size n can be divided into b instances of size n/b , with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.) Assuming that size n is a power of b , to simplify our analysis, we get the following recurrence for the running time $T(n)$:

$$T(n) = aT(n/b) + f(n), \quad (4.1)$$

where $f(n)$ is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions. (For the summation example,

$a = b = 2$ and $f(n) = 1$.) Recurrence (4.1) is called the **general divide-and-conquer recurrence**. Obviously, the order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$. The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem (see Appendix B).

THEOREM (Master Theorem) If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence equation (4.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

(Analogous results hold for the O and Ω notations, too.)

For example, the recurrence equation for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm (see above) on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2$, $b = 2$, and $d = 0$; hence, since $a > b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Note that we were able to find the solution's efficiency class without going through the drudgery of solving the recurrence. But, of course, this approach can only establish a solution's order of growth to within an unknown multiplicative constant, while solving a recurrence equation with a specific initial condition yields an exact answer (at least for n 's that are powers of b).

4.1 Mergesort

Mergesort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array $A[0..n - 1]$ by dividing it into two halves $A[0..[n/2] - 1]$ and $A[[n/2]..n - 1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

ALGORITHM *Mergesort(A[0..n - 1])*

```
//Sorts array A[0..n - 1] by recursive mergesort
//Input: An array A[0..n - 1] of orderable elements
//Output: Array A[0..n - 1] sorted in nondecreasing order
if n > 1
    copy A[0..[n/2] - 1] to B[0..[n/2] - 1]
    copy A[[n/2]..n - 1] to C[0..[n/2] - 1]
    Mergesort(B[0..[n/2] - 1])
    Mergesort(C[0..[n/2] - 1])
    Merge(B, C, A)
```

The *merging* of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

```
ALGORITHM Merge( $B[0..p - 1]$ ,  $C[0..q - 1]$ ,  $A[0..p + q - 1]$ )
  //Merges two sorted arrays into one sorted array
  //Input: Arrays  $B[0..p - 1]$  and  $C[0..q - 1]$  both sorted
  //Output: Sorted array  $A[0..p + q - 1]$  of the elements of  $B$  and  $C$ 
   $i \leftarrow 0$ ;  $j \leftarrow 0$ ;  $k \leftarrow 0$ 
  while  $i < p$  and  $j < q$  do
    if  $B[i] \leq C[j]$ 
       $A[k] \leftarrow B[i]$ ;  $i \leftarrow i + 1$ 
    else  $A[k] \leftarrow C[j]$ ;  $j \leftarrow j + 1$ 
     $k \leftarrow k + 1$ 
  if  $i = p$ 
    copy  $C[j..q - 1]$  to  $A[k..p + q - 1]$ 
  else copy  $B[i..p - 1]$  to  $A[k..p + q - 1]$ 
```

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in Figure 4.2.

How efficient is mergesort? Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons $C(n)$ is

$$C(n) = 2C(n/2) + C_{\text{merge}}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

Let us analyze $C_{\text{merge}}(n)$, the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needed to be processed is reduced by one. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{\text{merge}}(n) = n - 1$, and we have the recurrence

$$C_{\text{worst}}(n) = 2C_{\text{worst}}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{\text{worst}}(1) = 0.$$

Hence, according to the Master Theorem, $C_{\text{worst}}(n) \in \Theta(n \log n)$ (why?). In fact, it is easy to find the exact solution to the worst-case recurrence for $n = 2^k$:

$$C_{\text{worst}}(n) = n \log_2 n - n + 1.$$

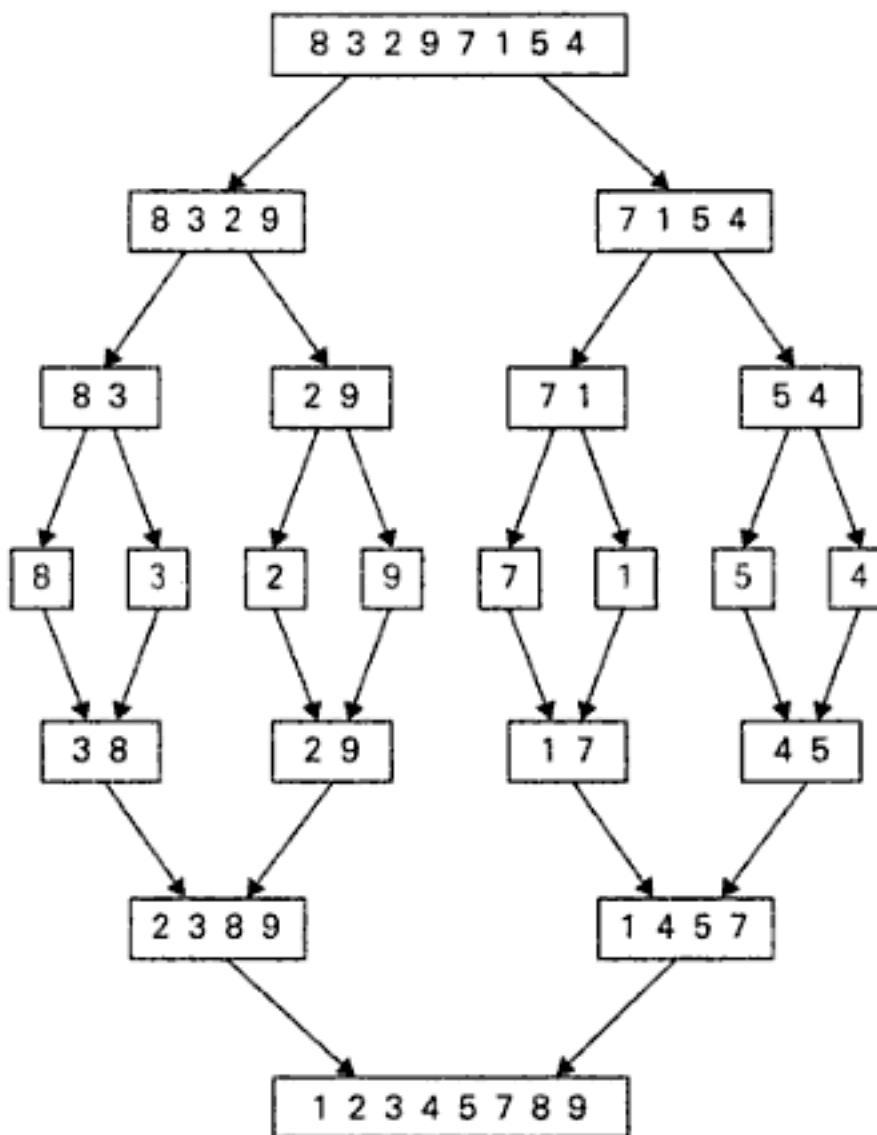


FIGURE 4.2 Example of mergesort operation

The number of key comparisons made by mergesort in the worst case comes very close to the theoretical minimum¹ that any general comparison-based sorting algorithm can have. The principal shortcoming of mergesort is the linear amount of extra storage the algorithm requires. Though merging can be done in place, the resulting algorithm is quite complicated and, since it has a significantly larger multiplicative constant, the in-place mergesort is of theoretical interest only.

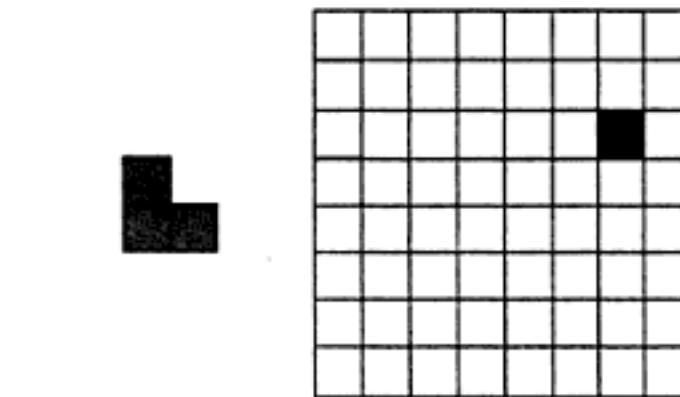
Exercises 4.1

1. Write a pseudocode for the divide-and-conquer algorithm for finding the position of an element in an array of n numbers and estimate the number of key comparisons made by your algorithm.
2. a. Write a pseudocode for a divide-and-conquer algorithm for finding values of both the largest and smallest elements in an array of n numbers.

1. As we shall see in Section 11.2, this theoretical minimum is $\lceil \log_2 n! \rceil \approx \lceil n \log_2 n - 1.44n \rceil$.

- b. Set up and solve (for $n = 2^k$) a recurrence relation for the number of key comparisons made by your algorithm.
 - c. How does this algorithm compare with the brute-force algorithm for this problem?
3. a. Write a pseudocode for a divide-and-conquer algorithm for the exponentiation problem of computing a^n where $a > 0$ and n is a positive integer.
- b. Set up and solve a recurrence relation for the number of multiplications made by this algorithm.
 - c. How does this algorithm compare with the brute-force algorithm for this problem?
4. We mentioned in Chapter 2, that logarithm bases are irrelevant in most contexts arising in the analysis of an algorithm's efficiency class. Is this true for both assertions of the Master Theorem that include logarithms?
5. Find the order of growth for solutions of the following recurrences.
- a. $T(n) = 4T(n/2) + n$, $T(1) = 1$
 - b. $T(n) = 4T(n/2) + n^2$, $T(1) = 1$
 - c. $T(n) = 4T(n/2) + n^3$, $T(1) = 1$
6. Apply mergesort to sort the list E, X, A, M, P, L, E in alphabetical order.
7. Using an example for mergesort, sketch the divide-and-conquer technique. State if the mergesort is a stable sorting algorithm.
8. a. Solve the recurrence relation for the number of key comparisons made by mergesort in the worst case. (You may assume that $n = 2^k$.)
- b. Set up a recurrence relation for the number of key comparisons made by mergesort on best-case inputs and solve it for $n = 2^k$.
 - c. Set up a recurrence relation for the number of key moves made by the version of mergesort given in Section 4.1. Does taking the number of key moves into account change the algorithm's efficiency class?
9. Let $A\{i, \dots, n\}$ be an array of n distinct real numbers. A pair $([A[i], A[j]])$ is said to be an **index-value inversion** if these numbers are such that $A[i] = j$ and $A[j] = i$. Design an algorithm for counting the number of index-value inversions.
10. One can implement mergesort without a recursion by starting with merging adjacent elements of a given array, then merging sorted pairs, and so on. Implement this bottom-up version of mergesort in the language of your choice.
11. *Tromino puzzle* A tromino is an L-shaped tile formed by 1-by-1 adjacent squares. The problem is to cover any 2^n -by- 2^n chessboard with one missing square (anywhere on the board) with trominos. Trominos should cover all the squares except the missing one with no overlaps.





Design a divide-and-conquer algorithm for this problem.

4.2 Quicksort

Quicksort is another important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input's elements according to their position in the array, quicksort divides them according to their value. Specifically, it rearranges elements of a given array $A[0..n - 1]$ to achieve its **partition**, a situation where all the elements before some position s are smaller than or equal to $A[s]$ and all the elements after position s are greater than or equal to $A[s]$:

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are } \geq A[s]}$$

Obviously, after a partition has been achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two subarrays of the elements preceding and following $A[s]$ independently (e.g., by the same method).

ALGORITHM *Quicksort(A[l..r])*

```

//Sorts a subarray by quicksort
//Input: A subarray  $A[l..r]$  of  $A[0..n - 1]$ , defined by its left and right indices
// l and r
//Output: Subarray  $A[l..r]$  sorted in nondecreasing order
if  $l < r$ 
     $s \leftarrow \text{Partition}(A[l..r])$  //s is a split position
    Quicksort(A[l..s - 1])
    Quicksort(A[s + 1..r])

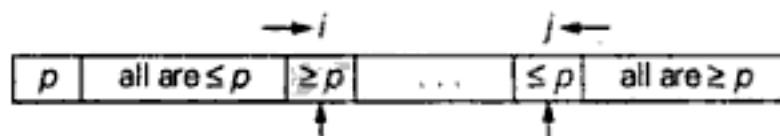
```

A partition of $A[0..n - 1]$ and, more generally, of its subarray $A[l..r]$ ($0 \leq l < r \leq n - 1$) can be achieved by the following algorithm. First, we select an element with respect to whose value we are going to divide the subarray. Because of its guiding role, we call this element the **pivot**. There are several different strategies for selecting a pivot; we will return to this issue when we analyze the algorithm's

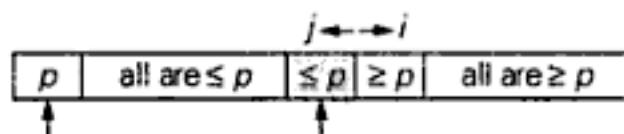
efficiency. For now, we use the simplest strategy of selecting the subarray's first element: $p = A[l]$.

There are also several alternative procedures for rearranging elements to achieve a partition. Here we use an efficient method based on two scans of the subarray: one is left-to-right and the other right-to-left, each comparing the subarray's elements with the pivot. The left-to-right scan, denoted below by index i , starts with the second element. Since we want elements smaller than the pivot to be in the first part of the subarray, this scan skips over elements that are smaller than the pivot and stops on encountering the first element greater than or equal to the pivot. The right-to-left scan, denoted below by index j , starts with the last element of the subarray. Since we want elements larger than the pivot to be in the second part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

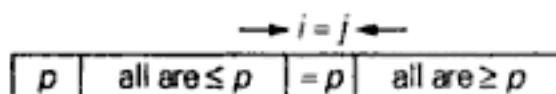
After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume the scans by incrementing i and decrementing j , respectively:



If the scanning indices have crossed over, i.e., $i > j$, we will have partitioned the array after exchanging the pivot with $A[j]$:



Finally, if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p (why?). Thus, we have the array partitioned, with the split position $s = i = j$:



We can combine the last case with the case of crossed-over indices ($i > j$) by exchanging the pivot with $A[j]$ whenever $i \geq j$.

Here is a pseudocode implementing this partitioning procedure.

ALGORITHM *Partition(A[l..r])*

```
//Partitions a subarray by using its first element as a pivot
//Input: A subarray A[l..r] of A[0..n - 1], defined by its left and right
// indices l and r ( $l < r$ )
//Output: A partition of A[l..r], with the split position returned as
// this function's value
```

```

 $p \leftarrow A[l]$ 
 $i \leftarrow l; j \leftarrow r + 1$ 
repeat
    repeat  $i \leftarrow i + 1$  until  $A[i] \geq p$ 
    repeat  $j \leftarrow j - 1$  until  $A[j] \leq p$ 
    swap( $A[i], A[j]$ )
until  $i \geq j$ 
swap( $A[i], A[j]$ ) //undo last swap when  $i \geq j$ 
swap( $A[l], A[j]$ )
return  $j$ 

```

Note that index i can go out of the subarray bounds in this pseudocode. Rather than checking for this possibility every time index i is incremented, we can append to array $A[0..n-1]$ a “sentinel” that would prevent index i from advancing beyond position n . The more sophisticated method of pivot selection mentioned at the end of the section makes such a sentinel unnecessary.

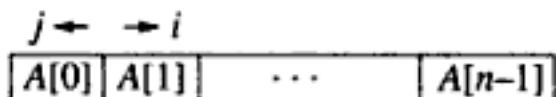
An example of sorting an array by quicksort is given in Figure 4.3.

We start our discussion of quicksort’s efficiency by noting that the number of key comparisons made before a partition is achieved is $n + 1$ if the scanning indices cross over, n if they coincide (why?). If all the splits happen in the middle of corresponding subarrays, we will have the best case. The number of key comparisons in the best case will satisfy the recurrence

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, while the size of the other will be just one less than the size of the subarray being partitioned. This unfortunate situation will happen, in particular, for increasing arrays, i.e., for inputs for which the problem is already solved! Indeed, if $A[0..n-1]$ is a strictly increasing array and we use $A[0]$ as the pivot, the left-to-right scan will stop on $A[1]$ while the right-to-left scan will go all the way to reach $A[0]$, indicating the split at position 0:



So, after making $n + 1$ comparisons to get to this partition and exchanging the pivot $A[0]$ with itself, the algorithm will find itself with the strictly increasing array $A[1..n-1]$ to sort. This sorting of strictly increasing arrays of diminishing sizes will continue until the last one $A[n-2..n-1]$ has been processed. The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n + 1) + n + \cdots + 3 = \frac{(n + 1)(n + 2)}{2} - 3 \in \Theta(n^2).$$

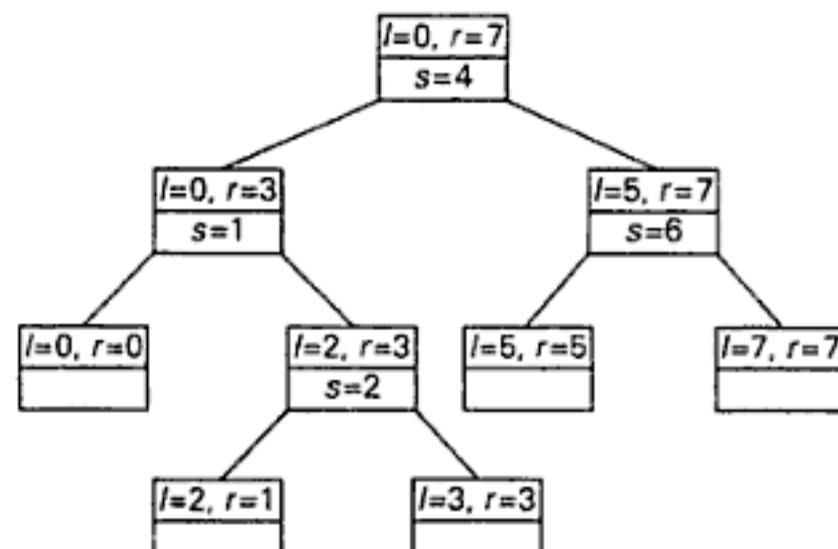
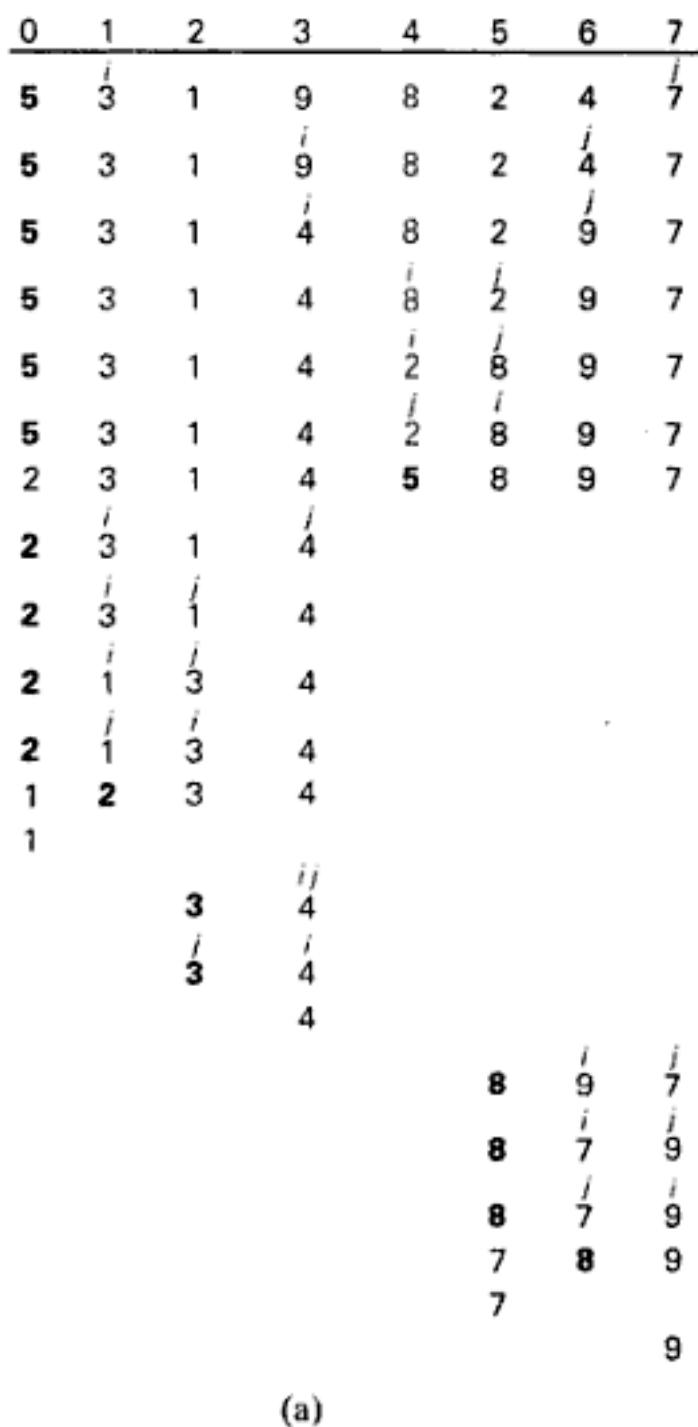


FIGURE 4.3 Example of Quicksort operation. (a) The array's transformations with pivots shown in bold. (b) The tree of recursive calls to *Quicksort* with input values *l* and *r* of subarray bounds and split position *s* of a partition obtained.

Thus, the question about the utility of quicksort comes to its average-case behavior. Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n . Assuming that the partition split can happen in each position s ($0 \leq s \leq n - 1$) with the same probability $1/n$, we get the following recurrence relation

$$C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} [(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

Though solving this recurrence is easier than one might expect, it is still much trickier than the worst- and best-case analyses, and we will leave it for the exercises. Its solution turns out to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.38n \log_2 n.$$

Thus, on the average, quicksort makes only 38% more comparisons than in the best case. Moreover, its innermost loop is so efficient that it runs faster than mergesort (and heapsort, another $n \log n$ algorithm that we discuss in Chapter 6) on randomly ordered arrays, justifying the name given to the algorithm by its inventor, the prominent British computer scientist C.A.R. Hoare.²

Given the importance of quicksort, there have been persistent efforts over the years to refine the basic algorithm. Among several improvements discovered by researchers are: better pivot selection methods (such as the *median-of-three partitioning* that uses as a pivot the median of the leftmost, rightmost, and the middle element of the array); switching to a simpler sort on smaller subfiles; and recursion elimination (so-called nonrecursive quicksort). According to R. Sedgewick [Sed98], the world's leading expert on quicksort, these improvements in combination can cut the running time of the algorithm by 20%–25%.

We should also point out that the idea of partitioning can be useful in applications other than sorting. In particular, it underlines a fast algorithm for the important *selection problem* discussed in Section 5.6.

Exercises 4.2

1. Apply quicksort to sort the list

M, E, R, G, E, S, O, R, T

in alphabetical order. Find the element whose position is unchanged in the sorted list.

2. For the partitioning procedure outlined in Section 4.2:
 - a. Prove that if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p .
 - b. Prove that when the scanning indices stop, j cannot point to an element more than one position to the left of the one pointed to by i .
 - c. Why is it worth stopping the scans after encountering an element equal to the pivot?

2. The young Hoare invented his algorithm while trying to sort words of a Russian dictionary for a machine translation project from Russian to English. Says Hoare, "My first thought on how to do this was bubblesort and, by an amazing stroke of luck, my second thought was Quicksort." It is hard to disagree with his overall assessment: "I have been very lucky. What a wonderful way to start a career in Computing, by discovering a new sorting algorithm!" [Hoa96]

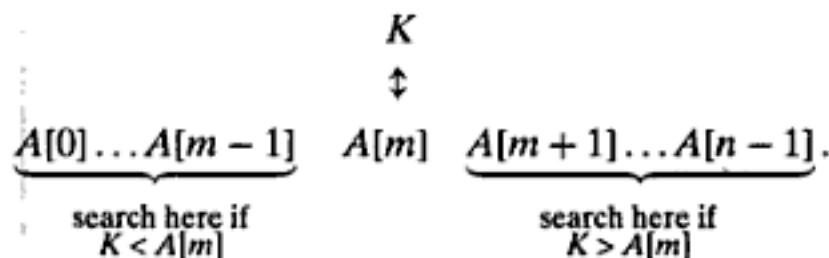
3. Compare the stabilities and the time complexities of quicksort and mergesort algorithms.
4. Give an example of an array of n elements for which the sentinel mentioned in the text is actually needed. What should be its value? Also explain why a single sentinel suffices for any input.
5. For the version of quicksort given in the text:
 - a. Are arrays made up of all equal elements the worst-case input, the best-case input, or neither?
 - b. Are strictly decreasing arrays the worst-case input, the best-case input, or neither?
6. a. For quicksort with the median-of-three pivot selection, are increasing arrays the worst-case input, the best-case input, or neither?
b. Answer the same question for decreasing arrays.
7. Solve the average-case recurrence for quicksort.
8. Design an algorithm to rearrange an array of n real numbers so that all the elements are in decreasing order (For example: $A[1] > A[2] > A[3]$) in a time-efficient manner.
9. The **Dutch flag problem** is to rearrange any array of characters *R*, *W*, and *B* (red, white, and blue are the colors of the Dutch national flag) so that all the *R*'s come first, the *W*'s come next, and the *B*'s come last. Design a linear in-place algorithm for this problem.
10. Implement quicksort in the language of your choice. Run your program on a sample of inputs to verify the theoretical assertions about the algorithm's efficiency.
11. **Nuts and bolts** You are given a collection of n bolts of different widths and n corresponding nuts. You are allowed to try a nut and bolt together, from which you can determine whether the nut is larger than the bolt, smaller than the bolt, or matches the bolt exactly. However, there is no way to compare two nuts together or two bolts together. The problem is to match each bolt to its nut. Design an algorithm for this problem with average-case efficiency in $\Theta(n \log n)$. [Raw91], p. 293



4.3 Binary Search

Binary search is a remarkably efficient algorithm for searching in a sorted array. It works by comparing a search key K with the array's middle element $A[m]$. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if $K < A[m]$, and for the second half

if $K > A[m]$:



As an example, let us apply binary search to searching for $K = 70$ in the array

3	14	27	31	39	42	55	70	74	81	85	93	98
---	----	----	----	----	----	----	----	----	----	----	----	----

The iterations of the algorithm are given in the following table:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1								<i>m</i>					<i>r</i>
iteration 2									<i>l</i>	<i>m</i>			<i>r</i>
iteration 3									<i>l, m</i>	<i>r</i>			

Though binary search is clearly based on a recursive idea, it can be easily implemented as a nonrecursive algorithm, too. Here is a pseudocode for this nonrecursive version.

ALGORITHM *BinarySearch(A[0..n - 1], K)*

```

//Implements nonrecursive binary search
//Input: An array  $A[0..n - 1]$  sorted in ascending order and
//       a search key  $K$ 
//Output: An index of the array's element that is equal to  $K$ 
//       or  $-1$  if there is no such element
l  $\leftarrow 0$ ; r  $\leftarrow n - 1$ 
while l  $\leq r$  do
    m  $\leftarrow \lfloor (l + r)/2 \rfloor$ 
    if  $K = A[m]$  return m
    else if  $K < A[m]$  r  $\leftarrow m - 1$ 
    else l  $\leftarrow m + 1$ 
return  $-1$ 

```

The standard way to analyze the efficiency of binary search is to count the number of times the search key is compared with an element of the array. Moreover, for the sake of simplicity, we will count the so-called three-way comparisons. This

assumes that after one comparison of K with $A[m]$, the algorithm can determine whether K is smaller, equal to, or larger than $A[m]$.

How many such comparisons does the algorithm make on an array of n elements? The answer obviously depends not only on n but also on the specifics of a particular instance of the problem. Let us find the number of key comparisons in the worst case $C_{\text{worst}}(n)$. The worst-case inputs include all arrays that do not contain a given search key (and, in fact, some cases of successful searches as well). Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for $C_{\text{worst}}(n)$:

$$C_{\text{worst}}(n) = C_{\text{worst}}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{\text{worst}}(1) = 1. \quad (4.2)$$

(Stop and convince yourself that $n/2$ must be, indeed, rounded down and that the initial condition must be written as specified.)

As we discussed in Section 2.4, the standard way of solving recurrences such as recurrence (4.2) is to assume that $n = 2^k$ and solve the resulting recurrence by backward substitutions or another method. We leave this as a straightforward exercise to obtain the solution

$$C_{\text{worst}}(2^k) = k + 1 = \log_2 n + 1. \quad (4.3)$$

Actually, one can prove that the solution given by formula (4.3) for $n = 2^k$ can be tweaked to get a solution valid for an arbitrary positive integer n :

$$C_{\text{worst}}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil. \quad (4.4)$$

Let us verify by substitution that $C_{\text{worst}}(n) = \lfloor \log_2 n \rfloor + 1$ indeed satisfies equation (4.2) for any positive even number n . (You are asked to do this for odd n 's in Exercises 4.3). If n is positive and even, $n = 2i$ where $i > 0$. The left-hand side of equation (4.2) for $n = 2i$ is

$$\begin{aligned} C_{\text{worst}}(n) &= \lfloor \log_2 n \rfloor + 1 = \lfloor \log_2 2i \rfloor + 1 = \lfloor \log_2 2 + \log_2 i \rfloor + 1 \\ &= (1 + \lfloor \log_2 i \rfloor) + 1 = \lfloor \log_2 i \rfloor + 2. \end{aligned}$$

The right-hand side of equation (4.2) for $n = 2i$ is

$$\begin{aligned} C_{\text{worst}}(\lfloor n/2 \rfloor) + 1 &= C_{\text{worst}}(\lfloor 2i/2 \rfloor) + 1 = C_{\text{worst}}(i) + 1 \\ &= (\lfloor \log_2 i \rfloor + 1) + 1 = \lfloor \log_2 i \rfloor + 2. \end{aligned}$$

Since both expressions are the same, we proved the assertion.

Formula (4.4) deserves attention. First, it implies that the worst-case efficiency of binary search is in $\Theta(\log n)$. (Incidentally, we could get this fact by applying the Master Theorem, but this approach would not give us the value of the multiplicative constant.) Second, it is the answer we should have fully expected: since the algorithm simply reduces the size of the remaining array by about half on each iteration, the number of such iterations needed to reduce the initial size n to the final size 1 has to be about $\log_2 n$. Third, to reiterate the point made in Section 2.1,

the logarithmic function grows so slowly that its values remain small even for very large values of n . In particular, according to formula (4.4), it will take no more than $\lfloor \log_2 10^3 \rfloor + 1 = 10$ three-way comparisons to find an element of a given value (or establish that there is no such element) in any sorted array of 1000 elements, and it will take no more than $\lfloor \log_2 10^6 \rfloor + 1 = 20$ comparisons to do this for any sorted array of size one million!

What can we say about the average-case efficiency of binary search? A sophisticated analysis shows that the average number of key comparisons made by binary search is only slightly smaller than that in the worst case:

$$C_{\text{avg}}(n) \approx \log_2 n.$$

(More accurate formulas for the average number of comparisons in a successful and an unsuccessful search are $C_{\text{avg}}^{\text{yes}}(n) \approx \log_2 n - 1$ and $C_{\text{avg}}^{\text{no}}(n) \approx \log_2(n + 1)$, respectively.)

Though binary search is an optimal searching algorithm if we restrict our operations only to comparisons between keys (see Section 10.2), there are searching algorithms (see interpolation search in Section 5.6 and hashing in Section 7.3) with a better average-case efficiency, and one of them (hashing) does not even require the array to be sorted! These algorithms do require some special calculations in addition to key comparisons, however. Finally, the idea behind binary search has several applications beyond searching (see, e.g., [Ben00]). In addition, it can be applied to solving nonlinear equations in one unknown; we discuss this continuous analogue of binary search, called the method of bisection, in Section 12.4.

Before we leave this section, one other remark about binary search needs to be made. Binary search is sometimes presented as the quintessential example of a divide-and-conquer algorithm. This interpretation is flawed because, in fact, binary search is a very atypical case of divide-and-conquer. Indeed, according to the definition given at the beginning of this chapter, the divide-and-conquer technique divides a problem into *several* subproblems, each of which needs to be solved. That is not the case for binary search, where only one of the two subproblems needs to be solved. Therefore, if binary search is to be considered as a divide-and-conquer algorithm, it should be looked on as a degenerate case of this technique. As a matter of fact, binary search fits much better into the class of decrease-by-half algorithms, which we discuss in Section 5.5. Why then is this discussion of binary search in this chapter? Partly because of tradition and partly because a bad example can sometimes make a point that a good example cannot.

Exercises 4.3

1. a. What is the largest number of key comparisons made by binary search in searching for a key in the following array?

10	12	5	9	4	11	8	3	13	2	7	1	6
----	----	---	---	---	----	---	---	----	---	---	---	---

- b. List all the keys of this array that will require the largest number of key comparisons when the search is made by binary search.
2. Solve the recurrence $C_{\text{worst}}(n) = C_{\text{worst}}(\lfloor n/2 \rfloor) + 1$ for $n > 1$, $C_{\text{worst}}(1) = 1$, for $n = 2^k$ by backward substitutions.
3. a. Prove the equality
- $$\lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil \quad \text{for } n \geq 1.$$
- b. Prove that $C_{\text{worst}}(n) = \lfloor \log_2 n \rfloor + 1$ satisfies equation (4.2) for every positive odd integer n .
4. Estimate how many times faster an average successful search will be in a sorted array of 100,000 elements if it is done by binary search versus sequential search.
5. Sequential search can be used with about the same efficiency whether a list is implemented as an array or as a linked list. Is it also true for binary search? (Of course, we assume that a list is sorted for binary search.)
6. How can one use binary search for range searching, i.e., for finding all the elements in a sorted array whose values fall between two given values L and U (inclusively), $L \leq U$? What is the worst-case efficiency of this algorithm?
7. Write a C-code for a recursive version of binary search.
8. Design a version of binary search that uses only two-way comparisons such as \leq and $=$. Implement your algorithm in the language of your choice and carefully debug it (such programs are notorious for being prone to bugs).
9. a. Find the average number of key comparisons made by a successful binary search in the following array.

3	14	27	31	39	42	55	70	74	81	85	93	98
---	----	----	----	----	----	----	----	----	----	----	----	----



- b. In the above array find the average number of key comparisons made by an unsuccessful binary search.
10. *Picture guessing* A version of the popular problem-solving task involves presenting people with an array of 42 pictures—seven rows of six pictures each—and asking them to identify the target picture by asking questions that can be answered yes or no. Further, people are then required to identify the picture with as few questions as possible. Suggest the most efficient algorithm for this problem and indicate the largest number of questions that may be necessary.

4.4 Binary Tree Traversals and Related Properties

In this section, we see how the divide-and-conquer technique can be applied to binary trees. A *binary tree* T is defined as a finite set of nodes that is either empty or

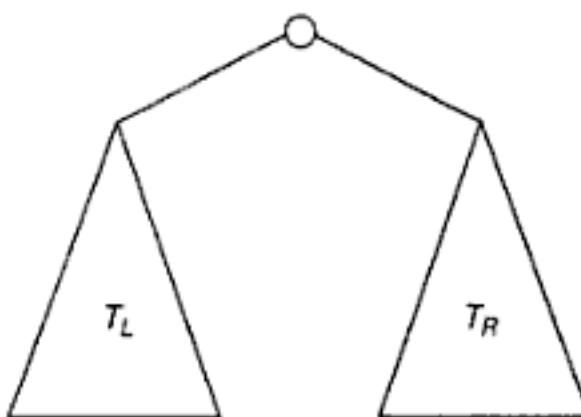


FIGURE 4.4 Standard representation of a binary tree

consists of a root and two disjoint binary trees T_L and T_R called, respectively, the left and right subtree of the root. We usually think of a binary tree as a special case of an ordered tree (Figure 4.4). (This standard interpretation was an alternative definition of a binary tree in Section 1.4.)

Since the definition itself divides a binary tree into two smaller structures of the same type, the left subtree and the right subtree, many problems about binary trees can be solved by applying the divide-conquer technique. As an example, let us consider a recursive algorithm for computing the height of a binary tree. Recall that the height is defined as the length of the longest path from the root to a leaf. Hence, it can be computed as the maximum of the heights of the root's left and right subtrees plus 1. (We add 1 to account for the extra level of the root.) Also note that it is convenient to define the height of the empty tree as -1 . Thus, we have the following recursive algorithm.

ALGORITHM *Height(T)*

```
//Computes recursively the height of a binary tree
//Input: A binary tree  $T$ 
//Output: The height of  $T$ 
if  $T = \emptyset$  return  $-1$ 
else return  $\max\{\text{Height}(\mathcal{T}_L), \text{Height}(\mathcal{T}_R)\} + 1$ 
```

We measure the problem's instance size by the number of nodes $n(T)$ in a given binary tree T . Obviously, the number of comparisons made to compute the maximum of two numbers and the number of additions $A(n(T))$ made by the algorithm are the same. We have the following recurrence relation for $A(n(T))$:

$$\begin{aligned}A(n(T)) &= A(n(\mathcal{T}_L)) + A(n(\mathcal{T}_R)) + 1 \quad \text{for } n(T) > 0, \\A(0) &= 0.\end{aligned}$$

Before we solve this recurrence (can you tell what its solution is?), let us note that addition is not the most frequently executed operation of this algorithm. What is? Checking—and this is very typical for binary tree algorithms—that the tree is not empty. For example, for the empty tree, the comparison $T = \emptyset$ is executed once but there are no additions, and for a single-node tree, the comparison and addition numbers are three and one, respectively.

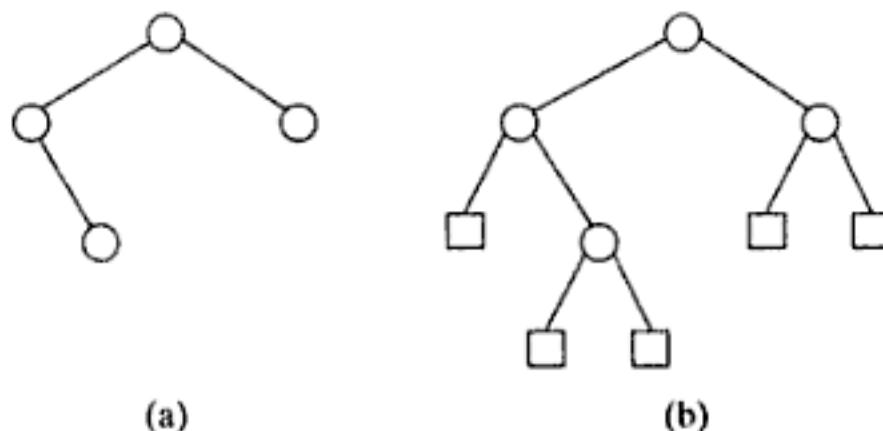


FIGURE 4.5 (a) Binary tree. (b) Its extension. Internal nodes are shown as circles; external nodes are shown as squares.

It helps in the analysis of tree algorithms to draw the tree's extension by replacing the empty subtrees by special nodes. The extra nodes (shown by little squares in Figure 4.5) are called *external*; the original nodes (shown by little circles) are called *internal*. By definition, the extension of the empty binary tree is a single external node.

It is easy to see that the height algorithm makes exactly one addition for every internal node of the extended tree, and it makes one comparison to check whether the tree is empty for every internal and external node. Thus, to ascertain the algorithm's efficiency, we need to know how many external nodes an extended binary tree with n internal nodes can have. Checking Figure 4.5 and a few similar examples, it is easy to hypothesize that the number of external nodes x is always one more than the number of internal nodes n :

$$x = n + 1. \quad (4.5)$$

To prove this formula, consider the total number of nodes, both internal and external. Since every node, except the root, is one of the two children of an internal node, we have the equation

$$2n + 1 = x + n,$$

which immediately implies equation (4.5).

Note that equation (4.5) also applies to any nonempty *full binary tree*, in which, by definition, every node has either zero or two children: for a full binary tree, n and x denote the numbers of parental nodes and leaves, respectively.

Returning to algorithm *Height*, the number of comparisons to check whether the tree is empty is

$$C(n) = n + x = 2n + 1,$$

while the number of additions is

$$A(n) = n.$$

The most important divide-and-conquer algorithms for binary trees are the three classic traversals: preorder, inorder, and postorder. All three traversals visit nodes of a binary tree recursively, i.e., by visiting the tree's root and its left and

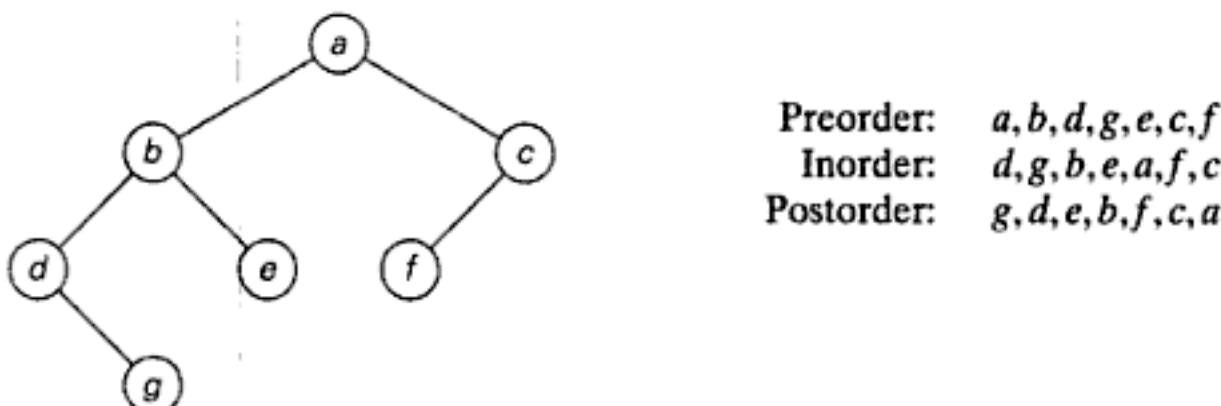


FIGURE 4.6 Binary tree and its traversals

right subtrees. They differ just by the timing of the root's visit:

In the **preorder traversal**, the root is visited before the left and right subtrees are visited (in that order).

In the **inorder traversal**, the root is visited after visiting its left subtree but before visiting the right subtree.

In the **postorder traversal**, the root is visited after visiting the left and right subtrees (in that order).

These traversals are illustrated in Figure 4.6. Their pseudocodes are quite straightforward, repeating the descriptions given above. (These traversals are also a standard feature of data structures textbooks.) As to their efficiency analysis, it is identical to the above analysis of the *Height* algorithm because a recursive call is made for each node of an extended binary tree.

Finally, we should note that, obviously, not all questions about binary trees require traversals of both left and right subtrees. For example, the find and insert operations for a binary search tree require processing only one of the two subtrees. Hence, they should be considered not as applications of divide-and-conquer but rather as examples of the variable-size decrease technique discussed in Section 5.6.

Exercises 4.4

1. Design a divide-and-conquer algorithm for computing the number of levels in a binary tree. (In particular, the algorithm must return 0 and 1 for the empty and single-node trees, respectively.) What is the efficiency class of your algorithm?
2. The following algorithm seeks to compute the number of leaves in a binary tree.

ALGORITHM *LeafCounter*(T)

//Computes recursively the number of leaves in a binary tree

//Input: A binary tree T

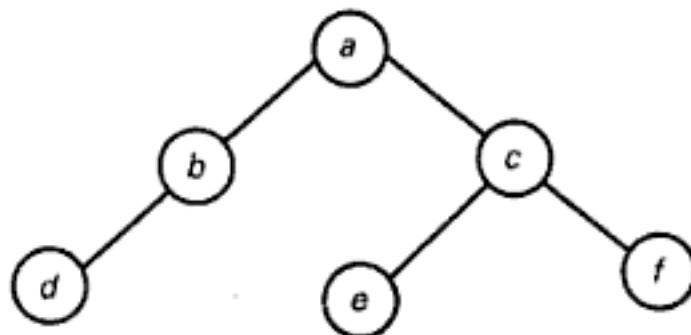
//Output: The number of leaves in T

if $T = \emptyset$ **return** 0

else return *LeafCounter*(T_L) + *LeafCounter*(T_R)

Is this algorithm correct? If it is, prove it; if it is not, make an appropriate correction.

3. Prove equality (4.5) by mathematical induction.
4. Discuss the following binary tree traversals:
 - a. preorder.
 - b. inorder.
 - c. postorder.



5. Write a pseudocode for computing the height of a binary tree. Assume the algorithm to be recursive and find the number of recursive calls made.
6. Write a report on the following:
 - a. binary tree.
 - b. full binary tree.
 - c. internal and external node.
 - d. traversals and their types.
7. a. Draw a binary tree with ten nodes labeled $0, 1, 2, \dots, 9$ in such a way that the inorder and postorder traversals of the tree yield the following lists: $9, 3, 1, 0, 4, 2, 7, 6, 8, 5$ (inorder) and $9, 1, 4, 0, 3, 6, 7, 5, 8, 2$ (postorder).
 b. Give an example of two permutations of the same n labels $0, 1, 2, \dots, n - 1$ that cannot be inorder and postorder traversal lists of the same binary tree.
 c. Design an algorithm that constructs a binary tree for which two given lists of n labels $0, 1, 2, \dots, n - 1$ are generated by the inorder and postorder traversals of the tree. Your algorithm should also identify inputs for which the problem has no solution.
8. The ***internal path length I*** of an extended binary tree is defined as the sum of the lengths of the paths—taken over all internal nodes—from the root to each internal node. Similarly, the ***external path length E*** of an extended binary tree is defined as the sum of the lengths of the paths—taken over all external nodes—from the root to each external node. Prove that $E = I + 2n$ where n is the number of internal nodes in the tree.
9. Write a program for computing the internal path length of an extended binary tree. Use it to investigate empirically the average number of key comparisons for searching in a randomly generated binary search tree.
10. ***Chocolate bar puzzle*** Given an n -by- m chocolate bar, you need to break it into nm 1-by-1 pieces. You can break a bar only in a straight line, and only one



bar can be broken at a time. Design an algorithm that solves the problem with the minimum number of bar breaks. What is this minimum number? Justify your answer by using properties of a binary tree.

4.5 Multiplication of Large Integers and Strassen's Matrix Multiplication

In this section, we examine two surprising algorithms for seemingly straightforward tasks: multiplying two numbers and multiplying two square matrices. Both seek to decrease the total number of multiplications performed at the expense of a slight increase in the number of additions. Both do this by exploiting the divide-and-conquer idea.

Multiplication of Large Integers

Some applications, notably modern cryptology, require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer, they require special treatment. This practical need supports investigations of algorithms for efficient manipulation of large integers. In this section, we outline an interesting algorithm for multiplying such numbers. Obviously, if we use the classic pen-and-pencil algorithm for multiplying two n -digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications. (If one of the numbers has fewer digits than the other, we can pad a shorter number with leading zeros to equal their lengths.) Though it might appear that it would be impossible to design an algorithm with fewer than n^2 digit multiplications, it turns out not to be the case. The miracle of divide-and-conquer comes to the rescue to accomplish this feat.

To demonstrate the basic idea of the algorithm, let us start with a case of two-digit integers, say, 23 and 14. These numbers can be represented as follows:

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0 \text{ and } 14 = 1 \cdot 10^1 + 4 \cdot 10^0.$$

Now let us multiply them:

$$\begin{aligned} 23 * 14 &= (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0) \\ &= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0. \end{aligned}$$

The last formula yields the correct answer of 322, of course, but it uses the same four digit multiplications as the pen-and-pencil algorithm. Fortunately, we can compute the middle term with just one digit multiplication by taking advantage of the products $2 * 1$ and $3 * 4$ that need to be computed anyway:

$$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - 2 * 1 - 3 * 4.$$

Of course, there is nothing special about the numbers we just multiplied. For any pair of two-digit integers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0,$$

where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a 's digits and the sum of the b 's digits minus the sum of c_2 and c_0 .

Now we apply this trick to multiplying two n -digit integers a and b where n is a positive even number. Let us divide both numbers in the middle—after all, we promised to take advantage of the divide-and-conquer technique. We denote the first half of the a 's digits by a_1 and the second half by a_0 ; for b , the notations are b_1 and b_0 , respectively. In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$, and $b = b_1b_0$ implies that $b = b_1 10^{n/2} + b_0$. Therefore, taking advantage of the same trick we used for two-digit numbers, we get

$$\begin{aligned} c = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\ &= (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 * b_1) 10^{n/2} + (a_0 * b_0) \\ &= c_2 10^n + c_1 10^{n/2} + c_0, \end{aligned}$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,

$c_0 = a_0 * b_0$ is the product of their second halves,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a 's halves and the sum of the b 's halves minus the sum of c_2 and c_0 .

If $n/2$ is even, we can apply the same method for computing the products c_2 , c_0 , and c_1 . Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n -digit integers. In its pure form, the recursion is stopped when n becomes one. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

How many digit multiplications does this algorithm make? Since multiplication of n -digit numbers requires three multiplications of $n/2$ -digit numbers, the recurrence for the number of multiplications $M(n)$ will be

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Solving it by backward substitutions for $n = 2^k$ yields

$$\begin{aligned} M(2^k) &= 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2 M(2^{k-2}) \\ &= \dots = 3^i M(2^{k-i}) = \dots = 3^k M(2^{k-k}) = 3^k. \end{aligned}$$

Since $k = \log_2 n$,

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}.$$

(On the last step, we took advantage of the following property of logarithms: $a^{\log_b c} = c^{\log_b a}$.)

You should keep in mind that for moderately large integers, this algorithm will probably run longer than the classic one. Brassard and Bratley ([Bra96], pp. 70–71) report that in their experiments the divide-and-conquer algorithm started to outperform the pen-and-pencil method on integers over 600 digits long. If you program in an object-oriented language such as Java, C++, or Smalltalk, you should also be aware that these languages have special classes for dealing with large integers.

Strassen's Matrix Multiplication

Now that we have seen that the divide-and-conquer approach can reduce the number of one-digit multiplications in multiplying two integers, we should not be surprised that a similar feat can be accomplished for multiplying matrices. Such an algorithm was published by V. Strassen in 1969 [Str69]. The principal insight of the algorithm lies in the discovery that we can find the product C of two 2-by-2 matrices A and B with just seven multiplications as opposed to the eight required by the brute-force algorithm (see Example 3, Section 2.3). This is accomplished by using the following formulas:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} \\ = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

where

$$\begin{aligned} m_1 &= (a_{00} + a_{11}) * (b_{00} + b_{11}) \\ m_2 &= (a_{10} + a_{11}) * b_{00} \\ m_3 &= a_{00} * (b_{01} - b_{11}) \\ m_4 &= a_{11} * (b_{10} - b_{00}) \\ m_5 &= (a_{00} + a_{01}) * b_{11} \\ m_6 &= (a_{10} - a_{00}) * (b_{00} + b_{01}) \\ m_7 &= (a_{01} - a_{11}) * (b_{10} + b_{11}) \end{aligned}$$

Thus, to multiply two 2-by-2 matrices, Strassen's algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions. These numbers should not lead us to multiplying 2-by-2 matrices by Strassen's algorithm. Its importance stems from its *asymptotic* superiority as matrix order n goes to infinity.

Let A and B be two n -by- n matrices where n is a power of two. (If n is not a power of two, matrices can be padded with rows and columns of zeros.) We can divide A , B , and their product C into four $n/2$ -by- $n/2$ submatrices each as follows:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

It is not difficult to verify that one can treat these submatrices as numbers to get the correct product. For example, C_{00} can be computed either as $A_{00} * B_{00} + A_{01} * B_{10}$ or as $M_1 + M_4 - M_5 + M_7$ where M_1, M_4, M_5 , and M_7 are found by Strassen's formulas, with the numbers replaced by the corresponding submatrices. If the seven products of $n/2$ -by- $n/2$ matrices are computed recursively by the same method, we have Strassen's algorithm for matrix multiplication.

Let us evaluate the asymptotic efficiency of this algorithm. If $M(n)$ is the number of multiplications made by Strassen's algorithm in multiplying two n -by- n matrices (where n is a power of 2), we get the following recurrence relation for it:

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Since $n = 2^k$,

$$\begin{aligned} M(2^k) &= 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \dots \\ &= 7^iM(2^{k-i}) \dots = 7^kM(2^{k-k}) = 7^k. \end{aligned}$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

which is smaller than n^3 required by the brute-force algorithm.

Since this saving in the number of multiplications was achieved at the expense of making extra additions, we must check the number of additions $A(n)$ made by Strassen's algorithm. To multiply two matrices of order $n > 1$, the algorithm needs to multiply seven matrices of order $n/2$ and make 18 additions of matrices of size $n/2$; when $n = 1$, no additions are made since two numbers are simply multiplied. These observations yield the following recurrence relation:

$$A(n) = 7A(n/2) + 18(n/2)^2 \quad \text{for } n > 1, \quad A(1) = 0.$$

Though one can obtain a closed-form solution to this recurrence (see Problem 8), here we simply establish the solution's order of growth. According to the Master Theorem stated in the beginning of the chapter, $A(n) \in \Theta(n^{\log_2 7})$. In other words, the number of additions has the same order of growth as the number of multiplications. This puts Strassen's algorithm in $\Theta(n^{\log_2 7})$, which is a better efficiency class than $\Theta(n^3)$ of the brute-force method.

Since the time of Strassen's discovery, several other algorithms for multiplying two n -by- n matrices of real numbers in $O(n^\alpha)$ time with progressively smaller constants α have been invented. The fastest algorithm so far is that of Coopersmith and Winograd [Coo87], with its efficiency in $O(n^{2.376})$. The decreasing values of

the exponents have been obtained at the expense of increasing complexity of these algorithms. Because of large multiplicative constants, none of them is of practical value. However, they are interesting from a theoretical point of view. Although these algorithms get closer and closer to the best theoretical lower bound known for matrix multiplication, which is n^2 multiplications, the gap between this bound and the best available algorithm remains unresolved. It is also worth mentioning that matrix multiplication is known to be computationally equivalent to some other important problems such as solving systems of linear equations.

Exercises 4.5

1. What are the smallest and largest numbers of digits the product of two decimal n -digit integers can have?
2. Compute $2101 * 1130$ by applying the divide-and-conquer algorithm outlined in the text.
3.
 - a. Prove the equality $a^{\log_b c} = c^{\log_b a}$, which was used twice in Section 4.5.
 - b. Why is $n^{\log_2 3}$ better than $3^{\log_2 n}$ as a closed-form formula for $M(n)$?
4.
 - a. Why did we not include multiplications by 10^n in the multiplication count $M(n)$ of the large-integer multiplication algorithm?
 - b. In addition to assuming that n is a power of 2, we made, for the sake of simplicity, another, more subtle, assumption in setting up a recurrence relation for $M(n)$, which is not always true (it does not change the final answer, however). What is this assumption?
5. If only the number of additions and the number of multiplications are taken into account, which one of the following algorithms would be perfect and why?
 - a. Strassen's algorithm.
 - b. brute-force algorithm.
6. Write a pseudocode for obtaining the product of two matrices using the Strassen's matrix multiplication.
7. Apply Strassen's algorithm to compute:

$$\begin{bmatrix} 1 & 2 & 1 & 1 \\ 0 & 3 & 2 & 4 \\ 0 & 1 & 1 & 1 \\ 5 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} 2 & 1 & 0 & 2 \\ 1 & 2 & 1 & 1 \\ 0 & 3 & 2 & 1 \\ 4 & 0 & 0 & 4 \end{bmatrix}$$

Exiting the recursion when $n = 2$, compute the product of 2-by-2 matrices by brute-force algorithm.

8. Solve the recurrence for the number of additions required by Strassen's algorithm. (Assume that n is a power of 2.)
9. V. Pan [Pan78] has discovered a divide-and-conquer matrix multiplication algorithm that is based on multiplying two 70-by-70 matrices using 143,640

- multiplications. Find the asymptotic efficiency of Pan's algorithm (you may ignore additions) and compare it with that of Strassen's algorithm.
10. Practical implementations of Strassen's algorithm usually switch to the brute-force method after matrix sizes become smaller than some "crossover point." Run an experiment to determine such crossover point on your computer system.
-

4.6 Closest-Pair and Convex-Hull Problems by Divide-and-Conquer

In Section 3.3, we discussed the brute-force approach to solving two classic problems of computational geometry: the closest-pair problem and the convex-hull problem. We saw that the two-dimensional versions of these problems can be solved by brute-force algorithms in $\Theta(n^2)$ and $O(n^3)$ time, respectively. In this section, we discuss more sophisticated and asymptotically more efficient algorithms for these problems, which are based on the divide-and-conquer technique.

Closest-Pair Problem

Let $P_1 = (x_1, y_1), \dots, P_n = (x_n, y_n)$ be a set S of n points in the plane, where n , for simplicity, is a power of two. With no loss of generality, we can assume that the points are ordered in ascending order of their x coordinates. (If they were not, we can sort them in $O(n \log n)$ time, e.g., by mergesort.) We can divide the points given into two subsets S_1 and S_2 of $n/2$ points each by drawing a vertical line $x = c$ so that $n/2$ points lie to the left of or on the line itself, and $n/2$ points lie to the right of or on the line. (One way of finding an appropriate value for constant c for doing this is to use the median μ of the x coordinates.)

Following the divide-and-conquer approach, we can find recursively the closest pairs for the left subset S_1 and the right subset S_2 . Let d_1 and d_2 be the smallest distances between pairs of points in S_1 and S_2 , respectively, and let $d = \min\{d_1, d_2\}$. Unfortunately, d is not necessarily the smallest distance between all pairs of points in S_1 and S_2 because a closer pair of points can lie on the opposite sides of the separating line. So, as a step of combining the solutions to the smaller subproblems, we need to examine such points. Obviously, we can limit our attention to the points in the symmetric vertical strip of width $2d$ since the distance between any other pair of points is greater than d (Figure 4.7a).

Let C_1 and C_2 be the subsets of points in the left and right parts of the strip, respectively. Now, for every point $P(x, y)$ in C_1 , we need to inspect points in C_2 that may be closer to P than d . Obviously, such points must have their y coordinates in the interval $[y - d, y + d]$. The critical insight here is an observation that there can be no more than six such points because any pair of points in C_2 is at least d apart from each other. (Recall that $d \leq d_2$ where d_2 is the smallest distance between

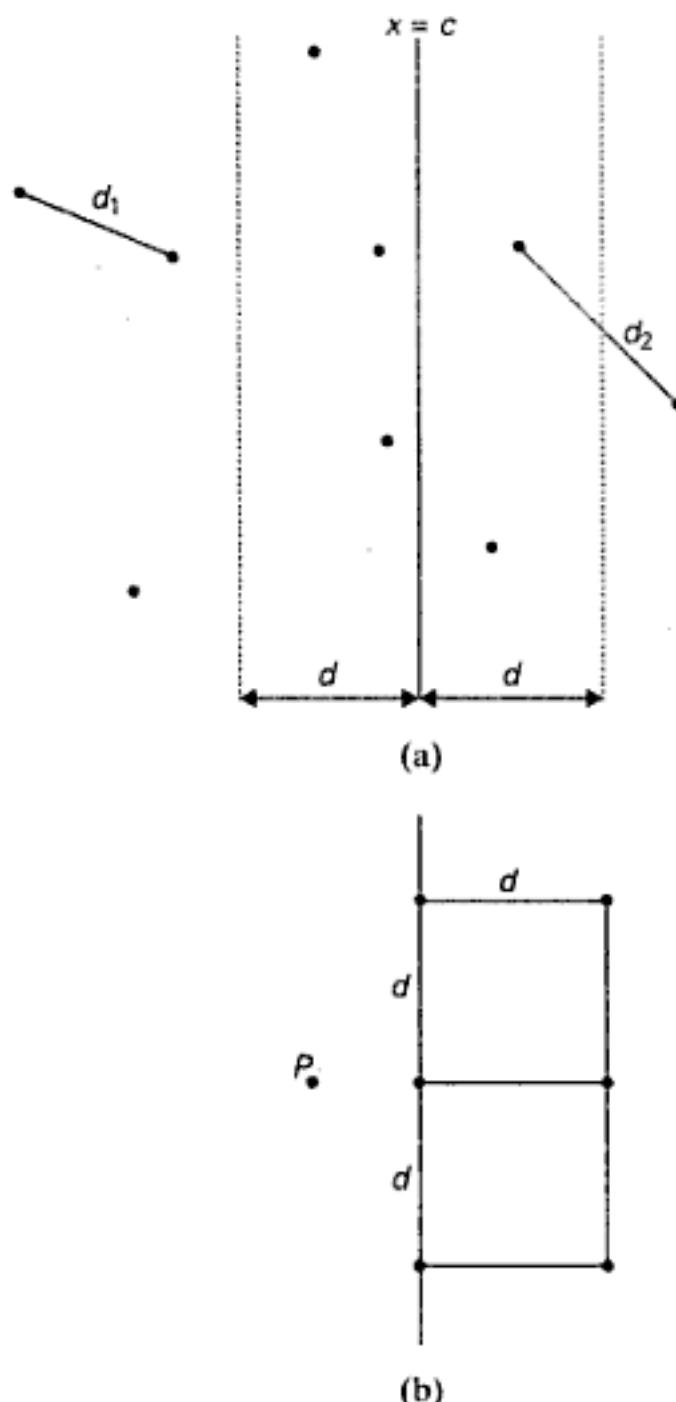


FIGURE 4.7 (a) Idea of the divide-and-conquer algorithm for the closest-pair problem.
(b) The six points that may need to be examined for point P .

pairs of points to the right of the dividing line.) The worst case is illustrated in Figure 4.7b.

Another important observation is that we can maintain lists of points in C_1 and C_2 sorted in ascending order of their y coordinates. (You can think of these lists as projections of the points on the dividing line.) Moreover, this ordering can be maintained not by resorting points on each iteration but rather by merging two previously sorted lists (see algorithm *Merge* in Section 4.1). We can process the C_1 points sequentially while a pointer into the C_2 list scans an interval of width $2d$ to fetch up to six candidates for computing their distances to a current point P of the C_1 list. The time $M(n)$ for this “merging” of solutions to the smaller subproblems is in $O(n)$.

We have the following recurrence for $T(n)$, the running time of this algorithm on n presorted points:

$$T(n) = 2T(n/2) + M(n).$$

Applying the O version of the Master Theorem (with $a = 2$, $b = 2$, and $d = 1$), we get $T(n) \in O(n \log n)$. The possible necessity to presort input points does not change the overall efficiency class if sorting is done by a $O(n \log n)$ algorithm. In fact, this is the best efficiency class we can achieve because it has been proved that any algorithm for this problem must be in $\Omega(n \log n)$ (see [Pre85], p. 188).

Convex-Hull Problem

Let us revisit the convex-hull problem introduced in Section 3.3: find the smallest convex polygon that contains n given points in the plane. We consider here a divide-and-conquer algorithm called *quickhull* because of its resemblance to quicksort.

Let $P_1 = (x_1, y_1), \dots, P_n = (x_n, y_n)$ be a set S of $n > 1$ points in the plane. We assume that the points are sorted in increasing order of their x coordinates, with ties resolved by increasing order of the y coordinates of the points involved. It is not difficult to prove the geometrically obvious fact that the leftmost point P_1 and the rightmost point P_n are two distinct extreme points of the set's convex hull (Figure 4.8). Let $\overrightarrow{P_1P_n}$ be the straight line through points P_1 and P_n directed from P_1 to P_n . This line separates the points of S into two sets: S_1 is the set of points to the left of or on this line and S_2 is the set of points to the right of or on this line. (We say that point p_3 is to the left of the line $\overrightarrow{p_1p_2}$ directed from point p_1 to point p_2 if $p_1p_2p_3$ forms a counterclockwise cycle. Later, we cite an analytical way to check this condition based on checking the sign of a determinant formed by the coordinates of the three points.) The points of S on the line $\overrightarrow{P_1P_n}$, other than P_1 and P_n , cannot be extreme points of the convex hull and hence are excluded from further consideration.

The boundary of the convex hull of S is made up of two polygonal chains: an “upper” boundary and a “lower” boundary. The “upper” boundary, called the *upper hull*, is a sequence of line segments with vertices at P_1 , some of the points

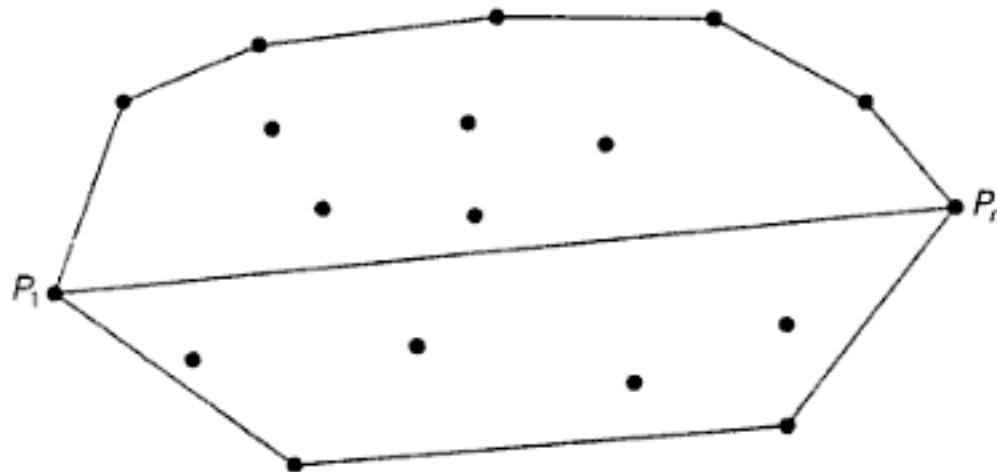


FIGURE 4.8 Upper and lower hulls of a set of points

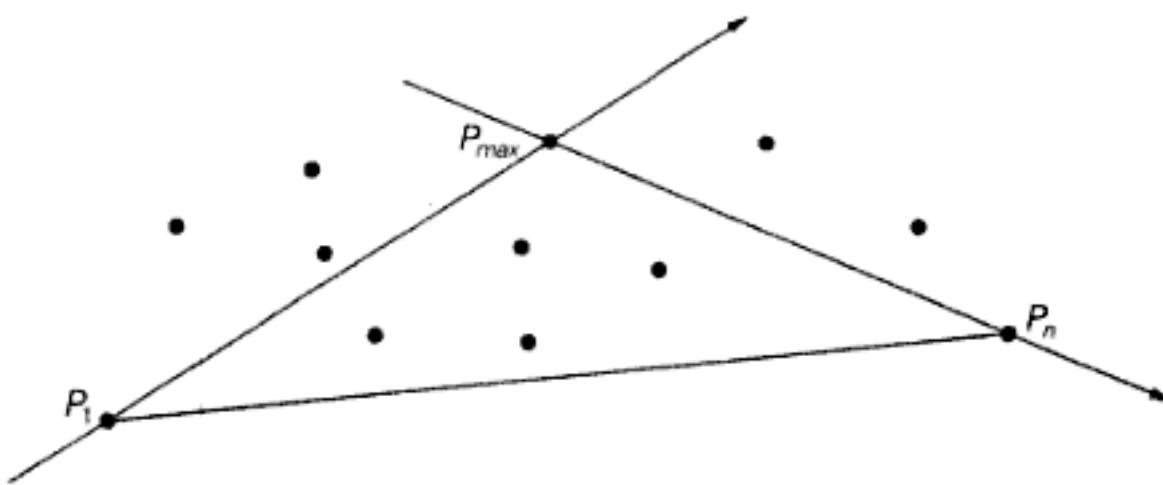


FIGURE 4.9 The idea of quickhull

in S_1 (if S_1 is not empty), and P_n . The “lower” boundary, called the **lower hull**, is a sequence of line segments with vertices at P_1 , some of the points in S_2 (if S_2 is not empty) and P_n .

The fact that the convex hull of the entire set S is composed of the upper and lower hulls, which can be constructed independently and in a similar fashion, is a very useful observation that is exploited by several algorithms for this problem.

For concreteness, let us discuss how quickhull proceeds to construct the upper hull; the lower hull can be constructed in the same manner. If S_1 is empty, the upper hull is simply the line segment with the endpoints at P_1 and P_n . If S_1 is not empty, the algorithm identifies vertex P_{\max} in S_1 , which is the farthest from the line $\overrightarrow{P_1P_n}$ (Figure 4.9). If there is a tie, the point that maximizes the angle $\angle P_{\max}P_1P_n$ can be selected. (Note that point P_{\max} maximizes the area of the triangle with two vertices at P_1 and P_n and the third at some other point of S_1 .) Then the algorithm identifies all the points of set S_1 that are to the left of the line $\overrightarrow{P_1P_{\max}}$; these are the points that, along with P_1 and P_{\max} , will make up the set $S_{1,1}$. The points of S_1 to the left of the line $\overrightarrow{P_{\max}P_n}$ will make up, along with P_{\max} and P_n , the set $S_{1,2}$. It is not difficult to prove that

- P_{\max} is a vertex of the upper hull;
- the points inside $\Delta P_1P_{\max}P_n$ cannot be vertices of the upper hull (and hence can be eliminated from further consideration); and
- there are no points to the left of both lines $\overrightarrow{P_1P_{\max}}$ and $\overrightarrow{P_{\max}P_n}$.

Therefore, the algorithm can continue constructing the upper hulls of $P_1 \cup S_{1,1} \cup P_{\max}$ and $P_{\max} \cup S_{1,2} \cup P_n$ recursively and then simply concatenate them to get the upper hull of the entire set $P_1 \cup S_1 \cup P_n$.

Now we have to figure out how the algorithm’s geometric operations can be implemented. Fortunately, we can take advantage of the following very useful fact from analytical geometry: if $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, and $p_3 = (x_3, y_3)$ are three arbitrary points in the Cartesian plane, then the area of the triangle $\Delta p_1p_2p_3$ is

equal to one half of the magnitude of the determinant

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3,$$

while the sign of this expression is positive if and only if the point $p_3 = (x_3, y_3)$ is to the left of the line $\overrightarrow{p_1 p_2}$. Using this formula, we can check in constant time whether a point lies to the left of the line determined by two other points as well as find the distance from the point to the line.

Quickhull has the same $\Theta(n^2)$ worst-case efficiency as quicksort (Problem 8 in the exercises). In the average case, however, we should expect a much better performance. First, the algorithm should benefit from the quicksort-like savings from the on-average balanced split of the problem into two smaller subproblems. Second, a significant fraction of the points—namely those inside $\Delta P_1 P_{\max} P_n$ (see Figure 4.9)—are eliminated from further processing. Under a natural assumption that points given are chosen randomly from a uniform distribution over some convex region (e.g., a circle or a rectangle), the average-case efficiency of quickhull turns out to be linear [Ove80].

Exercises 4.6

1. a. For the one-dimensional version of the closest-pair problem, i.e., for the problem of finding two closest numbers among a given set of n real numbers, design an algorithm that is directly based on the divide-and-conquer technique and determine its efficiency class.
b. Is it a good algorithm for this problem?
2. Consider the version of the divide-and-conquer two-dimensional closest-pair algorithm in which we simply sort each of the two sets C_1 and C_2 in ascending order of their y coordinates on each recursive call. Assuming that sorting is done by mergesort, set up a recurrence relation for the running time in the worst case and solve it for $n = 2^k$.
3. Implement in C, the divide-and-conquer closest-pair algorithm.
4. Find on the Web a visualization of an algorithm for the closest-pair problem. What algorithm does this visualization represent?
5. The **Voronoi polygon** for a point P of a set S of points in the plane is defined to be the perimeter of the set of all points in the plane closer to P than to any other point in S . The union of all the Voronoi polygons of the points in S is called the **Voronoi diagram** of S .
 - a. What is the Voronoi diagram for a set of three points?
 - b. Find on the Web a visualization of an algorithm for generating the Voronoi diagram and study a few examples of such diagrams. Based on your observations, can you tell how the solution to the previous question is generalized to the general case?

6. Explain the upper and lower hulls in the convex-hull problem, with an example.
7. What is the best-case efficiency of quickhull?
8. Give a specific example of inputs that make the quickhull algorithm run in quadratic time.
9. Write a pseudocode for the quickhull algorithm.
10. *Shortest path around* There is a fenced area in the two-dimensional Euclidean plane in the shape of a convex polygon with vertices at points $P_1(x_1, y_1), P_2(x_2, y_2), \dots, P_n(x_n, y_n)$ (not necessarily in this order). There are two more points, $A(x_A, y_A)$ and $B(x_B, y_B)$, such that $x_A < \min\{x_1, x_2, \dots, x_n\}$ and $x_B > \max\{x_1, x_2, \dots, x_n\}$. Design a reasonably efficient algorithm for computing the length of the shortest path between A and B . [ORo98], p. 68



SUMMARY

- *Divide-and-conquer* is a general algorithm design technique that solves a problem's instance by dividing it into several smaller instances (ideally, of equal size), solving each of them recursively, and then combining their solutions to get a solution to the original instance of the problem. Many efficient algorithms are based on this technique, although it can be both inapplicable and inferior to simpler algorithmic solutions.
- Running time $T(n)$ of many divide-and-conquer algorithms satisfies the recurrence $T(n) = aT(n/b) + f(n)$. The *Master Theorem* establishes the order of growth of its solutions.
- *Mergesort* is a divide-and-conquer sorting algorithm. It works by dividing an input array into two halves, sorting them recursively, and then *merging* the two sorted halves to get the original array sorted. The algorithm's time efficiency is in $\Theta(n \log n)$ in all cases, with the number of key comparisons being very close to the theoretical minimum. Its principal drawback is a significant extra storage requirement.
- *Quicksort* is a divide-and-conquer sorting algorithm that works by partitioning its input's elements according to their value relative to some preselected element. Quicksort is noted for its superior efficiency among $n \log n$ algorithms for sorting randomly ordered arrays but also for the quadratic worst-case efficiency.
- *Binary search* is a $O(\log n)$ algorithm for searching in sorted arrays. It is an atypical example of an application of the divide-and-conquer technique because it needs to solve just one problem of half the size on each of its iterations.
- The classic traversals of a binary tree—*preorder*, *inorder*, and *postorder*—and similar algorithms that require recursive processing of both left and right subtrees can be considered examples of the divide-and-conquer technique.

Their analysis is helped by replacing all the empty subtrees of a given tree with special *external nodes*.

- There is a divide-and-conquer algorithm for multiplying two n -digit integers that requires about $n^{1.585}$ one-digit multiplications.
- *Strassen's algorithm* needs only seven multiplications to multiply two 2-by-2 matrices but requires more additions than the definition-based algorithm. By exploiting the divide-and-conquer technique, this algorithm can multiply two n -by- n matrices with about $n^{2.807}$ multiplications.
- The divide-and-conquer technique can be successfully applied to two important problems of computational geometry: the closest-pair problem and the convex-hull problem.

5

Decrease-and-Conquer

Plutarch says that Sertorius, in order to teach his soldiers that perseverance and wit are better than brute force, had two horses brought before them, and set two men to pull out their tails. One of the men was a burly Hercules, who tugged and tugged, but all to no purpose; the other was a sharp, weasel-faced tailor, who plucked one hair at a time, amidst roars of laughter, and soon left the tail quite bare.

—E. Cobham Brewer, *Dictionary of Phrase and Fable*, 1898

The **decrease-and-conquer** technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. Once such a relationship is established, it can be exploited either top down (recursively) or bottom up (without a recursion). There are three major variations of decrease-and-conquer:

- decrease by a constant
- decrease by a constant factor
- variable size decrease

In the **decrease-by-a-constant** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (Figure 5.1), although reduction-by-two cases do happen occasionally, for example, in algorithms that have to act differently for instances of odd and even sizes.

Consider, as an example, the exponentiation problem of computing a^n for positive integer exponents. The relationship between a solution to an instance of size n and an instance of size $n - 1$ is obtained by the obvious formula: $a^n = a^{n-1} \cdot a$. So the function $f(n) = a^n$ can be computed either “top down” by using its recursive definition

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases} \quad (5.1)$$

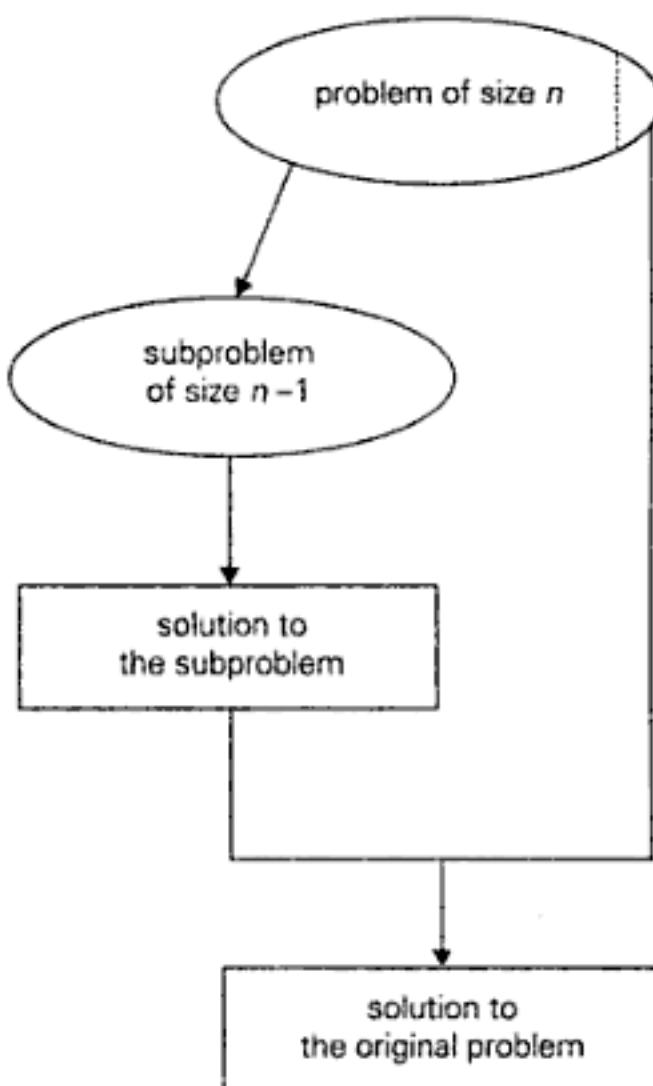


FIGURE 5.1 Decrease (by one)-and-conquer technique

or “bottom up” by multiplying a by itself $n - 1$ times. (Yes, it is the same as the brute-force algorithm, but we have come to it by a different thought process.) More interesting examples of decrease-by-one algorithms appear in Sections 5.1–5.4.

The **decrease-by-a-constant-factor** technique suggests reducing a problem’s instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two. (Can you give an example of such an algorithm?) The decrease-by-half idea is illustrated in Figure 5.2.

For an example, let us revisit the exponentiation problem. If the instance of size n is to compute a^n , the instance of half its size will be to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = (a^{n/2})^2$. But since we consider here instances of the exponentiation problem with integer exponents only, the former does not work for odd n . If n is odd, we have to compute a^{n-1} by using the rule for even-valued exponents and then multiply the result by a . To summarize, we have the following formula:

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd and greater than 1} \\ a & \text{if } n = 1. \end{cases} \quad (5.2)$$

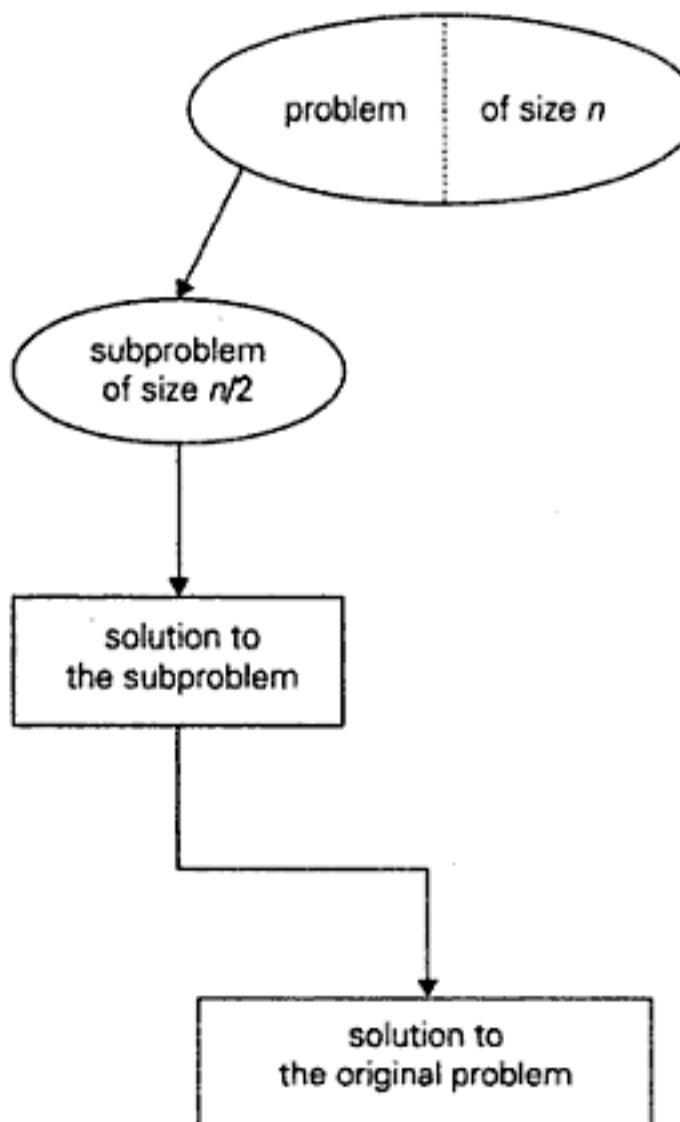


FIGURE 5.2 Decrease (by half)-and-conquer technique

If we compute a^n recursively according to formula (5.2) and measure the algorithm's efficiency by the number of multiplications, we should expect the algorithm to be in $O(\log n)$ because, on each iteration, the size is reduced by at least one half at the expense of no more than two multiplications.

Note a difference between this algorithm and the one based on the divide-and-conquer idea of solving two instances of the exponentiation problem of size $n/2$:

$$a^n = \begin{cases} a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil} & \text{if } n > 1 \\ a & \text{if } n = 1. \end{cases} \quad (5.3)$$

The algorithm based on formula (5.3) is inefficient (why?), whereas the one based on (5.2) is much faster.

A few other examples of decrease-by-a-constant-factor algorithms are given in Section 5.5 and its exercises. Such algorithms are so efficient, however, that there are few examples of this kind.

Finally, in the **variable-size-decrease** variety of decrease-and-conquer, a size reduction pattern varies from one iteration of an algorithm to another. Euclid's

algorithm for computing the greatest common divisor provides a good example of such a situation. Recall that this algorithm is based on the formula

$$\gcd(m, n) = \gcd(n, m \bmod n).$$

Though the arguments on the right-hand side are always smaller than those on the left-hand side (at least starting with the second iteration of the algorithm), they are smaller neither by a constant nor by a constant factor. A few other examples of such algorithms appear in Section 5.6.

5.1 Insertion Sort

In this section, we consider an application of the decrease-by-one technique to sorting an array $A[0..n - 1]$. Following the technique's idea, we assume that the smaller problem of sorting the array $A[0..n - 2]$ has already been solved to give us a sorted array of size $n - 1$: $A[0] \leq \dots \leq A[n - 2]$. How can we take advantage of this solution to the smaller problem to get a solution to the original problem by taking into account the element $A[n - 1]$? Obviously, all we need is to find an appropriate position for $A[n - 1]$ among the sorted elements and insert it there.

There are three reasonable alternatives for doing this. First, we can scan the sorted subarray from left to right until the first element greater than or equal to $A[n - 1]$ is encountered and then insert $A[n - 1]$ right before that element. Second, we can scan the sorted subarray from right to left until the first element smaller than or equal to $A[n - 1]$ is encountered and then insert $A[n - 1]$ right after that element. These two alternatives are essentially equivalent; usually, it is the second one that is implemented in practice because it is better for sorted and almost-sorted arrays (why?). The resulting algorithm is called ***straight insertion sort*** or simply ***insertion sort***. The third alternative is to use binary search to find an appropriate position for $A[n - 1]$ in the sorted portion of the array. The resulting algorithm is called ***binary insertion sort***. We ask you to implement this idea and investigate the efficiency of binary insertion sort in the exercises to this section.

Though insertion sort is clearly based on a recursive idea, it is more efficient to implement this algorithm bottom up, i.e., iteratively. As shown in Figure 5.3, starting with $A[1]$ and ending with $A[n - 1]$, $A[i]$ is inserted in its appropriate place among the first i elements of the array that have been already sorted (but, unlike selection sort, are generally not in their final positions).

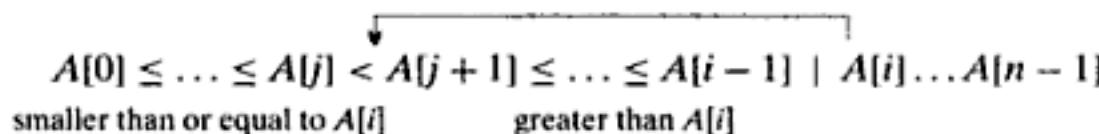


FIGURE 5.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

Here is a pseudocode of this algorithm.

```
ALGORITHM InsertionSort(A[0..n - 1])
  //Sorts a given array by insertion sort
  //Input: An array A[0..n - 1] of n orderable elements
  //Output: Array A[0..n - 1] sorted in nondecreasing order
  for i ← 1 to n - 1 do
    v ← A[i]
    j ← i - 1
    while j ≥ 0 and A[j] > v do
      A[j + 1] ← A[j]
      j ← j - 1
    A[j + 1] ← v
```

The operation of the algorithm is illustrated in Figure 5.4.

The basic operation of the algorithm is the key comparison $A[j] > v$. (Why not $j \geq 0$? Because it will almost certainly be faster than the former in an actual computer implementation. Moreover, it is not germane to the algorithm: a better implementation with a sentinel—see Problem 5 in the exercises—eliminates it altogether.)

The number of key comparisons in this algorithm obviously depends on the nature of the input. In the worst case, $A[j] > v$ is executed the largest number of times, i.e., for every $j = i - 1, \dots, 0$. Since $v = A[i]$, it happens if and only if $A[j] > A[i]$ for $j = i - 1, \dots, 0$. (Note that we are using the fact that on the i th iteration of insertion sort all the elements preceding $A[i]$ are the first i elements in the input, albeit in the sorted order.) Thus, for the worst-case input, we get $A[0] > A[1]$ (for $i = 1$), $A[1] > A[2]$ (for $i = 2$), ..., $A[n - 2] > A[n - 1]$ (for $i = n - 1$). In other words, the worst-case input is an array of strictly decreasing values. The number of key comparisons for such an input is

$$C_{\text{worst}}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

89	45	68	90	29	34	17
45	89	68	90	29	34	17
45	68	89	90	29	34	17
45	68	89	90	29	34	17
29	45	68	89	90	34	17
29	34	45	68	89	90	17
17	29	34	45	68	89	90

FIGURE 5.4 Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

Thus, in the worst case, insertion sort makes exactly the same number of comparisons as selection sort (see Section 3.1).

In the best case, the comparison $A[j] > v$ is executed only once on every iteration of the outer loop. It happens if and only if $A[i - 1] \leq A[i]$ for every $i = 1, \dots, n - 1$, i.e., if the input array is already sorted in ascending order. (Though it “makes sense” that the best case of an algorithm happens when the problem is already solved, it is not always the case: recall our discussion of quicksort in Chapter 4.) Thus, for sorted arrays, the number of key comparisons is

$$C_{\text{best}}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

This very good performance in the best case of sorted arrays is not very useful by itself, because we cannot expect such convenient inputs. However, almost-sorted files arise in a variety of applications, and insertion sort preserves its excellent performance on such inputs. For example, while sorting an array by quicksort, we can stop the algorithm’s iterations after subarrays become smaller than some predefined size (say, 10 elements). By that time, the entire array is almost sorted and we can finish the job by applying insertion sort to it. This modification typically decreases the total running time of quicksort by about 10%.

A rigorous analysis of the algorithm’s average-case efficiency is based on investigating the number of element pairs that are out of order (see Problem 8). It shows that on randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing arrays, i.e.,

$$C_{\text{avg}}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

This twice-as-fast average-case performance coupled with an excellent efficiency on almost-sorted arrays makes insertion sort stand out among its principal competitors among elementary sorting algorithms, selection sort and bubble sort. In addition, its extension named *shellsort*, after its inventor D. L. Shell [She59], gives us an even better algorithm for sorting moderately large files (see Problem 10).

Exercises 5.1



- Ferrying soldiers* A detachment of n soldiers must cross a wide and deep river with no bridge in sight. They notice two 12-year-old boys playing in a rowboat by the shore. The boat is so tiny, however, that it can only hold two boys or one soldier. How can the soldiers get across the river and leave the boys in joint possession of the boat? How many times need the boat pass from shore to shore?



- Alternating glasses* There are $2n$ glasses standing next to each other in a row, the first n of them filled with a soda drink, while the remaining n glasses are empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of glass moves. [Gar78], p. 7



3. Design a decrease-by-one algorithm for generating the power set of a set of n elements. (The power set of a set S is the set of all the subsets of S , including the empty set and S itself.)
4. Apply insertion sort to sort the list $I, N, S, E, R, T, L, O, N$ in alphabetical order.
5.
 - a. What sentinel should be put before the first element of an array being sorted to avoid checking the in-bound condition $j \geq 0$ on each iteration of the inner loop of insertion sort?
 - b. Will the version with the sentinel be in the same efficiency class as the original version?
6. Is it possible to implement insertion sort for sorting linked lists? Will it have the same $O(n^2)$ efficiency as the array version?
7. Consider the following version of insertion sort.

ALGORITHM *InsertSort2($A[0..n - 1]$)*

```

for  $i \leftarrow 1$  to  $n - 1$  do
     $j \leftarrow i - 1$ 
    while  $j \geq 0$  and  $A[j] > A[j + 1]$  do
        swap( $A[j], A[j + 1]$ )
         $j \leftarrow j - 1$ 
    
```

What is its time efficiency? How is it compared to that of the version given in the text?

8. Let $A[1..n]$ be an array of sortable elements. A pair $(A[i], A[j])$ is called as the **index inverted pair** if $A[i] = j$ and $A[j] = i$.
 - a. Find the largest number of index inverted pairs. How many arrays of size n have the largest number of index inverted pairs?
 - b. Find the average number of key comparisons in insertion sort.
9. Derive the best-case efficiency of the binary insertion sort, which uses binary search to insert an element.
10. Shellsort (more accurately Shell's sort) is an important sorting algorithm that works by applying insertion sort to each of several interleaving sublists of a given list. On each pass through the list, the sublists in question are formed by stepping through the list with an increment h_i taken from some predefined decreasing sequence of step sizes, $h_1 > \dots > h_i > \dots > 1$, which must end with 1. (The algorithm works for any such sequence, though some sequences are known to yield a better efficiency than others. For example, the sequence 1, 4, 13, 40, 121, ..., used, of course, in reverse, is known to be among the best for this purpose.)

- a. Apply shellsort to the list

S, H, E, L, L, S, O, R, T, I, S, U, S, E, F, U, L

- b. Is shellsort a stable sorting algorithm?

- c. Implement shellsort, straight insertion sort, binary insertion sort, merge-sort, and quicksort in the language of your choice and compare their performance on random arrays of sizes 10^2 , 10^3 , 10^4 , and 10^5 as well as on increasing and decreasing arrays of these sizes.
-

5.2 Depth-First Search and Breadth-First Search

In the next two sections of this chapter, we deal with very important graph algorithms that can be viewed as applications of the decrease-by-one technique. We assume familiarity with the notion of a graph, its main varieties (undirected, directed, and weighted graphs), the two principal representations of a graph (adjacency matrix and adjacency lists), and such notions as graph connectivity and acyclicity. If needed, a brief review of this material can be found in Section 1.4.

As pointed out in Section 1.3, graphs are interesting structures with a wide variety of applications. Many graph algorithms require processing vertices or edges of a graph in a systematic fashion. There are two principal algorithms for doing such traversals: **depth-first search (DFS)** and **breadth-first search (BFS)**. In addition to doing their main job of visiting vertices and traversing edges of a graph, these algorithms have proved to be very useful in investigating several important properties of a graph.

Depth-First Search

Depth-first search starts visiting vertices of a graph at an arbitrary vertex by marking it as having been visited. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. (If there are several such vertices, a tie can be resolved arbitrarily. As a practical matter, which of the adjacent unvisited candidates is chosen is dictated by the data structure representing the graph. In our examples, we will always break ties by the alphabetical order of the vertices.) This process continues until a dead end—a vertex with no adjacent unvisited vertices—is encountered. At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there. The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth-first search must be restarted at any one of them.

It is convenient to use a stack to trace the operation of depth-first search. We push a vertex onto the stack when the vertex is reached for the first time (i.e., the visit of the vertex starts), and we pop a vertex off the stack when it becomes a dead end (i.e., the visit of the vertex ends).

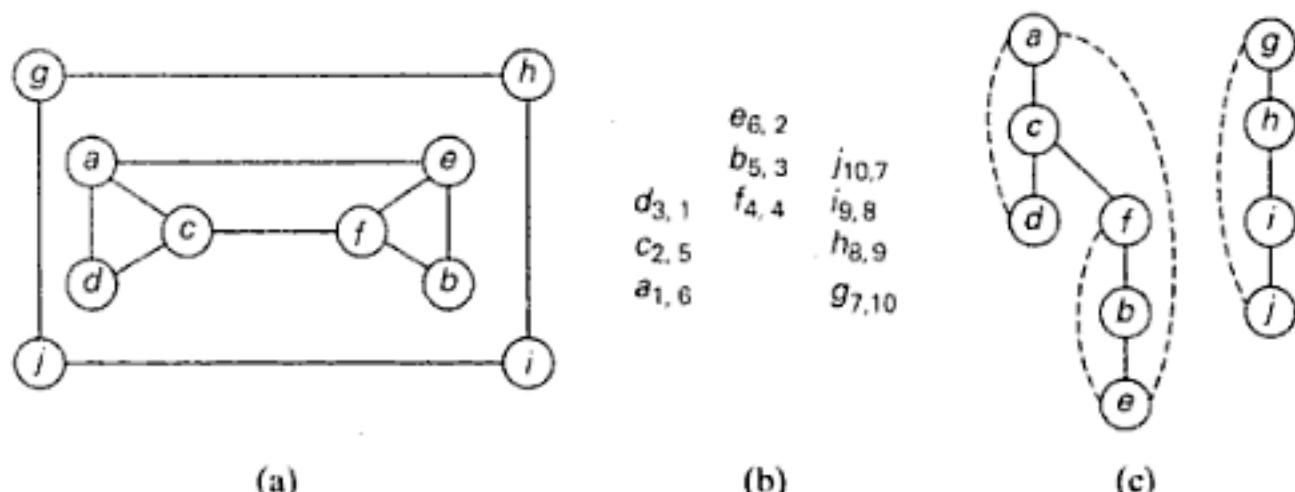


FIGURE 5.5 Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex was visited, i.e., pushed onto the stack; the second one indicates the order in which it became a dead-end, i.e., popped off the stack). (c) DFS forest (with the tree edges shown with solid lines and the back edges shown with dashed lines).

It is also very useful to accompany a depth-first search traversal by constructing the so-called ***depth-first search forest***. The traversal's starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached. Such an edge is called a ***tree edge*** because the set of all such edges forms a forest. The algorithm may also encounter an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree). Such an edge is called a ***back edge*** because it connects a vertex to its ancestor, other than the parent, in the depth-first search forest. Figure 5.5 provides an example of a depth-first search traversal, with the traversal's stack and corresponding depth-first search forest shown as well.

Here is a pseudocode of the depth-first search.

ALGORITHM algorithm

DFS(G)

//Implements a depth-first search traversal of a given graph

//Input: Graph $G = (V, E)$

//Output: Graph G with its vertices marked with consecutive integers

//in the order they've been first encountered by the DES tray

mark each

count \leftarrow 0

each vertex v in V do

if

dfs(v)

100

473(1)

//visits recursively all the unvisited vertices connected to vertex V by a path
 //and numbers them in the order they are encountered.

//and numbers them in the order they are encountered
//via global variable count

//via global variable *collm*

```

count  $\leftarrow$  count + 1; mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)

```

The brevity of the DFS pseudocode and the ease with which it can be performed by hand may create a wrong impression about the level of sophistication of this algorithm. To appreciate its true power and depth, you should trace the algorithm's action by looking not at a graph's diagram but at its adjacency matrix or adjacency lists. (Try it for the graph in Figure 5.5 or a smaller example.)

How efficient is depth-first search? It is not difficult to see that this algorithm is, in fact, quite efficient since it takes just the time proportional to the size of the data structure used for representing the graph in question. Thus, for the adjacency matrix representation, the traversal's time is in $\Theta(|V|^2)$, and for the adjacency list representation, it is in $\Theta(|V|+|E|)$ where $|V|$ and $|E|$ are the number of the graph's vertices and edges, respectively.

A DFS forest, which is obtained as a by-product of a DFS traversal, deserves a few comments, too. To begin with, it is not actually a forest. Rather, we can look at it as the given graph with its edges classified by the DFS traversal into two disjoint classes: tree edges and back edges. (No other types are possible for a DFS forest of an undirected graph.) Again, tree edges are edges used by the DFS traversal to reach previously unvisited vertices. If we consider only the edges in this class, we will indeed get a forest. Back edges connect vertices to previously visited vertices other than their immediate predecessors in the traversal. They connect vertices to their ancestors in the forest other than their parents.

A DFS traversal itself and the forest-like representation of a graph it provides have proved to be extremely helpful for the development of efficient algorithms for checking many important properties of graphs.¹ Note that the DFS yields two orderings of vertices: the order in which the vertices are reached for the first time (pushed onto the stack) and the order in which the vertices become dead ends (popped off the stack). These orders are qualitatively different, and various applications can take advantage of either of them.

Important elementary applications of DFS include checking connectivity and checking acyclicity of a graph. Since DFS halts after visiting all the vertices connected by a path to the starting vertex, checking a graph's connectivity can be done as follows. Start a DFS traversal at an arbitrary vertex and check, after the algorithm halts, whether all the graph's vertices will have been visited. If they have, the graph is connected; otherwise, it is not connected. More generally, we can use DFS for identifying connected components of a graph (how?).

As for checking for a cycle presence in a graph, we can take advantage of the graph's representation in the form of a DFS forest. If the latter does not have back edges, the graph is clearly acyclic. If there is a back edge from some vertex *u* to

1. The discovery of several such applications was an important breakthrough achieved by the two American computer scientists John Hopcroft and Robert Tarjan in the 1970s. For this and other contributions, they subsequently won the Turing Award—the most important prize given in theoretical computer science [Hop87, Tar87].

its ancestor v (e.g., the back edge from d to a in Figure 5.5c), the graph has a cycle that comprises the path from v to u via a sequence of tree edges in the DFS forest followed by the back edge from u to v .

You will find a few other applications of DFS later in the book, although more sophisticated applications, such as finding articulation points of a graph, are not included. (A vertex of a connected graph is said to be its *articulation point* if its removal with all edges incident to it breaks the graph into disjoint pieces.)

Breadth-First Search

If depth-first search is a traversal for the brave (the algorithm goes as far from “home” as it can), breadth-first search is a traversal for the cautious. It proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited. If there still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.

It is convenient to use a queue (note the difference from depth-first search!) to trace the operation of breadth-first search. The queue is initialized with the traversal’s starting vertex, which is marked as visited. On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, and adds them to the queue; after that, the front vertex is removed from the queue.

Similarly to a DFS traversal, it is useful to accompany a BFS traversal by constructing the so-called *breadth-first search forest*. The traversal’s starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called a *tree edge*. If an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree) is encountered, the edge is noted as a *cross edge*. Figure 5.6 provides an example of a breadth-first search traversal, with the traversal’s queue and corresponding breadth-first search forest shown.

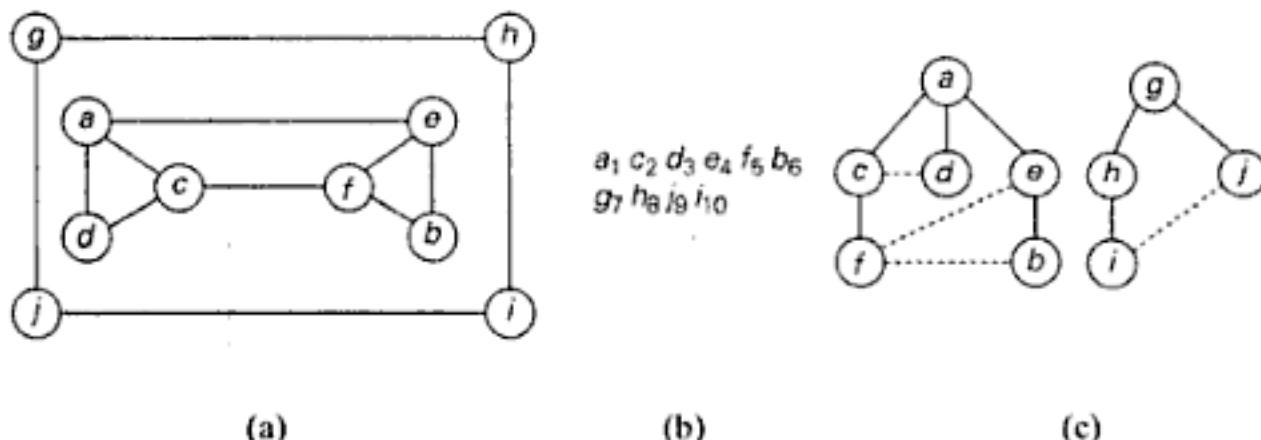


FIGURE 5.6 Example of a BFS traversal. (a) Graph. (b) Traversal’s queue, with the numbers indicating the order in which the vertices were visited, i.e., added to (or removed from) the queue. (c) BFS forest (with the tree edges shown with solid lines and the cross edges shown with dotted lines).

Here is a pseudocode of the breadth-first search.

ALGORITHM *BFS(G)*

```

//Implements a breadth-first search traversal of a given graph
//Input: Graph  $G = (V, E)$ 
//Output: Graph  $G$  with its vertices marked with consecutive integers
//in the order they have been visited by the BFS traversal
mark each vertex in  $V$  with 0 as a mark of being “unvisited”
count  $\leftarrow 0$ 
for each vertex  $v$  in  $V$  do
    if  $v$  is marked with 0
         $bfs(v)$ 
     $bfs(v)$ 
//visits all the unvisited vertices connected to vertex  $v$  by a path
//and assigns them the numbers in the order they are visited
//via global variable count
count  $\leftarrow count + 1$ ; mark  $v$  with  $count$  and initialize a queue with  $v$ 
while the queue is not empty do
    for each vertex  $w$  in  $V$  adjacent to the front vertex do
        if  $w$  is marked with 0
            count  $\leftarrow count + 1$ ; mark  $w$  with  $count$ 
            add  $w$  to the queue
    remove the front vertex from the queue

```

Breadth-first search has the same efficiency as depth-first search: it is in $\Theta(|V|^2)$ for the adjacency matrix representation and in $\Theta(|V| + |E|)$ for the adjacency list representation. Unlike depth-first search, it yields a single ordering of vertices because the queue is a FIFO (first-in first-out) structure and hence the order in which vertices are added to the queue is the same order in which they are removed from it. As to the structure of a BFS forest of an undirected graph, it can also have two kinds of edges: tree edges and cross edges. Tree edges are the ones used to reach previously unvisited vertices. Cross edges connect vertices to those visited before, but, unlike back edges in a DFS tree, they connect vertices either on the same or adjacent levels of a BFS tree.

Finally, BFS can be used to check connectivity and acyclicity of a graph, essentially in the same manner as DFS can. It is not applicable, however, for several less straightforward applications such as finding articulation points. On the other hand, it can be helpful in some situations where DFS cannot. For example, BFS can be used for finding a path with the fewest number of edges between two given vertices. We start a BFS traversal at one of the two vertices given and stop it as soon as the other vertex is reached. The simple path from the root of the BFS tree to the second vertex is the path sought. For example, path $a-b-c-g$ in the graph in Figure 5.7 has the fewest number of edges among all the paths between vertices a and g . Although the correctness of this application appears to stem immediately from the way BFS operates, a mathematical proof of its validity is not quite elementary (see, e.g., [Cor01]).

Table 5.1 summarizes the main facts about depth-first search and breadth-first search.

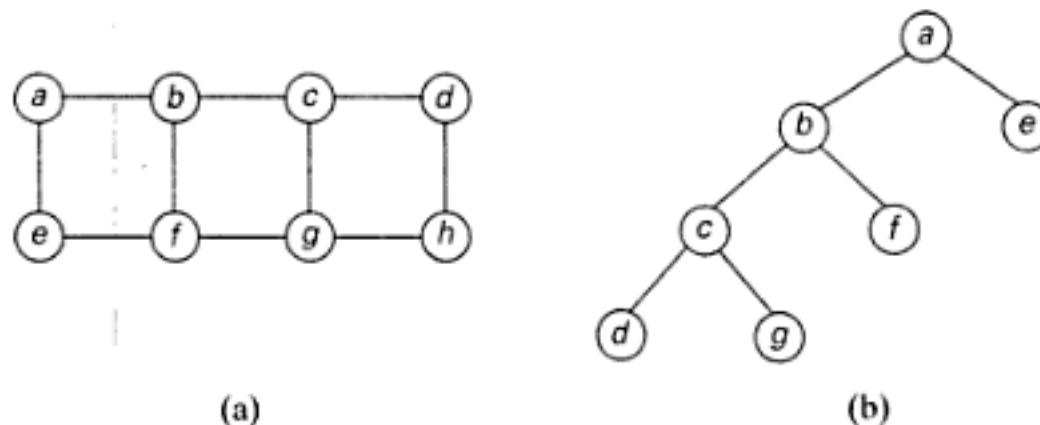


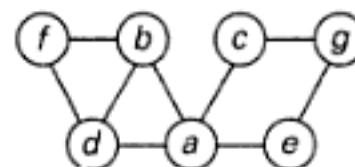
FIGURE 5.7 Illustration of the BFS-based algorithm for finding a minimum-edge path
 (a) Graph. (b) Part of its BFS tree that identifies the minimum-edge path from a to g .

TABLE 5.1. Main facts about depth-first search (DFS) and breadth-first search (BFS)

	DFS	BFS
Data structure	stack	queue
No. of vertex orderings	2 orderings	1 ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity minimum-edge paths
Efficiency for adjacent matrix	$\Theta(V^2)$	$\Theta(V^2)$
Efficiency for adjacent lists	$\Theta(V + E)$	$\Theta(V + E)$

Exercises 5.2

- 1.** Consider the following graph.



- Write down the adjacency matrix and adjacency lists specifying this graph (Assume that the matrix rows and columns and vertices in the adjacency lists follow in the alphabetical order of the vertex labels.)
 - Starting at vertex a and resolving ties by the vertex alphabetical order traverse the graph by depth-first search and construct the corresponding depth-first search tree. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack) and the order in which the vertices became dead ends (popped off the stack).

2. If we define sparse graphs as graphs for which $|E| \in O(|V|)$, which implementation of DFS will have a better time efficiency for such graphs, the one that uses the adjacency matrix or the one that uses the adjacency lists?

3. Let G be a graph with n vertices and m edges.
 - a. True or false: All its DFS forests (for traversals starting at different vertices) will have the same number of trees?
 - b. True or false: All its DFS forests will have the same number of tree edges and the same number of back edges?
4. Traverse the graph of Problem 1 by depth-first search, and construct the corresponding DFS tree. Start the traversal at vertex a and resolve the ties at the vertex in alphabetical order.
5. Define BFS. Explain briefly how it differs from DFS.
6.
 - a. Explain how one can check a graph's acyclicity by using breadth-first search.
 - b. Does either of the two traversals—DFS or BFS—always find a cycle faster than the other? If you answer yes, indicate which of them is better and explain why it is the case; if you answer no, give two examples supporting your answer.
7. Explain how one can identify connected components of a graph by using
 - a. a depth-first search.
 - b. a breadth-first search.
8. A graph is said to be **cyclic** if it contains a cycle in it.
 - a. Design a BFS-based algorithm for checking if a graph is cyclic or not.
 - b. Design a DFS-based algorithm for checking if a graph is cyclic or not.
9. Write a program that, for a given graph, outputs
 - a. vertices of each connected component;
 - b. its cycle or a message that the graph is acyclic.
10. One can model a maze by having a vertex for a starting point, a finishing point, dead ends, and all the points in the maze where more than one path can be taken, and then connecting the vertices according to the paths in the maze.
 - a. Construct such a graph for the following maze.



- b. Which traversal—DFS or BFS—would you use if you found yourself in a maze and why?

5.3 Topological Sorting

In this section, we discuss an important problem for directed graphs. Before we pose this problem though, let us review a few basic facts about directed graphs themselves. A *directed graph*, or *digraph* for short, is a graph with directions specified for all its edges (Figure 5.8a is an example). The adjacency matrix and adjacency lists are still two principal means of representing a digraph. There are only two notable differences between undirected and directed graphs in representing them: (1) the adjacency matrix of a directed graph does not have to be symmetric; (2) an edge in a directed graph has just one (not two) corresponding nodes in the digraph's adjacency lists.

Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs, but the structure of corresponding forests can be more complex. Thus, even for the simple example in Figure 5.8a, the depth-first search forest (Figure 5.8b) exhibits all four types of edges possible in a DFS forest of a directed graph: *tree edges* (ab , bc , de), *back edges* (ba) from vertices to their ancestors, *forward edges* (ac) from vertices to their descendants in the tree other than their children, and *cross edges* (dc), which are none of the aforementioned types.

Note that a back edge in a DFS forest of a directed graph can connect a vertex to its parent. Whether or not it is the case, the presence of a back edge indicates that the digraph has a directed cycle. (A *directed cycle* in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor.) Conversely, if a DFS forest of a digraph has no back edges, the digraph is a *dag*, an acronym for *directed acyclic graph*.

Directions on a graph's edges lead to new questions about the graph that are either meaningless or trivial for undirected graphs. In this section, we discuss one such problem. As a motivating example, consider a set of five required courses $\{C1, C2, C3, C4, C5\}$ a part-time student has to take in some degree program. The

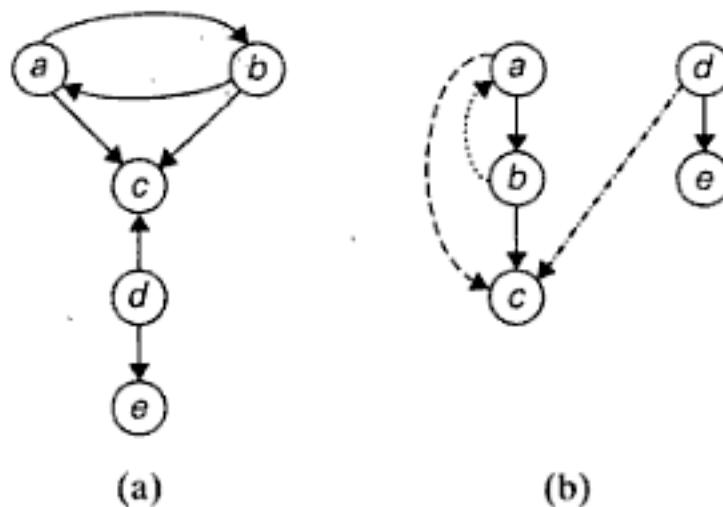


FIGURE 5.8 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a .

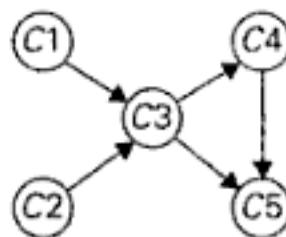


FIGURE 5.9 Digraph representing the prerequisite structure of five courses

courses can be taken in any order as long as the following course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4. The student can take only one course per term. In which order should the student take the courses?

The situation can be modeled by a digraph in which vertices represent courses and directed edges indicate prerequisite requirements (Figure 5.9). In terms of this digraph, the question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends. (Can you find such an ordering of this digraph's vertices?) This problem is called **topological sorting**. It can be posed for an arbitrary digraph, but it is easy to see that the problem cannot have a solution if a digraph has a directed cycle. Thus, for topological sorting to be possible, a digraph must be a dag. It turns out that being a dag is not only necessary but also sufficient for topological sorting to be possible; i.e., if a digraph has no cycles, the topological sorting problem for it has a solution. Moreover, there are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem.

The first algorithm is a simple application of depth-first search: perform a DFS traversal and note the order in which vertices become dead ends (i.e., are popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

Why does the algorithm work? When a vertex v is popped off a DFS stack, no vertex u with an edge from u to v can be among the vertices popped off before v . (Otherwise, (u, v) would have been a back edge.) Hence, any such vertex u will be listed after v in the popped-off order list, and before v in the reversed list.

Figure 5.10 illustrates an application of this algorithm to the digraph in Figure 5.9. Note that in Figure 5.10c, we have drawn the edges of the digraph, and they all point from left to right as the problem's statement requires. It is a convenient way to check visually the correctness of a solution to an instance of the topological sorting problem.

The second algorithm is based on a direct implementation of the decrease (by one)-and-conquer technique: repeatedly, identify in a remaining digraph a **source**, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. If there is none, stop because the problem cannot be solved—see Problem 6a.) The order in which the vertices are deleted yields a solution to the topological sorting

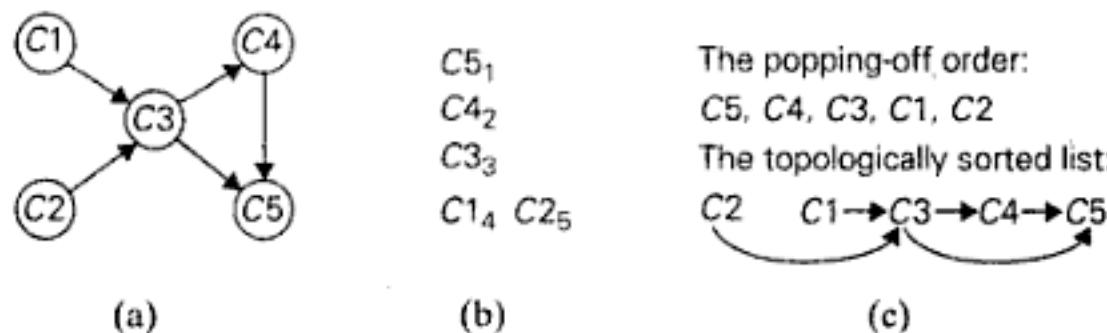


FIGURE 5.10 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

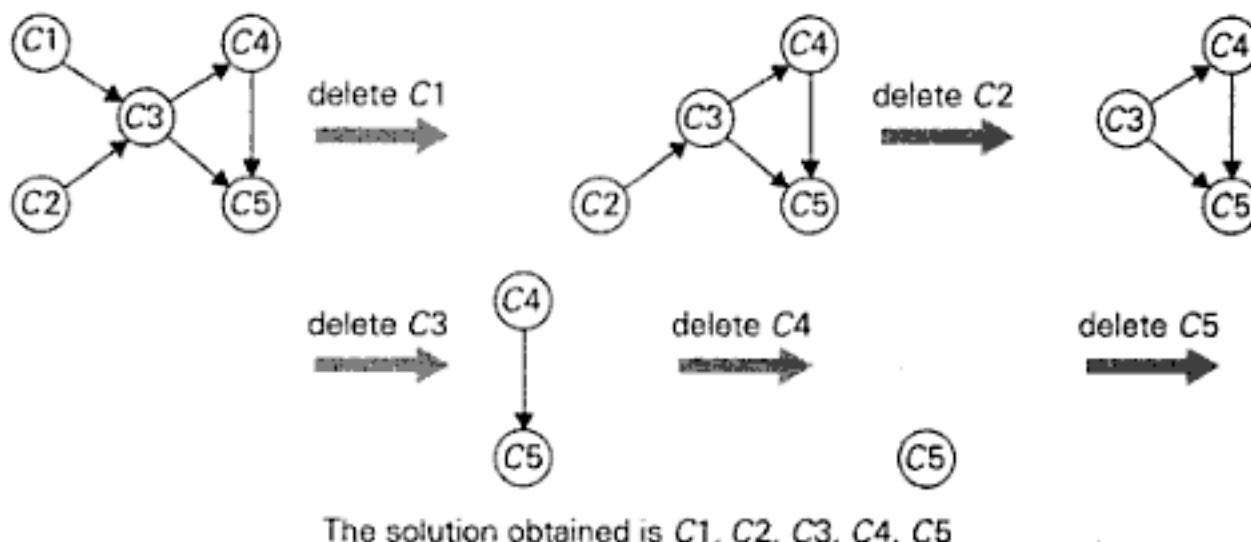


FIGURE 5.11 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

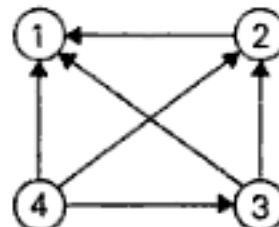
problem. The application of this algorithm to the same digraph representing the five courses is given in Figure 5.11.

Note that the solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, of course; the topological sorting problem may have several alternative solutions.

The tiny size of the example we used might create a wrong impression about the topological sorting problem. But imagine a large project—e.g., in construction or research—that involves thousands of interrelated tasks with known prerequisites. The first thing you should do in such a situation is to make sure that the set of given prerequisites is not contradictory. The convenient way of doing this is to solve the topological sorting problem for the project's digraph. Only then can you start thinking about scheduling your tasks to, say, minimize the total completion time of the project. This would require, of course, other algorithms that you can find in general books on operations research or in special ones on so-called CPM (Critical Path Method) and PERT (Program Evaluation and Review Technique) methodologies.

Exercises 5.3

1. Apply the DFS-based algorithm to solve the topological sorting problem for the following digraphs.



2. a. Prove that the topological sorting problem has a solution for a digraph if and only if it is a dag.
b. For a digraph with n vertices, what is the largest number of distinct solutions the topological sorting problem can have?
3. a. What is the time efficiency of the DFS-based algorithm for topological sorting?
b. How can one modify the DFS-based algorithm to avoid reversing the vertex ordering generated by DFS?
4. Can one use the order in which vertices are pushed onto the DFS stack (instead of the order they are popped off it) to solve the topological sorting problem?
5. Apply the source-removal algorithm to the digraphs of Problem 1.
6. a. Prove that a dag must have at least one source.
b. How would you find a source (or determine that such a vertex does not exist) in a digraph represented by its adjacency matrix? What is the time efficiency of this operation?
c. How would you find a source (or determine that such a vertex does not exist) in a digraph represented by its adjacency lists? What is the time efficiency of this operation?
7. Can you implement the source-removal algorithm for a digraph represented by its adjacency lists so that its running time is in $O(|V| + |E|)$?
8. Explain the differences between directed graphs and undirected graphs.
9. A digraph is called ***strongly connected*** if for any pair of two distinct vertices u and v there exists a directed path from u to v and a directed path from v to u . In general, a digraph's vertices can be partitioned into disjoint maximal subsets of vertices that are mutually accessible via directed paths of the digraph; these subsets are called ***strongly connected components***. There are two DFS-based algorithms for identifying strongly connected components. Here is the simpler (but somewhat less efficient) one of the two.

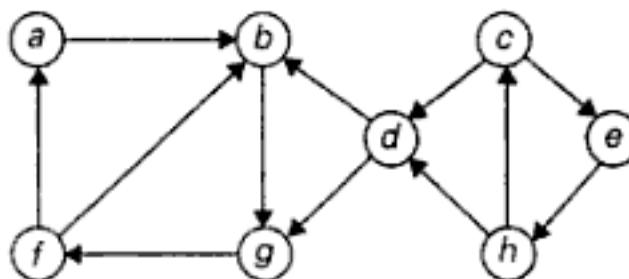
Step 1 Do a DFS traversal of the digraph given and number its vertices in the order that they become dead ends.

Step 2 Reverse the directions of all the edges of the digraph.

Step 3 Do a DFS traversal of the new digraph by starting (and, if necessary, restarting) the traversal at the highest numbered vertex among still unvisited vertices.

The strongly connected components are exactly the subsets of vertices in each DFS tree obtained during the last traversal.

- Apply this algorithm to the following digraph to determine its strongly connected components.



- What is the time efficiency class of this algorithm? Give separate answers for the adjacency matrix representation and adjacency list representation of an input graph.
 - How many strongly connected components does a dag have?
10. A legend among a group of n people is a person who is top rated in all aspects. The task is to identify a legend by asking a single question of the form “Who deserves it?” Design an efficient algorithm to identify a legend or determine if the group has no such person. How many questions does your algorithm need in the worst-case?

5.4 Algorithms for Generating Combinatorial Objects

In this section, we keep our promise to discuss algorithms for generating combinatorial objects. The most important types of combinatorial objects are permutations, combinations, and subsets of a given set. They typically arise in problems that require a consideration of different choices. We already encountered them in Chapter 3 when we discussed exhaustive search. Combinatorial objects are studied in a branch of discrete mathematics called combinatorics. Mathematicians, of course, are primarily interested in different counting formulas; we should be grateful for such formulas because they tell us how many items need to be generated. (In particular, they warn us that the number of combinatorial objects typically grows exponentially or even faster as a function of the problem’s size.) But our primary interest here lies in algorithms for generating combinatorial objects, not just in counting them.

Generating Permutations

We start with permutations. For simplicity, we assume that the underlying set whose elements need to be permuted is simply the set of integers from 1 to n ; more generally, they can be interpreted as indices of elements in an n -element set $\{a_1, \dots, a_n\}$. What would the decrease-by-one technique suggest for the problem of generating all $n!$ permutations of $\{1, \dots, n\}$? The smaller-by-one problem is to generate all $(n - 1)!$ permutations. Assuming that the smaller problem is solved, we can get a solution to the larger one by inserting n in each of the n possible positions among elements of every permutation of $n - 1$ elements. All the permutations obtained in this fashion will be distinct (why?), and their total number will be $n(n - 1)! = n!$. Hence, we will obtain all the permutations of $\{1, \dots, n\}$.

We can insert n in the previously generated permutations either left to right or right to left. It turns out that it is beneficial to start with inserting n into $12\dots(n-1)$ by moving right to left and then switch direction every time a new permutation of $\{1, \dots, n - 1\}$ needs to be processed. An example of applying this approach bottom up for $n = 3$ is given in Figure 5.12.

The advantage of this order stems from the fact that it satisfies the *minimal-change* requirement: each permutation can be obtained from its immediate predecessor by exchanging just two elements in it. (Check this for the permutations generated in Figure 5.12.) The minimal-change requirement is beneficial both for the algorithm's speed and for applications using the permutations. For example, in Section 3.4, we needed permutations of cities to solve the traveling salesman problem by exhaustive search. If such permutations are generated by a minimal-change algorithm, we can compute the length of a new tour from the length of its predecessor in constant rather than linear time (how?).

It is possible to get the same ordering of permutations of n elements without explicitly generating permutations for smaller values of n . It can be done by associating a direction with each element k in a permutation. We indicate such a direction by a small arrow written above the element in question, e.g.,

$\overrightarrow{3} \overleftarrow{2} \overrightarrow{4} \overleftarrow{1}$.

The element k is said to be *mobile* in such an arrow-marked permutation if its arrow points to a smaller number adjacent to it. For example, for the permutation $\overrightarrow{3} \overleftarrow{2} \overrightarrow{4} \overleftarrow{1}$, 3 and 4 are mobile while 2 and 1 are not. Using the notion of a mobile element, we can give the following description of the *Johnson-Trotter algorithm* for generating permutations.

start		1
insert 2 into 1 right to left	12	21
insert 3 into 21 right to left	123	132
insert 3 into 21 left to right	321	231
		213

FIGURE 5.12 Generating permutations bottom up

ALGORITHM *JohnsonTrotter(n)*

```

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of {1, ..., n}
initialize the first permutation with  $\overleftarrow{1} \overleftarrow{2} \dots \overleftarrow{n}$ 
while the last permutation has a mobile element do
    find its largest mobile element k
    swap k and the adjacent integer k's arrow points to
    reverse the direction of all the elements that are larger than k
    add the new permutation to the list

```

Here is an application of this algorithm for $n = 3$, with the largest mobile integer shown in bold:

$$\overleftarrow{1} \overleftarrow{2} \overleftarrow{3} \quad \overleftarrow{1} \overleftarrow{3} \overleftarrow{2} \quad \overleftarrow{3} \overleftarrow{1} \overleftarrow{2} \quad \overrightarrow{3} \overleftarrow{2} \overleftarrow{1} \quad \overleftarrow{2} \overrightarrow{3} \overleftarrow{1} \quad \overrightarrow{2} \overrightarrow{1} \overrightarrow{3}.$$

This algorithm is one of the most efficient for generating permutations; it can be implemented to run in time proportional to the number of permutations, i.e., in $\Theta(n!)$. Of course, it is horribly slow for all but very small values of n ; however, this is not the algorithm's fault but rather the "fault" of the problem: it simply asks to generate too many items.

One can argue that the permutation ordering generated by the Johnson-Trotter algorithm is not quite natural; e.g., the natural place for permutation $n \ n-1 \dots 1$ seems to be the last one on the list. This would be the case if permutations were listed in increasing order—also called the *lexicographic order*—which is the order in which they would be listed in a dictionary if the numbers were interpreted as letters of an alphabet:

$$123 \ 132 \ 213 \ 231 \ 312 \ 321.$$

So how can we generate the permutation following $a_1 a_2 \dots a_{n-1} a_n$ in lexicographic order? If $a_{n-1} < a_n$, we can simply transpose these last two elements. For example, 123 is followed by 132. If $a_{n-1} > a_n$, we have to engage a_{n-2} . If $a_{n-2} < a_{n-1}$, we should rearrange the last three elements by increasing the $(n-2)$ th element as little as possible by putting there the next larger than a_{n-2} element chosen from a_{n-1} and a_n and filling positions $n-1$ and n with the remaining two of the three elements a_{n-2} , a_{n-1} , and a_n in increasing order. For example, 132 is followed by 213 while 231 is followed by 312. In general, we scan a current permutation from right to left looking for the first pair of consecutive elements a_i and a_{i+1} such that $a_i < a_{i+1}$ (and, hence, $a_{i+1} > \dots > a_n$). Then we find the smallest element in the tail that is larger than a_i , i.e., $\min\{a_j \mid a_j > a_i, j > i\}$, and put it in position i ; the positions from $i+1$ through n are filled with the elements a_i, a_{i+1}, \dots, a_n , from which the element put in the i th position has been eliminated, in increasing order. For example, 163542 would be followed by 164235. We leave writing a complete pseudocode of this algorithm for the exercises.

Generating Subsets

Recall that in Section 3.4 we examined the knapsack problem that asks to find the most valuable subset of items that fits a knapsack of a given capacity. The exhaustive-search approach to solving this problem discussed there was based on generating all subsets of a given set of items. In this section, we discuss algorithms for generating all 2^n subsets of an abstract set $A = \{a_1, \dots, a_n\}$. (Mathematicians call the set of all subsets of a set its *power set*.)

The decrease-by-one idea is immediately applicable to this problem, too. All subsets of $A = \{a_1, \dots, a_n\}$ can be divided into two groups: those that do not contain a_n and those that do. The former group is nothing but all the subsets of $\{a_1, \dots, a_{n-1}\}$, while each and every element of the latter can be obtained by adding a_n to a subset of $\{a_1, \dots, a_{n-1}\}$. Thus, once we have a list of all subsets of $\{a_1, \dots, a_{n-1}\}$, we can get all the subsets of $\{a_1, \dots, a_n\}$ by adding to the list all its elements with a_n put into each of them. An application of this algorithm to generate all subsets of $\{a_1, a_2, a_3\}$ is illustrated in Figure 5.13.

Similarly to generating permutations, we do not have to generate power sets of smaller sets. A convenient way of solving the problem directly is based on a one-to-one correspondence between all 2^n subsets of an n element set $A = \{a_1, \dots, a_n\}$ and all 2^n bit strings b_1, \dots, b_n of length n . The easiest way to establish such a correspondence is to assign to a subset the bit string in which $b_i = 1$ if a_i belongs to the subset and $b_i = 0$ if a_i does not belong to it. (We mentioned this idea of bit vectors in Section 1.4.) For example, the bit string 000 will correspond to the empty subset of a three-element set, 111 will correspond to the set itself, i.e., $\{a_1, a_2, a_3\}$, while 110 will represent $\{a_1, a_2\}$. With this correspondence in place, we can generate all the bit strings of length n by generating successive binary numbers from 0 to $2^n - 1$, padded, when necessary, with an appropriate number of leading 0's. For example, for the case of $n = 3$, we obtain

bit strings	000	001	010	011	100	101	110	111
subsets	\emptyset	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

Note that while the bit strings are generated by this algorithm in the lexicographic order (in the two-symbol alphabet of 0 and 1), the order of the subsets looks anything but natural. For example, we might want the so-called *squashed order*, in which any subset involving a_j can be listed only after all the subsets

n	subsets							
0	\emptyset							
1	\emptyset	$\{a_1\}$						
2	\emptyset	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	\emptyset	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

FIGURE 5.13 Generating subsets bottom up

involving a_1, \dots, a_{j-1} , as was the case for the list of the three-element set in Figure 5.13. It is easy to adjust the bit string-based algorithm to yield a squashed ordering of the subsets involved (Problem 6).

A more challenging question is whether there exists a minimal-change algorithm for generating bit strings so that every one of them differs from its immediate predecessor by only a single bit. (In the language of subsets, we want every subset to differ from its immediate predecessor by either an addition or a deletion, but not both, of a single element). The answer to this question is yes (Problem 9); for example, for $n = 3$, we can get

000 001 011 010 110 111 101 100.

Such a sequence of bit strings is called the *binary reflected Gray code*. Gray codes have many interesting properties and a few useful applications; you can read about them in such books as [Bru04].

Exercises 5.4

1. Is it realistic to implement an algorithm that requires generating all permutations of a 25-element set on your computer? What about all the subsets of such a set?
2. Generate all permutations of $\{1, 2, 3, 4\}$ by
 - a. the bottom-up minimal-change algorithm.
 - b. the Johnson-Trotter algorithm.
 - c. the lexicographic-order algorithm.
3. Write a program for generating combinations in lexicographic order.
4. Consider the following implementation of the algorithm for generating permutations discovered by B. Heap [Hea63].

ALGORITHM *HeapPermute(n)*

```
//Implements Heap's algorithm for generating permutations
//Input: A positive integer n and a global array A[1..n]
//Output: All permutations of elements of A
if n = 1
    write A
else
    for i ← 1 to n do
        HeapPermute(n - 1)
        if n is odd
            swap A[1] and A[n]
        else swap A[i] and A[n]
```

- a. Trace the algorithm by hand for $n = 2, 3$, and 4 .
 - b. Prove correctness of Heap's algorithm.
 - c. What is the time efficiency of *HeapPermute*?
5. Generate all the subsets of a four-element set $A = \{a_1, a_2, a_3, a_4\}$ by each of the two algorithms outlined in this section.
 6. What simple trick would make the bit string-based algorithm generate subsets in squashed order?
 7. Write a pseudocode for nonrecursive algorithm for generating all 2^n bit strings of length n .
 8. Write a nonrecursive algorithm for generating 2^n bit strings of length n that implements bit strings as arrays and does not use binary additions.
 9. a. Use the decrease-by-one technique to generate a Gray code for $n = 4$.
b. Design a general decrease-by-one algorithm for generating a Gray code of order n .
 10. Design a decrease-and-conquer algorithm for generating all permutations of k items chosen from n , i.e., all k -element subsets of a given n -element set. Is your algorithm a minimal-change algorithm?
- 11. Gray code and the Tower of Hanoi**
- a. Show that the disk moves made in the classic recursive algorithm for the Tower of Hanoi puzzle can be used for generating the binary reflected Gray code.
 - b. Show how the binary reflected Gray code can be used for solving the Tower of Hanoi puzzle.

5.5 Decrease-by-a-Constant-Factor Algorithms

You may recall from the introduction to this chapter that decrease-by-a-constant-factor is the second major variety of decrease-and-conquer. You have already encountered examples of this design technique in this book: binary search (Section 4.3) and exponentiation by squaring (introduction to Section 5.1). In this section, you will find a few other examples of algorithms based on the decrease-by-a-constant-factor idea. We should not expect a wealth of examples of this kind, however, because these algorithms are usually logarithmic and, being very fast, do not happen often; a reduction by a factor other than two is especially rare.

Fake-Coin Problem

Of several versions of the fake-coin identification problem, we consider the one that best illustrates the decrease-by-a-constant-factor strategy. Among n identically looking coins, one is fake. With a balance scale, we can compare any two sets of coins. That is, by tipping to the left, to the right, or staying even, the balance

<i>n</i>	<i>m</i>		<i>n</i>	<i>m</i>
50	65		50	65
25	130		25	130
12	260	(+130)	12	260
6	520		6	520
3	1,040		3	1,040
1	2,080	(+1040)	1	2,080
	2,080	+ (130 + 1040) = 3,250		3,250

(a)

(b)

FIGURE 5.14 Computing $50 \cdot 65$ by multiplication à la russe

If n is odd, we need only a slight adjustment of this formula:

$$n \cdot m = \frac{n-1}{2} \cdot 2m + m.$$

Using these formulas and the trivial case of $1 \cdot m = m$ to stop, we can compute product $n \cdot m$ either recursively or iteratively. An example of computing $50 \cdot 65$ with this algorithm is given in Figure 5.14. Note that all the extra addends shown in parentheses in Figure 5.14a are in the rows that have odd values in the first column. Therefore we can find the product by simply adding all the elements in the m column that have an odd number in the n column (Figure 5.14b).

Also note that the algorithm involves just the simple operations of halving, doubling, and adding—a feature that might be attractive, for example, to those who do not want to memorize the table of multiplications. It is this feature of the algorithm that most probably made it attractive to Russian peasants who, according to Western visitors, used it widely in the nineteenth century, and for whom the method is named. (In fact, the algorithm's idea was used by Egyptian mathematicians as early as 1650 b.c. [Cha98], p. 16.) It also leads to very fast hardware implementation since doubling and halving of binary numbers can be performed using shifts, which are among the most basic operations at the machine level.

Josephus Problem

Our last example is the *Josephus problem*, named for Flavius Josephus, a famous Jewish historian who participated in and chronicled the Jewish revolt of 66–70 C.E. against the Romans. Josephus, as a general, managed to hold the fortress of Jotapata for 47 days, but after the fall of the city he took refuge with 40 diehards in a nearby cave. There, the rebels voted to perish rather than surrender. Josephus proposed that each man in turn should dispatch his neighbor, the order to be determined by casting lots. Josephus contrived to draw the last lot, and, as one of the two surviving men in the cave, he prevailed upon his intended victim to surrender to the Romans.

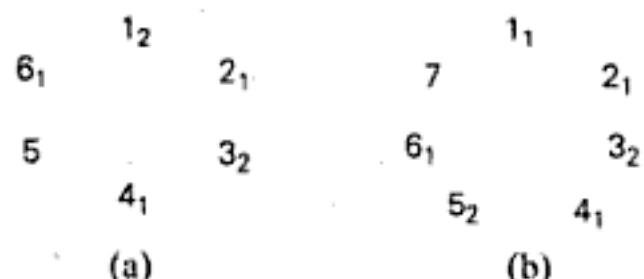


FIGURE 5.15 Instances of the Josephus problem for (a) $n = 6$ and (b) $n = 7$. Subscript numbers indicate the pass on which the person in that position is eliminated. The solutions are $J(6) = 5$ and $J(7) = 7$, respectively.

So let n people numbered 1 to n stand in a circle. Starting the grim count with person number 1, we eliminate every second person until only one survivor is left. The problem is to determine the survivor's number $J(n)$. For example (Figure 5.15), if n is 6, people in positions 2, 4, and 6 will be eliminated on the first pass through the circle, and people in initial positions 3 and 1 will be eliminated on the second pass, leaving a sole survivor in initial position 5—thus, $J(6) = 5$. To give another example, if n is 7, people in positions 2, 4, 6, and 1 will be eliminated on the first pass (it is more convenient to include 1 in the first pass) and people in positions 5 and, for convenience, 3 on the second—thus, $J(7) = 7$.

It is convenient to consider the cases of even and odd n 's separately. If n is even, i.e., $n = 2k$, the first pass through the circle yields an instance of exactly the same problem but half its initial size. The only difference is in position numbering; for example, a person in initial position 3 will be in position 2 for the second pass, a person in initial position 5 will be in position 3, and so on (check Figure 5.15a). It is easy to see that to get the initial position of a person, we simply need to multiply his new position by two and subtract one. This relationship will hold, in particular, for the survivor, i.e.,

$$J(2k) \equiv 2J(k) = 1.$$

Let us now consider the case of an odd n ($n > 1$), i.e., $n = 2k + 1$. The first pass eliminates people in all even positions. If we add to this the elimination of the person in position 1 right after that, we are left with an instance of size k . Here, to get the initial position that corresponds to the new position numbering, we have to multiply the new position number by two and add one (check Figure 5.15b). Thus, for odd values of n , we get

$$J(2k+1) \equiv 2J(k)+1.$$

Can we get a closed-form solution to the two-case recurrence (subject to the initial condition $J(1) = 1$)? The answer is yes, though getting it requires more ingenuity than just applying backward substitutions. In fact, one way to find a solution is to apply forward substitutions to get, say, the first 15 values of $J(n)$, discern a pattern, and then prove its general validity by mathematical induction. We leave the execution of this plan to the exercises; alternatively, you can look it up in [Gra94], whose exposition of the Josephus problem we have been following. Interestingly, the most elegant form of the closed-form answer involves the binary

representation of size n : $J(n)$ can be obtained by a one-bit cyclic shift left of n itself! For example, $J(6) = J(110_2) = 101_2 = 5$ and $J(7) = J(111_2) = 111_2 = 7$.

Exercises 5.5



1. Design a divide-into-three algorithm for computing $\lceil \log_2 n \rceil$ and determine its efficiency.
2. Consider ***ternary search***—the following algorithm for searching in a sorted array $A[0..n - 1]$. If $n = 1$, simply compare the search key K with the single element of the array; otherwise, search recursively by comparing K with $A[\lfloor n/3 \rfloor]$, and if K is larger, compare it with $A[\lfloor 2n/3 \rfloor]$ to determine in which third of the array to continue the search.
 - a. What design technique is this algorithm based on?
 - b. Set up a recurrence for the number of key comparisons in the worst case. (You may assume that $n = 3^k$.)
 - c. Solve the recurrence for $n = 3^k$.
 - d. Compare this algorithm's efficiency with that of binary search.
3. a. Write a pseudocode for the divide-into-three algorithm for the fake-coin problem. (Make sure that your algorithm handles properly all values of n , not only those that are multiples of 3.)
 - b. Set up a recurrence relation for the number of weighings in the divide-into-three algorithm for the fake-coin problem and solve it for $n = 3^k$.
 - c. For large values of n , about how many times faster is this algorithm than the one based on dividing coins into two piles? (Your answer should not depend on n .)
4. Apply multiplication à la russe to compute $26 \cdot 47$.
5. a. From the standpoint of time efficiency, does it matter whether we multiply n by m or m by n by the multiplication à la russe algorithm?
 - b. What is the efficiency class of multiplication à la russe?
6. Write a pseudocode for the multiplication à la russe algorithm.
7. Find $J(30)$ —the solution to the Josephus problem for $n = 30$.
8. Prove that the solution to the Josephus problem is not 1 for every n that is not a power of 2.
9. For the Josephus problem,
 - a. compute $J(n)$ for $n = 1, 2, \dots, 15$.
 - b. discern a pattern in the solutions for the first fifteen values of n and prove its general validity.
 - c. prove the validity of getting $J(n)$ by a one-bit cyclic shift left of the binary representation of n .

5.6 Variable-Size-Decrease Algorithms

As mentioned in the introduction to this chapter, in the third principal variety of decrease-and-conquer, the size reduction pattern varies from one iteration of the algorithm to another. Euclid's algorithm for computing the greatest common divisor (Section 1.1) provides a good example of this kind of algorithm. In this section, we encounter a few more examples of this variety.

Computing a Median and the Selection Problem

The **selection problem** is the problem of finding the k th smallest element in a list of n numbers. This number is called the k th **order statistic**. Of course, for $k = 1$ or $k = n$, we can simply scan the list in question to find the smallest or largest element, respectively. A more interesting case of this problem is for $k = \lceil n/2 \rceil$, which asks to find an element that is greater than one half of the list's elements and smaller than the other half. This middle value is called the **median**, and it is one of the most important quantities in mathematical statistics. Obviously, we can find the k th smallest element in a list by sorting the list first and then selecting the k th element in the output of a sorting algorithm. The time of such an algorithm is determined by the efficiency of the sorting algorithm used. Thus, with a good sorting algorithm such as mergesort, the algorithm's efficiency is in $O(n \log n)$.

You should immediately suspect, however, that sorting the entire list is most likely overkill since the problem asks not to order the entire list but just to find its k th smallest element. Fortunately, we do have a very efficient (on average) algorithm for doing a similar task of partitioning an array's elements into two subsets: the one containing the elements that are less than or equal to some value p pivoting the partition and the other containing the elements that are greater than or equal to p :

$$\underbrace{a_{i_1} \dots a_{i_{s-1}}}_{\leq p} \ p \ \underbrace{a_{i_{s+1}} \dots a_{i_n}}_{\geq p}.$$

Such partitioning was the principal part of quicksort, discussed in Chapter 4.

How can we take advantage of a list's partition? Let s be the partition's split position. If $s = k$, the pivot p obviously solves the selection problem. (Had we indexed the list starting at 0, it would have been $s = k - 1$, of course.) If $s > k$, the k th smallest element in the entire list can be found as the k th smallest element in the left part of the partitioned array. And if $s < k$, we can proceed by searching for the $(k - s)$ th smallest element in its right part. Thus, if we do not solve the problem outright, we reduce its instance to a smaller one, which can be solved by the same approach, i.e., recursively. In fact, the same idea can be implemented without recursion as well. For the nonrecursive version, we need not even adjust the value of k but just continue until $s = k$.

EXAMPLE Find the median of the following list of nine numbers: 4, 1, 10, 9, 7, 12, 8, 2, 15. Here, $k = \lceil 9/2 \rceil = 5$ and our task is to find the fifth smallest element in the array. As earlier, we assume for the sake of convenience that the elements of the list are indexed from 1 to 9.

We use the same version of array partitioning that we used in our discussion of quicksort in Chapter 4, which selects the first element as a pivot and rearranges elements by two oppositely directed scans of the array:

4	1	10	9	7	12	8	2	15
2	1	4	9	7	12	8	10	15

Since $s = 3 < k = 5$, we proceed with the right part of the list:

9	7	12	8	10	15
8	7	9	12	10	15

Since $s = 6 > k = 5$, we continue with the left part of the previous sublist:

8	7
7	8

Now $s = k = 5$, and hence we can stop: the found median is 8, which is greater than 2, 1, 4, and 7 but smaller than 9, 12, 10, and 15. ■

How efficient is this algorithm? We should expect it to be more efficient than quicksort in the average case because it has to deal with just a single subarray after a partition while quicksort has to work on two of them. Had the splits always happened in the middle of a remaining array, the recurrence for the number of comparisons would have been

$$C(n) = C(n/2) + (n + 1),$$

whose solution, according to the Master Theorem (see Chapter 4), is in $\Theta(n)$. Although the array's size is actually reduced in an unpredictable fashion from one iteration of the algorithm to another (with some size reductions less than half and some larger), a careful mathematical analysis shows the average-case efficiency to be the same as it would be had the size always been reduced by one half. In other words, the algorithm turns out to be linear in the average case. In the worst case, we have the same embarrassing deterioration of efficiency into $\Theta(n^2)$. Though computer scientists have discovered an algorithm that works in linear time even in the worst case [Blo73], it is too complicated to be recommended for practical applications.

Note also that the partitioning-based algorithm solves a somewhat more general problem of identifying the k smallest and $n - k$ largest elements of a given list, not just the value of its k th smallest element.

Interpolation Search

As the next example of a variable-size-decrease algorithm, we consider an algorithm for searching in a sorted array called *interpolation search*. Unlike binary search, which always compares a search key with the middle value of a given sorted array (and hence reduces the problem's instance size by half), interpolation search

takes into account the value of the search key in order to find the array's element to be compared with the search key. In a sense, the algorithm mimics the way we search for a name in a telephone book: if we are searching for someone named Brown, we open the book not in the middle but very close to the beginning, unlike our action when searching for someone named, say, Smith.

More precisely, on the iteration dealing with the array's portion between the leftmost element $A[l]$ and the rightmost element $A[r]$, the algorithm assumes that the array's values increase linearly, i.e., along the straight line through the points $(l, A[l])$ and $(r, A[r])$. (The accuracy of this assumption can influence the algorithm's efficiency but not its correctness.) Accordingly, the search key's value v is compared with the element whose index is computed as (the roundoff of) the x coordinate of the point on the straight line through the points $(l, A[l])$ and $(r, A[r])$ whose y coordinate is equal to the search value v (Figure 5.16).

Writing a standard equation for the straight line passing through the points $(l, A[l])$ and $(r, A[r])$, substituting v for y , and solving it for x leads to the following formula:

$$x = l + \lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \rfloor. \quad (5.4)$$

The logic behind this approach is quite straightforward. We know that the array values are increasing (more accurately, not decreasing) from $A[l]$ to $A[r]$, but we do not know how they do it. Had the array's values increased linearly, which is the simplest manner possible, the index computed by formula (5.4) would be the expected location of the array's element with the value equal to v . Of course, if v is not between $A[l]$ and $A[r]$, formula (5.4) need not be applied (why?).

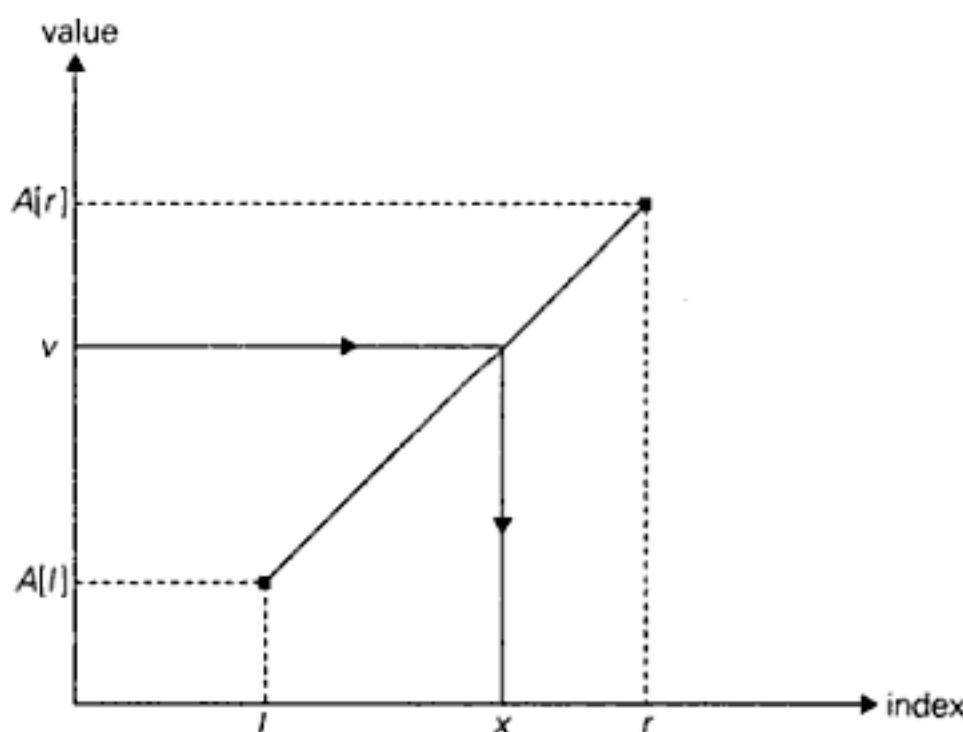


FIGURE 5.16 Index computation in interpolation search

After comparing v with $A[x]$, the algorithm either stops (if they are equal) or proceeds by searching in the same manner among the elements indexed either between l and $x - 1$ or between $x + 1$ and r , depending on whether $A[x]$ is smaller or larger than v . Thus, the size of the problem's instance is reduced, but we cannot tell a priori by how much.

The analysis of the algorithm's efficiency shows that interpolation search uses fewer than $\log_2 \log_2 n + 1$ key comparisons on the average when searching in a list of n random keys. This function grows so slowly that the number of comparisons will be a very small constant for all practically feasible inputs (see Problem 6). But in the worst case, interpolation search is only linear, which must be considered as a bad performance (why?). As a final assessment of the worthiness of interpolation search versus that of binary search, we can point to an opinion by R. Sedgewick [Sed88] that binary search is probably better for smaller files but interpolation search is worth considering for large files and for applications where comparisons are particularly expensive or access costs are very high. Note that in Section 12.4, we discuss a continuous counterpart of interpolation search, which can be seen as one more example of a variable-size-decrease algorithm.

Searching and Insertion in a Binary Search Tree

As the last example of this section, let us revisit the binary search tree. Recall that this is a binary tree whose nodes contain elements of a set of orderable items, one element per node, so that for every node all elements in the left subtree are smaller and all the elements in the right subtree are greater than the element in the subtree's root. When we need to search for an element of a given value (say, v) in such a tree, we do it recursively in the following manner. If the tree is empty, the search ends in failure. If the tree is not empty, we compare v with the tree's root $K(r)$. If they match, a desired element is found and the search can be stopped; if they do not match, we continue with the search in the left subtree of the root if $v < K(r)$ and in the right subtree if $v > K(r)$. Thus, on each iteration of the algorithm, the problem of searching in a binary search tree is reduced to searching in a smaller binary search tree. The most sensible measure of size of a search tree is its height; obviously, the decrease in a tree's height normally changes from one iteration to another of the binary tree search—thus giving us an excellent example of a variable-size-decrease algorithm.

In the worst case of the binary tree search, the tree is severely skewed. This happens, in particular, if a tree is constructed by successive insertions of an increasing or decreasing sequence of keys (Figure 5.17).

Obviously, the search for a_{n-1} in such a tree requires n comparisons, making the worst-case efficiency of the search operation fall into $\Theta(n)$. Fortunately, the average-case efficiency turns out to be in $\Theta(\log n)$. More precisely, the number of key comparisons needed for a search in a binary search tree built from n random keys is about $2\ln n \approx 1.39 \log_2 n$. Since the operation of insertion of a new key into a binary search tree is almost identical to that of searching there, it also exemplifies the variable-size-decrease technique and has the same efficiency characteristics as the search operation.

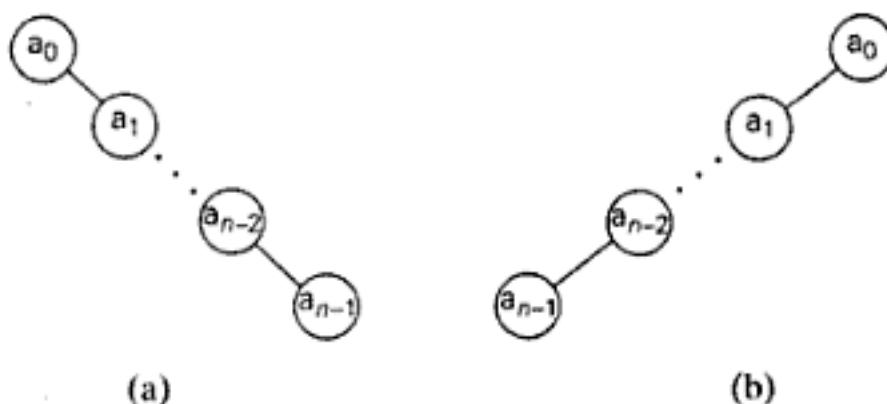


FIGURE 5.17 Binary search trees for (a) an increasing sequence of keys and (b) a decreasing sequence of keys

The Game of Nim

There are several well-known games that share the following features. There are two players, who move in turn. No randomness or hidden information is permitted: all players know all information about gameplay. A game is impartial: each player has the same moves available from the same game position. Each of a finite number of available moves leads to a smaller instance of the same game. The game ends with a win by one of the players (there are no ties). The winner is the last player who is able to move.

A prototypical example of such games is *Nim*. Generally, the game is played with several piles of chips, but we consider the one-pile version first. Thus, there is a single pile of n chips. Two players take turns by removing from the pile at least one and at most m chips; the number of chips taken may vary from one move to another, but both the lower and upper limits stay the same. Who wins the game by taking the last chip, the player moving first or second, if both players make the best moves possible?

Let us call an instance of the game a winning position for the player to move next if that player has a winning strategy, that is, a sequence of moves that results in a victory no matter what moves the opponent makes. Let us call an instance of the game a losing position for the player to move next if every move available for that player leads to a winning position for the opponent. The standard approach to determining which positions are winning and which are losing is to investigate small values of n first. It is logical to consider the instance of $n = 0$ as a losing one for the player to move next because this player is the first one who cannot make a move. Any instance with $1 \leq n \leq m$ chips is obviously a winning position for the player to move next (why?). The instance with $n = m + 1$ chips is a losing one because taking any allowed number of chips puts the opponent in a winning position. (See an illustration for $m = 4$ in Figure 5.18.) Any instance with $m + 2 \leq n \leq 2m + 1$ chips is a winning position for the player to move next because there is a move that leaves the opponent with $m + 1$ chips, which is a losing position; $2m + 2 = 2(m + 1)$ chips is the next losing position, and so on. It is not difficult to see the pattern that can be formally proved by mathematical induction: an instance with n chips is a winning position for the player to move next if and only if n is not a multiple of $m + 1$. The winning strategy is to take

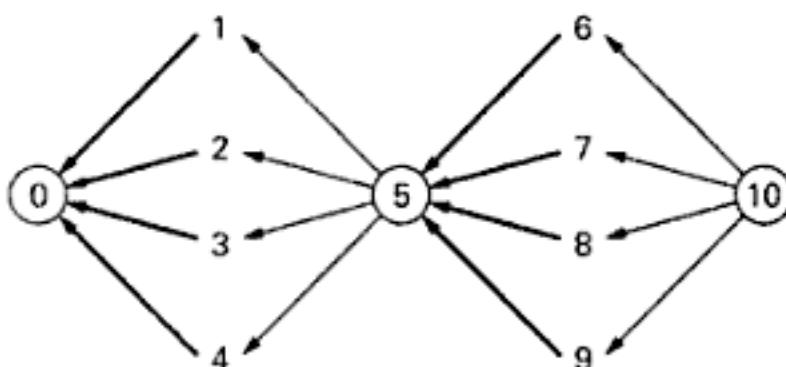


FIGURE 5.18 Figure 5.18 illustrates one-pile Nim with the maximum number of chips that may be taken on each move $m = 4$. The numbers indicate n , the number of chips in the pile. The losing positions for the player to move are circled. Only winning moves from the winning positions are shown (in bold).

$n \bmod (m + 1)$ chips on every move; any deviation from this strategy puts the opponent in a winning position.

One-pile Nim has been known for a very long time. It appeared, in particular, as the *summation game* in the first published book on recreational mathematics, authored by Claude-Gaspar Bachet, a French aristocrat and mathematician, in 1612: a player picks a positive integer less than, say, ten, and then his opponent and he take turns adding any integer less than ten; the first player to reach 100 exactly is the winner [Dud70].

In general, Nim is played with $I > 1$ piles of chips of sizes n_1, n_2, \dots, n_I . On each move, a player can take any available number of chips, including all of them, from any single pile. The goal is the same—to be the last player able to make a move. Note that for $I = 2$, it is easy to figure out who wins this game and how. Here is a hint: the answer for the game's instances with $n_1 = n_2$ differs from the answer for those with $n_1 \neq n_2$.

A solution to the general case of Nim is quite unexpected because it is based on the binary representation of the pile sizes.

Let b_1, b_2, \dots, b_I be the pile sizes in binary. Compute their *binary digital sum*, defined as the sum of binary digits discarding any carry. (In other words, a binary digit s_i in the sum is 0 if the number of 1's in the i th position in the addends is even, and it is 1 if the number of 1's is odd.) It turns out that an instance of Nim is a winning one (for the player to move next) if and only if its binary digital sum contains at least one 1; consequently, Nim's instance is a loosing instance if and only if its binary digit sum contains only zeros. For example, for the commonly played instance with $n_1 = 3, n_2 = 4, n_3 = 5$, the binary digital sum is

$$\begin{array}{r} 011 \\ 100 \\ 101 \\ \hline 010 \end{array}$$

Since this sum contains a 1, the instance is a winning one for the payer moving first. To find a winning move from this position, the player needs to change one of

the three bit strings so that the new binary digital sum contains only 0's. It is not difficult to see that the only way to accomplish this is to remove two chips from the first pile.

This ingenious solution to the game of Nim was discovered by Harvard mathematics professor C. L. Bouton more than 100 years ago. Since then, mathematicians have developed a much more general theory of such games. An excellent account of this theory, with applications to many specific games, is given in the monograph by E. R. Berlekamp, J. H. Conway, and R. K. Guy [Ber03].

Exercises 5.6

1. In the problem of computing the gcd of m and n , if we measure the size of an instance by the size of the first parameter m , by how much can the size decrease after one iteration of Euclid's algorithm?
2.
 - a. Apply the partition-based algorithm to find the median of the list of numbers 9, 12, 5, 17, 20.
 - b. Show that the worst-case efficiency of the partition-based algorithm for the selection problem is quadratic.
3.
 - a. Write a pseudocode for a nonrecursive implementation of the partition-based algorithm for the selection problem.
 - b. Write a pseudocode for a recursive implementation of this algorithm.
4. Derive the formula underlying interpolation search.
5. Describe briefly the interpolation search, and explain with an example the performance of the algorithm in the worst-case scenario.
6.
 - a. Find the smallest value of n for which $\log_2 \log_2 n + 1$ is greater than 6.
 - b. Determine which, if any, of the following assertions are true:
 - i. $\log \log n \in o(\log n)$
 - ii. $\log \log n \in \Theta(\log n)$
 - iii. $\log \log n \in \Omega(\log n)$
7.
 - a. Outline an algorithm for finding the smallest key in a binary search tree. Would you classify your algorithm as a variable-size decrease algorithm?
 - b. What is the time efficiency class of your algorithm in the worst-case?
8.
 - a. Outline an algorithm for deleting a key from a binary search tree. Would you classify this algorithm as a variable-size decrease algorithm?
 - b. What is the time efficiency class of your algorithm in the worst case?
9. **Misere one-pile Nim** Consider the so-called **misere version** of the one-pile Nim, in which the player taking the last chip loses the game. All the other conditions of the game remain the same, i.e., the pile contains n chips and on each move a player takes at least one but no more than m chips. Identify the winning and loosing positions (for the player to move next) in this game.
10.
 - a. **Moldy chocolate** Two players take turns by breaking an m -by- n chocolate bar, which has one spoiled 1-by-1 square. Each break must be a single



straight line cutting all the way across the bar along the boundaries between the squares. After each break, the player who broke the bar last eats the piece that does not contain the spoiled square. The player left with the spoiled square loses the game. Is it better to go first or second in this game?

- b. Write an interactive program to play this game with the computer. Your program should make a winning move in a winning position and a random legitimate move in a losing position.



- 11. Flipping pancakes** There are n pancakes all of different sizes that are stacked on top of each other. You are allowed to slip a flipper under one of the pancakes and flip over the whole sack above the flipper. The purpose is to arrange pancakes according to their size with the biggest at the bottom. (You can see a visualization of this puzzle on the *Interactive Mathematics Miscellany and Puzzles* site [Bog].) Design an algorithm for solving this puzzle.

SUMMARY

- *Decrease-and-conquer* is a general algorithm design technique, based on exploiting a relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. Once such a relationship is established, it can be exploited either top down (recursively) or bottom up (without a recursion).
- There are three major variations of decrease-and-conquer:
 - *decrease-by-a-constant*, most often by one (e.g., insertion sort);
 - *decrease-by-a-constant-factor*, most often by the factor of two (e.g., binary search);
 - *variable-size-decrease* (e.g., Euclid's algorithm).
- *Insertion sort* is a direct application of the decrease (by one)-and-conquer technique to the sorting problem. It is a $\Theta(n^2)$ algorithm both in the worst and average cases, but it is about twice as fast on average than in the worst case. The algorithm's notable advantage is a good performance on almost-sorted arrays.
- *Depth-first search (DFS)* and *breadth-first search (BFS)* are two principal graph traversal algorithms. By representing a graph in a form of a depth-first or breadth-first search forest, they help in the investigation of many important properties of the graph. Both algorithms have the same time efficiency: $\Theta(|V|^2)$ for the adjacency matrix representation and $\Theta(|V| + |E|)$ for the adjacency list representation.
- A *digraph* is a graph with directions on its edges. The *topological sorting problem* asks to list vertices of a digraph in an order such that for every edge of the digraph, the vertex it starts at is listed before the vertex it points to. This problem has a solution if and only if a digraph is a *dag (directed acyclic graph)*, i.e., it has no directed cycles.

- There are two algorithms for solving the topological sorting problem. The first one is based on depth-first search; the second is based on the direct implementation of the decrease-by-one technique.
- Decrease-by-one technique is a natural approach to developing algorithms for generating elementary combinatorial objects. The most efficient type of such algorithms are minimal-change algorithms. However, the number of combinatorial objects grows so fast that even the best algorithms are of practical interest only for very small instances of such problems.
- Identifying a fake coin with a balance scale, *multiplication à la russe*, and the *Josephus problem* are examples of problems that can be solved by *decrease-by-a-constant-factor* algorithms. Two other and more important examples are binary search and exponentiation by squaring.
- For some algorithms based on the decrease-and-conquer technique, the size reduction varies from one iteration of the algorithm to another. Examples of such *variable-size-decrease* algorithms include Euclid's algorithm, the partition-based algorithm for the *selection problem*, *interpolation search*, and searching and insertion in a binary search tree. *Nim* exemplifies games that proceed through a series of diminishing instances of the same game.

—

—

6

Transform-and-Conquer

That's the secret to life . . . replace one worry with another.

—Charles M. Schulz (1922–2000), American cartoonist, the creator of *Peanuts*

This chapter deals with a group of design methods that are based on the idea of transformation. We call this general technique ***transform-and-conquer*** because these methods work as two-stage procedures. First, in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution. Then, in the second or conquering stage, it is solved.

There are three major variations of this idea that differ by what we transform a given instance to (Figure 6.1):

- transformation to a simpler or more convenient instance of the same problem—we call it ***instance simplification***
- transformation to a different representation of the same instance—we call it ***representation change***
- transformation to an instance of a different problem for which an algorithm is already available—we call it ***problem reduction***

In the first three sections of this chapter, we will encounter examples of the instance simplification variety. Section 6.1 deals with the simple but fruitful idea of presorting. Many questions about lists are easier to answer

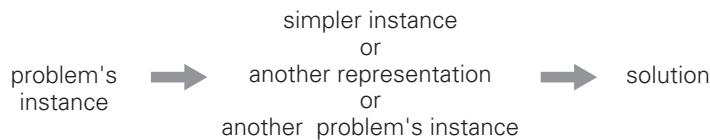


FIGURE 6.1 Transform-and-conquer strategy

if the lists are sorted. Of course, the benefits of a sorted list should more than compensate for the time spent on sorting; otherwise, we would be better off dealing with an unsorted list directly. Section 6.2 introduces one of the most important algorithms in applied mathematics: Gaussian elimination. This algorithm solves a system of linear equations by first transforming it to another system with a special property that makes finding a solution quite easy. In Section 6.3, the ideas of instance simplification and representation change are applied to search trees. The results are AVL trees and multiway balanced search trees; of the latter we consider the simplest case, 2–3 trees.

Section 6.4 presents heaps and heapsort. Even if you are already familiar with this important data structure and its application to sorting, you can still benefit from looking at them in this new light of transform-and-conquer design. In Section 6.5, we discuss Horner’s rule, a remarkable algorithm for evaluating polynomials. If there were an Algorithm Hall of Fame, Horner’s rule would be a serious candidate for induction based on the algorithm’s elegance and efficiency. We also consider there two algorithms for the exponentiation problem, both based on the representation change idea.

The chapter concludes with a review of several applications of the third variety of transform-and-conquer: problem reduction. This variety should be considered the most radical of the three: one problem is reduced to another, i.e., transformed into an entirely different problem. This is a very powerful idea, and it is extensively used in the complexity theory (Chapter 10). Its application to designing practical algorithms is not trivial, however. First, we need to identify a new problem into which the given problem should be transformed. Then we must make sure that the transformation algorithm followed by the algorithm for solving the new problem is time efficient compared to other algorithmic alternatives. Among several examples, we discuss an important special case of **mathematical modeling**, or expressing a problem in terms of purely mathematical objects such as variables, functions, and equations.

6.1 Presorting

Presorting is an old idea in computer science. In fact, interest in sorting algorithms is due, to a significant degree, to the fact that many questions about a list are easier to answer if the list is sorted. Obviously, the time efficiency of algorithms that involve sorting may depend on the efficiency of the sorting algorithm being used. For the sake of simplicity, we assume throughout this section that lists are implemented as arrays, because some sorting algorithms are easier to implement for the array representation.

So far, we have discussed three elementary sorting algorithms—selection sort, bubble sort, and insertion sort—that are quadratic in the worst and average cases and two advanced algorithms—mergesort, which is always in $\Theta(n \log n)$, and quicksort, whose efficiency is also $\Theta(n \log n)$ in the average case but is quadratic in the worst case. Are there faster sorting algorithms? As we have already stated in Section 1.3 (see also Section 10.2), no general comparison-based sorting algorithm can have a better efficiency than $n \log n$ in the worst case and the same result holds for average-case efficiency.¹

Following are three examples that illustrate the idea of presorting. More examples can be found in the exercises to this section.

EXAMPLE 1 *Checking element uniqueness in an array*

If this element uniqueness problem looks familiar to you, it should; we considered a brute-force algorithm for the problem in Section 2.3 (see Example 2). The brute-force algorithm compared pairs of the array’s elements until either two equal elements were found or no more pairs were left. Its worst-case efficiency was in $\Theta(n^2)$.

Alternatively, we can sort the array first and then check only its consecutive elements: if the array has equal elements, a pair of them must be next to each other and vice versa.

ALGORITHM *PresortElementUniqueness(A[0..n – 1])*

```
//Solves the element uniqueness problem by sorting the array first
//Input: An array A[0..n – 1] or orderable elements
//Output: Returns “true” if A has no equal elements, “false” otherwise
Sort the array A
for i ← 0 to n – 2 do
    if A[i] = A[i + 1] return false
return true
```

The running time of this algorithm is the sum of the time spent on sorting and the time spent on checking consecutive elements. Since the former requires at least $n \log n$ comparisons and the latter needs no more than $n – 1$ comparisons, it is the sorting part that will determine the overall efficiency of the algorithm. So, if we use a quadratic sorting algorithm here, the entire algorithm will not be more efficient than the brute-force one. But if we use a good sorting algorithm, such as mergesort, with worst-case efficiency in $\Theta(n \log n)$, the worst-case efficiency of the entire presorting-based algorithm will be also in $\Theta(n \log n)$:

$$T(n) = T_{\text{sort}}(n) + T_{\text{scan}}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n).$$

-
1. Sorting algorithms called **radix sorts** are linear but in terms of the total number of input bits. These algorithms work by comparing individual bits or pieces of keys rather than keys in their entirety. Although the running time of these algorithms is proportional to the number of input bits, they are still essentially $n \log n$ algorithms because the number of bits per key must be at least $\log_2 n$ in order to accommodate n distinct keys of input.

EXAMPLE 2 Computing a mode

A **mode** is a value that occurs most often in a given list of numbers. For example, for 5, 1, 5, 7, 6, 5, 7, the mode is 5. (If several different values occur most often, any of them can be considered a mode.) The brute-force approach to computing a mode would scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency. In order to implement this idea, we can store the values already encountered, along with their frequencies, in a separate list. On each iteration, the i th element of the original list is compared with the values already encountered by traversing this auxiliary list. If a matching value is found, its frequency is incremented; otherwise, the current element is added to the list of distinct values seen so far with the frequency of 1.

It is not difficult to see that the worst-case input for this algorithm is a list with no equal elements. For such a list, its i th element is compared with $i - 1$ elements of the auxiliary list of distinct values seen so far before being added to the list with a frequency of 1. As a result, the worst-case number of comparisons made by this algorithm in creating the frequency list is

$$C(n) = \sum_{i=1}^n (i - 1) = 0 + 1 + \dots + (n - 1) = \frac{(n - 1)n}{2} \in \Theta(n^2).$$

The additional $n - 1$ comparisons needed to find the largest frequency in the auxiliary list do not change the quadratic worst-case efficiency class of the algorithm.

As an alternative, let us first sort the input. Then all equal values will be adjacent to each other. To compute the mode, all we need to do is to find the longest run of adjacent equal values in the sorted array.

ALGORITHM PresortMode($A[0..n - 1]$)

```

//Computes the mode of an array by sorting it first
//Input: An array A[0..n - 1] of orderable elements
//Output: The array's mode
Sort the array A
i ← 0           //current run begins at position i
modefrequency ← 0 //highest frequency seen so far
while i ≤ n - 1 do
    runlength ← 1; runvalue ← A[i]
    while i + runlength ≤ n - 1 and A[i + runlength] = runvalue
        runlength ← runlength + 1
    if runlength > modefrequency
        modefrequency ← runlength; modevalue ← runvalue
    i ← i + runlength
return modevalue

```

The analysis here is similar to the analysis of Example 1: the running time of the algorithm will be dominated by the time spent on sorting since the remainder of the algorithm takes linear time (why?). Consequently, with an $n \log n$ sort, this method's worst-case efficiency will be in a better asymptotic class than the worst case efficiency of the brute-force algorithm. ■

EXAMPLE 3 *Searching problem*

Consider the problem of searching for a given value v in a given array of n sortable items. The brute-force solution here is sequential search (Section 3.1) that needs n comparisons in the worst case. If the array is sorted first, we can then apply binary search that requires only $\lfloor \log_2 n \rfloor + 1$ comparisons in the worst case. Assuming the most efficient $n \log n$ sort, the total running time of such a searching algorithm in the worst case will be

$$T(n) = T_{\text{sort}}(n) + T_{\text{search}}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n),$$

which is inferior to sequential search. The same will also be true for the average-case efficiency. Of course, if we are to search in the same list more than once, the time spent on sorting might well be justified. (Problem 6 in the exercises asks to estimate the smallest number of searches needed to justify presorting.) ■

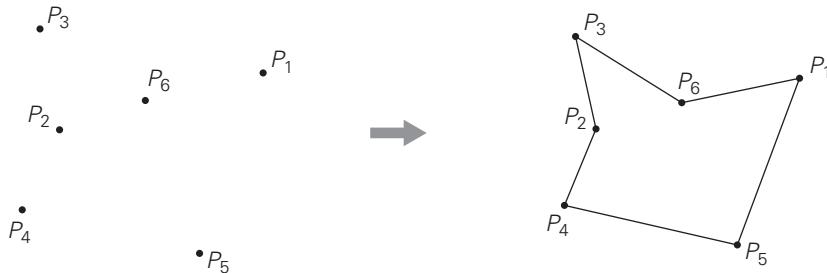
Exercises 6.1

Before we finish our discussion of presorting, we should mention that many, if not most, geometric algorithms dealing with sets of points use presorting in one way or another. Points can be sorted by one of their coordinates or by their distance from a particular line or by some angle, and so on. For example, presorting was used in the divide-and-conquer algorithms for the closest-pair problem and for the convex-hull problem, which were discussed in Section 4.6.

Exercises 6.1

1. Recall that the **median** of a list of n numbers is defined as its $[n/2]$ smallest element. (The median is larger than one half the elements and is smaller than the other half.) Design a presorting-based algorithm for finding the median and determine its efficiency class.
2. Consider the problem of finding the distance between the two closest numbers in an array of n numbers. (The distance between two numbers x and y is computed as $|x - y|$.)
 - a. Design a presorting-based algorithm for solving this problem and determine its efficiency class.

- b.** Compare the efficiency of this algorithm with that of the brute-force algorithm (see Problem 9 in Exercises 1.2).
- 3.** Let $A = \{a_1, \dots, a_n\}$ and $B = \{b_1, \dots, b_m\}$ be two sets of numbers. Consider the problem of finding their intersection, i.e., the set C of all the numbers that are in both A and B .
 - a.** Design a brute-force algorithm for solving this problem and determine its efficiency class.
 - b.** Design a presorting-based algorithm for solving this problem and determine its efficiency class.
- 4.** Consider the problem of finding the smallest and largest elements in an array of n numbers.
 - a.** Design a presorting-based algorithm for solving this problem and determine its efficiency class.
 - b.** Compare the efficiency of the three algorithms: (i) the brute-force algorithm, (ii) this presorting-based algorithm, and (iii) the divide-and-conquer algorithm (see Problem 2 in Exercises 4.1).
- 5.** Show that the average-case efficiency of one-time searching by the algorithm that consists of the most efficient comparison-based sorting algorithm followed by binary search is inferior to the average-case efficiency of sequential search.
- 6.** Estimate how many searches will be needed to justify time spent on presorting an array of 10^3 elements if sorting is done by mergesort and searching is done by binary search. (You may assume that all searches are for elements known to be in the array.) What about an array of 10^6 elements?
- 7.** To sort or not to sort? Design a reasonably efficient algorithm for solving each of the following problems and determine its efficiency class.
 - a.** You are given n telephone bills and m checks sent to pay the bills ($n \geq m$). Assuming that telephone numbers are written on the checks, find out who failed to pay. (For simplicity, you may also assume that only one check is written for a particular bill and that it covers the bill in full.)
 - b.** You have a file of n student records indicating each student's number, name, home address, and date of birth. Find out the number of students from each of the 50 U.S. states.
- 8.** Given a set of $n \geq 3$ points in the x - y coordinate plane, connect them in a simple polygon, i.e., a closed path through all the points so that its line segments (the polygon's edges) do not intersect (except for neighboring edges at their common vertex). For example,



- a. Does the problem always have a solution? Does it always have a unique solution?
 - b. Design a reasonably efficient algorithm for solving this problem and indicate its efficiency class.
9. You have an array of n numbers and an integer s . Find out whether the array contains two elements whose sum is s . (For example, for the array 5, 9, 1, 3 and $s = 6$, the answer is yes, but for the same array and $s = 7$, the answer is no.) Design an algorithm for this problem with a better than quadratic time efficiency.
10. a. Design an efficient algorithm for finding all sets of anagrams in a large file such as a dictionary of English words [Ben00]. For example, *eat*, *ate*, and *tea* belong to one such a set.
 b. Write a program implementing the algorithm.

Exercises 6.1


6.2 Gaussian Elimination

You are certainly familiar with systems of two linear equations in two unknowns:

$$\begin{aligned} a_{11}x + a_{12}y &= b_1 \\ a_{21}x + a_{22}y &= b_2 \end{aligned}$$

Recall that unless the coefficients of one equation are proportional to the coefficients of the other, the system has a unique solution. The standard method for finding this solution is to use either equation to express one of the variables as a function of the other and then substitute the result into the other equation, yielding a linear equation whose solution is then used for finding the value of the second variable.

In many applications, we need to solve a system of n equations in n unknowns—

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

—where n is a large number. Theoretically we can solve such a system by generalizing the substitution method for solving systems of two linear equations (what general design technique would such a method be based upon?); however, the resulting algorithm would be extremely cumbersome.

Fortunately, there is a much more elegant algorithm for solving systems of linear equations called **Gaussian elimination**.² The idea of Gaussian elimination is to transform a system of n linear equations with n unknowns to an equivalent system (i.e., a system with the same solution as the original one) with an upper-triangular coefficient matrix, a matrix with all zeros below its main diagonal:

$$\begin{array}{ll} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 & a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n = b'_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 & a'_{22}x_2 + \cdots + a'_{2n}x_n = b'_2 \\ \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n & a'_{nn}x_n = b'_n \end{array} \implies$$

In matrix notations, we can write it as

$$Ax = b \implies A'x = b',$$

where

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, A' = \begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} \\ 0 & a'_{22} & \cdots & a'_{2n} \\ \vdots & & & \\ 0 & 0 & \cdots & a'_{nn} \end{bmatrix}, b = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}.$$

(We added primes to the matrix elements and right-hand sides of the new system to stress the point that their values differ from their counterparts in the original system.)

Why is the system with the upper-triangular coefficient matrix better than a system with an arbitrary coefficient matrix? Because we can easily solve the system with an upper-triangular coefficient matrix by backward substitutions as follows. First, we can immediately find the value of x_n from the last equation; then we can substitute this value into the next to last equation to get x_{n-1} and so on until we substitute the known values of the last $n - 1$ variables into the first equation, from which we find the value of x_1 .

2. The method is named after Carl Friedrich Gauss (1777–1855), who—like other giants in the history of mathematics such as Isaac Newton and Leonhard Euler—made numerous fundamental contributions to both theoretical and computational mathematics.

So how can we get from a system with an arbitrary coefficient matrix A to an equivalent system with an upper-triangular coefficient matrix A' ? We can do it through a series of the so-called elementary operations:

- Exchanging two equations of the system
- Replacing an equation with its nonzero multiple
- Replacing an equation with a sum or difference of this equation and some multiple of another equation

Since no elementary operation can change a solution to a system, any system that is obtained through a series of such operations will have the same solution as the original one. Let us see how we can get to a system with an upper-triangular matrix. First, we use a_{11} as a pivot to make all x_1 coefficients zeros in the equations below the first one. Specifically, we replace the second equation with the difference between it and the first equation multiplied by a_{21}/a_{11} to get an equation with a zero coefficient for x_1 . Doing the same for the third, fourth, and finally n th equation—with the multiples $a_{31}/a_{11}, a_{41}/a_{11}, \dots, a_{n1}/a_{11}$ of the first equation, respectively—makes all the coefficients of x_1 below the first equation zero. Then we get rid of all the coefficients of x_2 by subtracting an appropriate multiple of the second equation from each of the equations below the second one. Repeating this elimination for each of the first $n - 1$ variables ultimately yields a system with an upper-triangular coefficient matrix.

Before we look at an example of Gaussian elimination, let us note that we can operate with just a system's coefficient matrix augmented, as its $(n + 1)$ st column, with the equations' right-hand side values. In other words, we need to write explicitly neither the variable names nor the plus and equality signs.

EXAMPLE Solve the system by Gaussian elimination

$$2x_1 - x_2 + x_3 = 1$$

$$4x_1 + x_2 - x_3 = 5$$

$$x_1 + x_2 + x_3 = 0$$

$$\left[\begin{array}{cccc} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{array} \right] \text{row } 2 - \frac{4}{2} \text{ row } 1$$

$$\left[\begin{array}{cccc} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{array} \right] \text{row } 3 - \frac{1}{2} \text{ row } 2$$

$$\left[\begin{array}{cccc} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{array} \right]$$

Now we can obtain the solution by backward substitutions:

$$x_3 = (-2)/2 = -1, x_2 = (3 - (-3)x_3)/3 = 0, \text{ and } x_1 = (1 - x_3 - (-1)x_2)/2 = 1. \quad \blacksquare$$

Here is a pseudocode for the elimination stage of Gaussian elimination.

```
ALGORITHM GaussElimination(A[1..n, 1..n], b[1..n])
    //Applies Gaussian elimination to matrix A of a system's coefficients,
    //augmented with vector b of the system's right-hand side values.
    //Input: Matrix A[1..n, 1..n] and column-vector b[1..n]
    //Output: An equivalent upper-triangular matrix in place of A with the
    //corresponding right-hand side values in the (n + 1)st column.
    for  $i \leftarrow 1$  to  $n$  do  $A[i, n + 1] \leftarrow b[i]$  //augments the matrix
    for  $i \leftarrow 1$  to  $n - 1$  do
        for  $j \leftarrow i + 1$  to  $n$  do
            for  $k \leftarrow i$  to  $n + 1$  do
                 $A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] / A[i, i]$ 
```

There are two important observations to make about this pseudocode. First, it is not always correct: if $A[i, i] = 0$, we cannot divide by it and hence cannot use the i th row as a pivot for the i th iteration of the algorithm. In such a case, we should take advantage of the first elementary operation and exchange the i th row with some row below it that has a nonzero coefficient in the i th column. (If the system has a unique solution, which is the normal case for systems under consideration, such a row must exist.)

Since we have to be prepared for the possibility of row exchanges anyway, we can take care of another potential difficulty: the possibility that $A[i, i]$ is so small and consequently the scaling factor $A[j, i]/A[i, i]$ so large that the new value of $A[j, k]$ might become distorted by a round-off error caused by a subtraction of two numbers of greatly different magnitudes.³ To avoid this problem, we can always look for a row with the largest absolute value of the coefficient in the i th column, exchange it with the i th row, and then use it as the i th iteration's pivot. This modification, called **partial pivoting**, guarantees that the magnitude of the scaling factor will never exceed 1.

The second observation is the fact that the innermost loop is written with a glaring inefficiency. Can you find it before checking the following pseudocode, which both incorporates partial pivoting and eliminates this inefficiency?

ALGORITHM *BetterGaussElimination(A[1..n, 1..n], b[1..n])*

```
//Implements Gaussian elimination with partial pivoting
//Input: Matrix A[1..n, 1..n] and column-vector b[1..n]
```

3. We discuss this phenomenon, called the **subtractive cancellation**, in more detail in Section 10.4.

```

//Output: An equivalent upper-triangular matrix in place of A and the
//corresponding right-hand side values in place of the (n + 1)st column.
for  $i \leftarrow 1$  to  $n$  do  $A[i, n + 1] \leftarrow b[i]$  //appends  $b$  to  $A$  as the last column
for  $i \leftarrow 1$  to  $n - 1$  do
     $pivotrow \leftarrow i$ 
    for  $j \leftarrow i + 1$  to  $n$  do
        if  $|A[j, i]| > |A[pivotrow, i]|$   $pivotrow \leftarrow j$ 
        for  $k \leftarrow i$  to  $n + 1$  do
             $swap(A[i, k], A[pivotrow, k])$ 
        for  $j \leftarrow i + 1$  to  $n$  do
             $temp \leftarrow A[j, i] / A[i, i]$ 
            for  $k \leftarrow i$  to  $n + 1$  do
                 $A[j, k] \leftarrow A[j, k] - A[i, k] * temp$ 

```

Let us find the time efficiency of this algorithm. Its innermost loop consists of a single line

$$A[j, k] \leftarrow A[j, k] - A[i, k] * temp$$

which contains one multiplication and one subtraction. On most computers, multiplication is unquestionably more expensive than addition/subtraction, and hence it is the multiplication count that is usually quoted as the algorithm's basic operation.⁴ The standard summation formulas and rules reviewed in Section 2.3 (see also Appendix A) are very helpful in the following derivation:

$$\begin{aligned}
C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^n (n+1-i+1) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n (n+2-i) \\
&= \sum_{i=1}^{n-1} (n+2-i)(n-(i+1)+1) = \sum_{i=1}^{n-1} (n+2-i)(n-i) \\
&= (n+1)(n-1) + n(n-2) + \dots + 3 \cdot 1 \\
&= \sum_{j=1}^{n-1} (j+2)j = \sum_{j=1}^{n-1} j^2 + \sum_{j=1}^{n-1} 2j = \frac{(n-1)n(2n-1)}{6} + 2 \frac{(n-1)n}{2} \\
&= \frac{n(n-1)(2n+5)}{6} \approx \frac{1}{3}n^3 \in \Theta(n^3).
\end{aligned}$$

4. As we mentioned in Section 2.1, on some computers, multiplication is not necessarily more expensive than addition/subtraction. For this algorithm, this point is moot since we can simply count the number of times the innermost loop is executed, which is, of course, exactly the same number as the number of multiplications and the number of subtractions there.

Since the second (backward substitution) stage of the Gaussian elimination is in $\Theta(n^2)$, as we will ask you to show in the exercises, the running time is dominated by the cubic elimination stage, making the entire algorithm cubic as well.

Theoretically, Gaussian elimination always either yields an exact solution to a system of linear equations when the system has a unique solution or discovers that no such solution exists. In the latter case, the system will have either no solutions or infinitely many of them. In practice, solving systems of significant size on a computer by this method is not nearly so straightforward as the method would lead us to believe. The principal difficulty lies in preventing an accumulation of round-off errors (see Section 10.4). Consult textbooks on numerical analysis that analyze this and other implementation issues in great detail.

LU Decomposition and Other Applications

Gaussian elimination has an interesting and very useful by-product called ***LU decomposition*** of the coefficient matrix. In fact, modern commercial implementations of Gaussian elimination are based on such a decomposition rather than on the basic algorithm outlined earlier.

EXAMPLE Let us return to the example at the beginning of this section, where we applied Gaussian elimination to the matrix

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 4 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Consider the lower-triangular matrix L made up of 1's on its main diagonal and the row multiples used in the Gaussian elimination process

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix}$$

and the upper-triangular matrix U that was the result of this elimination

$$U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{bmatrix}.$$

It turns out that the product LU of these matrices is equal to matrix A . (For this particular pair of L and U , you can verify this fact by direct multiplication, but as a general proposition, it needs, of course, a proof, which we will omit.)

Therefore, solving the system $Ax = b$ is equivalent to solving the system $LUX = b$. The latter system can be solved as follows. Denote $y = UX$, then $Ly = b$. Solve the system $Ly = b$ first, which is easy to do because L is a lower-triangular matrix; then solve the system $UX = y$, with the upper-triangular matrix

U , to find x . Thus, for the system at the beginning of this section, we first solve $Ly = b$:

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \\ 0 \end{bmatrix}.$$

Its solution is

$$y_1 = 1, \quad y_2 = 5 - 2y_1 = 3, \quad y_3 = 0 - \frac{1}{2}y_1 - \frac{1}{2}y_2 = -2.$$

Solving $Ux = y$ means solving

$$\begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ -2 \end{bmatrix},$$

and the solution is

$$x_3 = (-2)/2 = -1, \quad x_2 = (3 - (-3)x_3)/3 = 0, \quad x_1 = (1 - x_3 - (-1)x_2)/2 = 1. \quad \blacksquare$$

Note that once we have the LU decomposition of matrix A , we can solve systems $Ax = b$ with as many right-hand side vectors b as we want to, one at a time. This is a distinct advantage over the classic Gaussian elimination discussed earlier. Also note that the LU decomposition does not actually require extra memory because we can store the nonzero part of U in the upper-triangular part of A (including the main diagonal) and store the nontrivial part of L below the main diagonal of A .

Computing a Matrix Inverse

Gaussian elimination is a very useful algorithm that tackles one of the most important problems of applied mathematics: solving systems of linear equations. In fact, Gaussian elimination can also be applied to several other problems of linear algebra, such as computing a matrix **inverse**. The inverse of an n -by- n matrix A is an n -by- n matrix, denoted A^{-1} , such that

$$AA^{-1} = I,$$

where I is the n -by- n identity matrix (the matrix with all zero elements except the main diagonal elements, which are all ones). Not every square matrix has an inverse, but when it exists, the inverse is unique. If a matrix A does not have an inverse, it is called **singular**. One can prove that a matrix is singular if and only if one of its rows is a linear combination (a sum of some multiples) of the other rows. A convenient way to check whether a matrix is nonsingular is to apply Gaussian elimination: if it yields an upper-triangular matrix with no zeros on the main diagonal, the matrix is nonsingular; otherwise, it is singular. So being singular is a very special situation, and most square matrices do have their inverses.

Theoretically, inverse matrices are very important because they play the role of reciprocals in matrix algebra, overcoming the absence of the explicit division operation for matrices. For example, in a complete analogy with a linear equation in one unknown $ax = b$ whose solution can be written as $x = a^{-1}b$ (if a is not zero), we can express a solution to a system of n equations in n unknowns $Ax = b$ as $x = A^{-1}b$ (if A is nonsingular) where b is, of course, a vector, not a number.

According to the definition of the inverse matrix for a nonsingular n -by- n matrix A , to find it, we need to find n^2 numbers x_{ij} , $1 \leq i, j \leq n$, such that

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & & & \\ x_{n1} & x_{n2} & \dots & x_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & & & \\ 0 & 0 & \dots & 1 \end{bmatrix}.$$

We can find the unknowns by solving n systems of linear equations that have the same coefficient matrix A , the vector of unknowns x^j is the j th column of the inverse, and the right-hand side vector e^j is the j th column of the identity matrix ($1 \leq j \leq n$):

$$Ax^j = e^j.$$

We can solve these systems by applying Gaussian elimination to matrix A augmented by the n -by- n identity matrix. Better yet, we can use Gaussian elimination to find the LU decomposition of A and then solve the systems $LUX^j = e^j$, $j = 1, \dots, n$, as explained earlier.

Computing a Determinant

Another problem that can be solved by Gaussian elimination is computing a determinant. The **determinant** of an n -by- n matrix A , denoted $\det A$ or $|A|$, is a number whose value can be defined recursively as follows. If $n = 1$, i.e., if A consists of a single element a_{11} , $\det A$ is equal to a_{11} ; for $n > 1$, $\det A$ is computed by the recursive formula

$$\det A = \sum_{j=1}^n s_j a_{1j} \det A_j,$$

where s_j is $+1$ if j is odd and -1 if j is even, a_{1j} is the element in row 1 and column j , and A_j is the $n - 1$ -by- $n - 1$ matrix obtained from matrix A by deleting its row 1 and column j .

In particular, for a 2-by-2 matrix, the definition implies a formula that is easy to remember:

$$\det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = a_{11} \det [a_{22}] - a_{12} \det [a_{21}] = a_{11}a_{22} - a_{12}a_{21}.$$

In other words, the determinant of a 2-by-2 matrix is simply equal to the difference between the products of its diagonal elements.

For a 3-by-3 matrix, we get

$$\begin{aligned} \det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \\ = a_{11} \det \begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} - a_{12} \det \begin{bmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{bmatrix} + a_{13} \det \begin{bmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \\ = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{21}a_{32}a_{13} - a_{31}a_{22}a_{13} - a_{21}a_{12}a_{33} - a_{32}a_{23}a_{11}. \end{aligned}$$

Incidentally, this formula is very handy in a variety of applications. In particular, we used it twice already in Section 4.6 as a part of the quickhull algorithm.

But what if we need to compute a determinant of a large matrix? (Although this is a task that is rarely needed in practice, it is worth discussing nevertheless.) Using the recursive definition can be of little help because it implies computing the sum of $n!$ terms. Here, Gaussian elimination comes to the rescue again. The central point is the fact that the determinant of an upper-triangular matrix is equal to the product of elements on its main diagonal, and it is easy to see how elementary operations employed by the algorithm influence the determinant's value. (Basically, it either remains unchanged or changes a sign or is multiplied by the constant used by the elimination algorithm.) As a result, we can compute the determinant of an n -by- n matrix in cubic time.

Determinants play an important role in the theory of systems of linear equations. Specifically, a system of n linear equations in n unknowns $Ax = b$ has a unique solution if and only if the determinant of its coefficient matrix, $\det A$, is not equal to zero. Moreover, this solution can be found by the formulas called **Cramer's rule**:

$$x_1 = \frac{\det A_1}{\det A}, \dots, x_j = \frac{\det A_j}{\det A}, \dots, x_n = \frac{\det A_n}{\det A},$$

where $\det A_j$ is the determinant of the matrix obtained by replacing the j th column of A by the column b . (We will ask you to investigate whether using Cramer's rule is a good algorithm for solving systems of linear equations in the exercises.)

Exercises 6.2

Exercises 6.2

1. Solve the following system by Gaussian elimination.

$$x_1 + x_2 + x_3 = 2$$

$$2x_1 + x_2 + x_3 = 3$$

$$x_1 - x_2 + 3x_3 = 8$$

2.
 - a. Solve the system of the previous question by the *LU* decomposition method.
 - b. From the standpoint of general algorithm design techniques, how would you classify the *LU* decomposition method?
3. Solve the system of Problem 1 by computing the inverse of its coefficient matrix and then multiplying it by the right-hand side vector.
4. Would it be correct to get the efficiency class of the elimination stage of Gaussian elimination as follows?

$$\begin{aligned}
 C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1} (n+2-i)(n-i) \\
 &= \sum_{i=1}^{n-1} [(n+2)n - i(2n+2) + i^2] \\
 &= \sum_{i=1}^{n-1} (n+2)n - \sum_{i=1}^{n-1} (2n+2)i + \sum_{i=1}^{n-1} i^2.
 \end{aligned}$$

Since $s_1(n) = \sum_{i=1}^{n-1} (n+2)n \in \Theta(n^3)$, $s_2(n) = \sum_{i=1}^{n-1} (2n+2)i \in \Theta(n^3)$, and $s_3(n) = \sum_{i=1}^{n-1} i^2 \in \Theta(n^3)$, $s_1(n) - s_2(n) + s_3(n) \in \Theta(n^3)$.

5. Write a pseudocode for the back-substitution stage of Gaussian elimination and show that its running time is in $\Theta(n^2)$.
6. Assuming that division of two numbers takes three times longer than their multiplication, estimate how much faster *BetterGaussElimination* is than *GaussElimination*. (Of course, you should also assume that a compiler is not going to eliminate the inefficiency in *GaussElimination*.)
7.
 - a. Give an example of a system of two linear equations in two unknowns that has a unique solution and solve it by Gaussian elimination.
 - b. Give an example of a system of two linear equations in two unknowns that has no solution and apply Gaussian elimination to it.
 - c. Give an example of a system of two linear equations in two unknowns that has infinitely many solutions and apply Gaussian elimination to it.
8. The ***Gauss-Jordan elimination*** method differs from Gaussian elimination in that the elements above the main diagonal of the coefficient matrix are made zero at the same time and by the same use of a pivot row as the elements below the main diagonal.
 - a. Apply the Gauss-Jordan method to the system of Problem 1 of these exercises.
 - b. What general design strategy is this algorithm based on?

- c. In general, how many multiplications are made by this method while solving a system of n equations in n unknowns? How does this compare with the number of multiplications made by the Gaussian elimination method in both its elimination and back-substitution stages?
 - 9. A system $Ax = b$ of n linear equations in n unknowns has a unique solution if and only if $\det A \neq 0$. Is it a good idea to check this condition before applying Gaussian elimination to a system?
 - 10. a. Apply Cramer's rule to solve the system of Problem 1 of these exercises.
 b. Estimate how many times longer it will take to solve a system of n linear equations in n unknowns by Cramer's rule than by Gaussian elimination. (Assume that all the determinants in Cramer's rule formulas are computed independently by Gaussian elimination.)
-

6.3 Balanced Search Trees

Exercises 6.2

In Sections 1.4 and 4.4, we discussed the binary search tree—one of the principal data structures for implementing dictionaries. It is a binary tree whose nodes contain elements of a set of orderable items, one element per node, so that all elements in the left subtree are smaller than the element in the subtree's root and all the elements in the right subtree are greater than it. Note that this transformation from a set to a binary search tree is an example of the representation change technique. What do we gain by such transformation compared to the straightforward implementation of a dictionary by, say, an array? We gain in the time efficiency of searching, insertion, and deletion, which are all in $\Theta(\log n)$, but only in the average case. In the worst case, these operations are in $\Theta(n)$ because the tree can degenerate into a severely unbalanced one with height equal to $n - 1$.

Computer scientists have expended a lot of effort in trying to find a structure that preserves the good properties of the classical binary search tree—principally, the logarithmic efficiency of the dictionary operations and having the set's elements sorted—but avoids its worst-case degeneracy. They have come up with two approaches.

- The first approach is of the instance simplification variety: an unbalanced binary search tree is transformed to a balanced one. Specific implementations of this idea differ by their definition of balance. An **AVL tree** requires the difference between the heights of the left and right subtrees of every node never exceed 1. A **red-black tree** tolerates the height of one subtree being twice as large as the other subtree of the same node. If an insertion or deletion of a new node creates a tree with a violated balance requirement, the tree is restructured by one of a family of special transformations called **rotations** that restore the balance required. (In this section, we discuss only AVL trees.

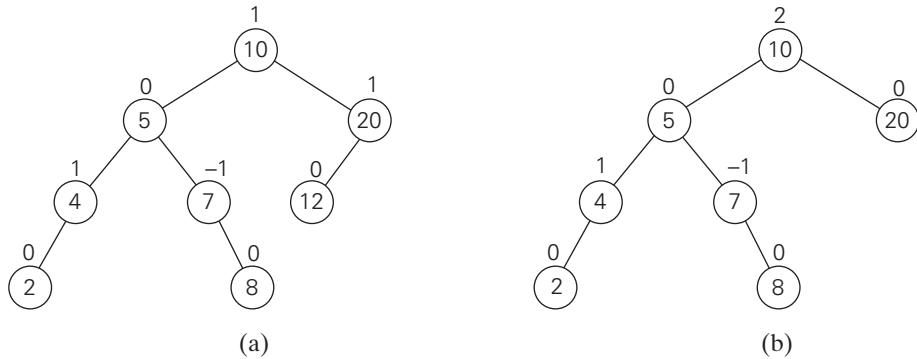


FIGURE 6.2 (a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

Information about other types of binary search trees that utilize the idea of rebalancing via rotations, including red-black trees and so-called *splay trees*, can be found in the references.)

- The second approach is of the representation change variety: allow more than one element in a node of a search tree. Specific cases of such trees are **2-3 trees**, **2-3-4 trees**, and more general and important **B-trees**. They differ in the number of elements admissible in a single node of a search tree, but all are perfectly balanced. (We discuss the simplest of such trees, the 2-3 tree, in this section, leaving the discussion of *B*-trees for Chapter 7.)

AVL Trees

AVL trees were invented in 1962 by two Russian scientists, G. M. Adelson-Velsky and E. M. Landis [AVL62], after whom this data structure is named.

DEFINITION An **AVL tree** is a binary search tree in which the **balance factor** of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or -1. (The height of the empty tree is defined as -1.)

For example, the binary search tree in Figure 6.2a is an AVL tree but the one in Figure 6.2b is not.

If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation. A **rotation** in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or -2; if there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf. There are only four types of rotations; in fact, two of them are mirror images of the other two. In their simplest form, the four rotations are shown in Figure 6.3.

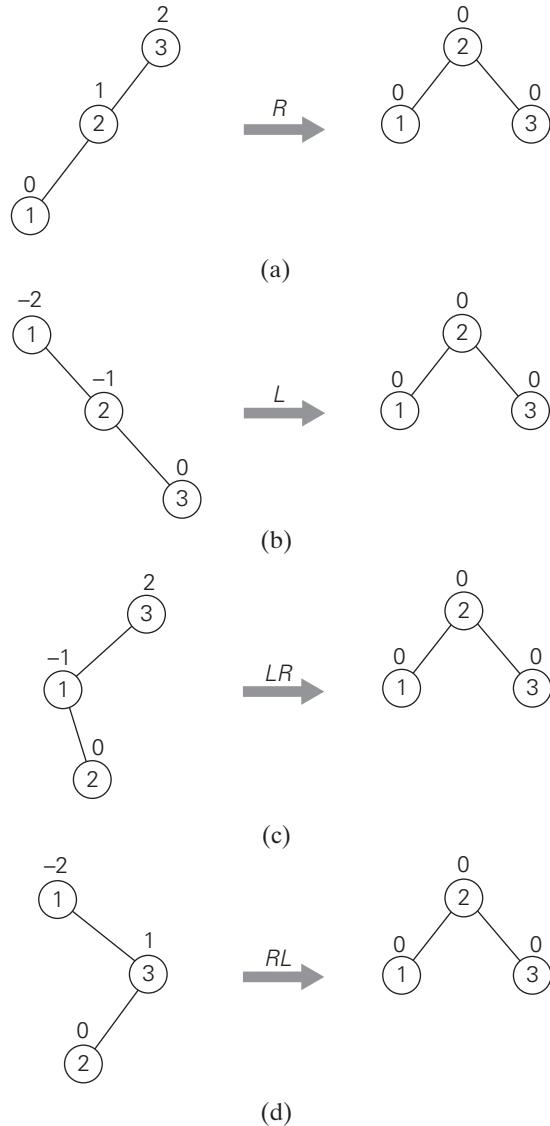


FIGURE 6.3 Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

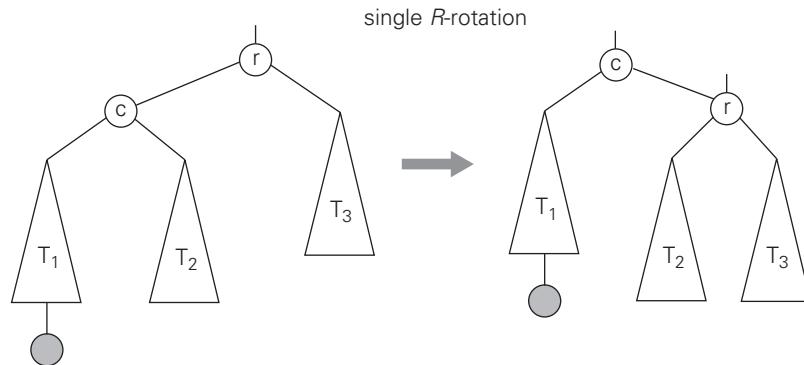


FIGURE 6.4 General form of the *R*-rotation in the AVL tree. A shaded node is the last one inserted.

The first rotation type is called the **single right rotation** or ***R*-rotation**. (Imagine rotating the edge connecting the root and its left child in the binary tree of Figure 6.3a to the right.) Figure 6.4 presents the single *R*-rotation in its most general form. Note that this rotation is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion.

The symmetric **single left rotation** or ***L*-rotation** is the mirror image of the single *R*-rotation. It is performed after a new key is inserted into the right subtree of the right child of a tree whose root had the balance of -1 before the insertion. (We will ask you to draw a diagram of the general case of the single *L*-rotation in the exercises.)

The second rotation type is called the **double left-right rotation (LR-rotation)**. It is, in fact, a combination of two rotations: we perform the *L*-rotation of the left subtree of root *r* followed by the *R*-rotation of the new tree rooted at *r* (Figure 6.5). It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.

The **double right-left rotation (RL-rotation)** is the mirror image of the double *LR*-rotation and is left for the exercises.

Note that the rotations are not trivial transformations, though, fortunately, they can be done in a constant time. Not only should they guarantee that a resulting tree is balanced, but they should also preserve the basic requirements of a binary search tree. For example, in the initial tree of Figure 6.4, all the keys of subtree *T₁* are smaller than *c*, which is smaller than all the keys of subtree *T₂*, which are smaller than *r*, which is smaller than all the keys of subtree *T₃*. And the same relationships among the key values hold, as they must, for the balanced tree after the rotation.

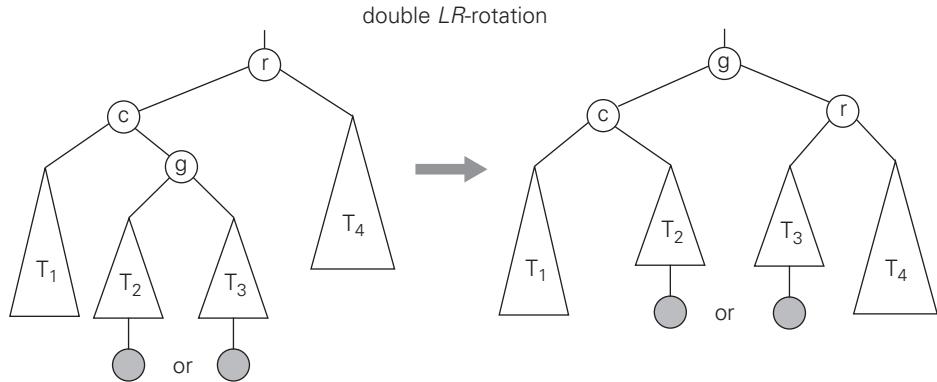


FIGURE 6.5 General form of the double *LR*-rotation in the AVL tree. A shaded node is the last one inserted. It can be either in the left subtree or in the right subtree of the root's grandchild.

An example of constructing an AVL tree for a given list of numbers is shown in Figure 6.6. As you trace the algorithm's operations, keep in mind that if there are several nodes with the ± 2 balance, the rotation is done for the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.

How efficient are AVL trees? As with any search tree, the critical characteristic is the tree's height. It turns out that it is bounded both above and below by logarithmic functions. Specifically, the height h of any AVL tree with n nodes satisfies the inequalities

$$\lfloor \log_2 n \rfloor \leq h < 1.4405 \log_2(n+2) - 1.3277.$$

(These weird-looking constants are roundoffs of some irrational numbers related to Fibonacci numbers and the golden ratio—see Section 2.5.)

The inequalities immediately imply that the operations of searching and insertion are $\Theta(\log n)$ in the worst case. Getting an exact formula for the average height of an AVL tree constructed for random lists of keys has proved to be difficult, but it is known from extensive experiments that it is about $1.01\log_2 n + 0.1$ except when n is small ([KnuIII], p. 468). Thus, searching in an AVL tree requires, on average, almost the same number of comparisons as searching in a sorted array by binary search. The operation of key deletion in an AVL tree is considerably more difficult than insertion, but fortunately it turns out to be in the same efficiency class as insertion, i.e., logarithmic.

These impressive efficiency characteristics come at a price, however. The drawbacks of AVL trees are frequent rotations, the need to maintain balances for the tree's nodes, and overall complexity, especially of the deletion operation.

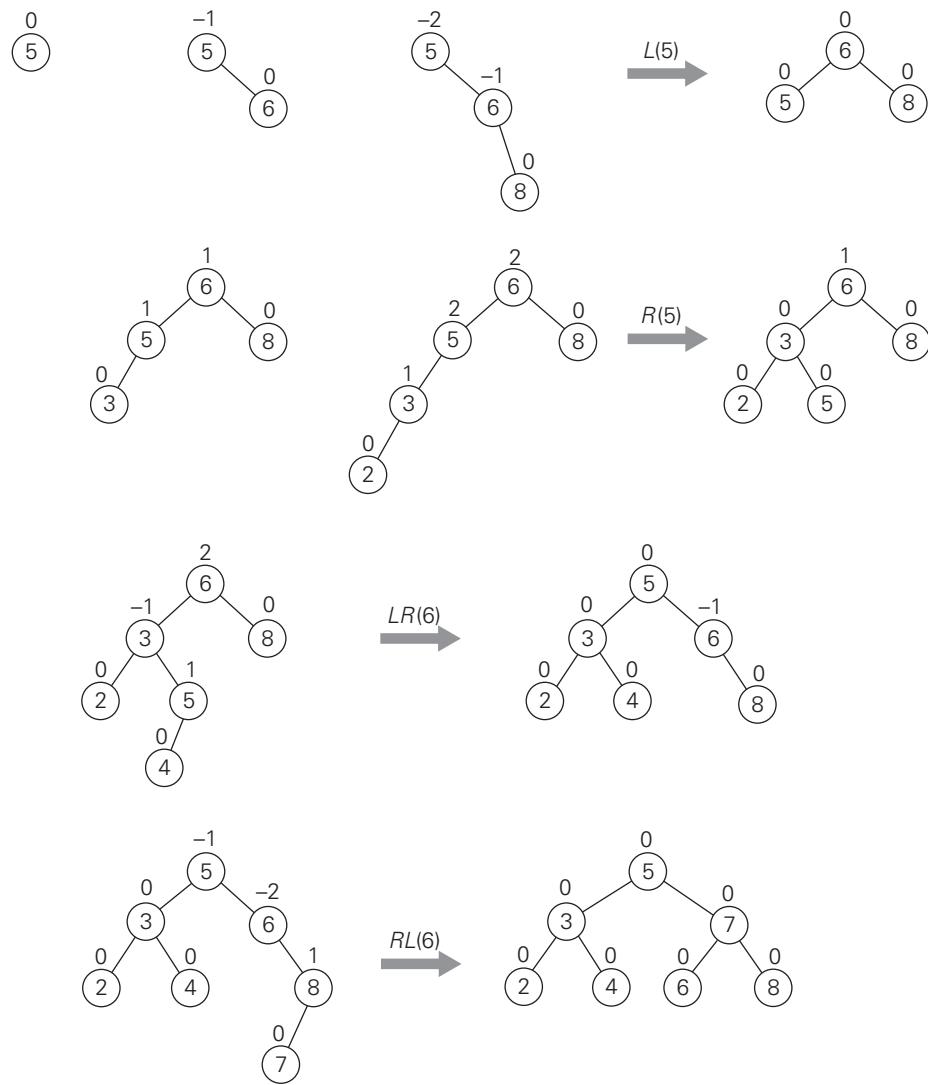


FIGURE 6.6 Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

These drawbacks have prevented AVL trees from becoming the standard structure for implementing dictionaries. At the same time, their underlying idea—that of rebalancing a binary search tree via rotations—has proved to be very fruitful and has led to discoveries of other interesting variations of the classical binary search tree.

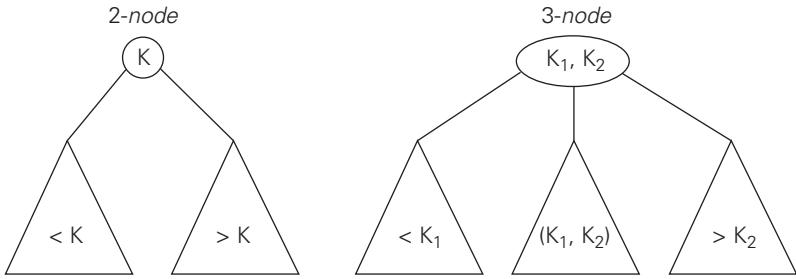


FIGURE 6.7 Two kinds of nodes of a 2-3 tree

2-3 Trees

As we mentioned at the beginning of this section, the second idea of balancing a search tree is to allow more than one key in the same node. The simplest implementation of this idea is 2-3 trees, introduced by the U.S. computer scientist John Hopcroft in 1970 (see [AHU74]). A **2-3 tree** is a tree that can have nodes of two kinds: 2-nodes and 3-nodes. A **2-node** contains a single key K and has two children: the left child serves as the root of a subtree whose keys are less than K and the right child serves as the root of a subtree whose keys are greater than K . (In other words, a 2-node is the same kind of node we have in the classical binary search tree.) A **3-node** contains two ordered keys K_1 and K_2 ($K_1 < K_2$) and has three children. The leftmost child serves as the root of a subtree with keys less than K_1 , the middle child serves as the root of a subtree with keys between K_1 and K_2 , and the rightmost child serves as the root of a subtree with keys greater than K_2 (Figure 6.7).

The last requirement of the 2-3 tree is that all its leaves must be on the same level, i.e., a 2-3 tree is always **height-balanced**: the length of a path from the root of the tree to a leaf must be the same for every leaf. It is this property that we “buy” by allowing more than a single key in a search tree’s node.

Searching for a given key K in a 2-3 tree is quite straightforward. We start with the root. If the root is a 2-node, we act as if it were a binary search tree: we either stop if K is equal to the root’s key or continue the search in the left or right subtree if K is, respectively, smaller or larger than the root’s key. If the root is a 3-node, we know after no more than two key comparisons whether the search can be stopped (if K is equal to one of the root’s keys) or in which of the root’s three subtrees it needs to be continued.

Inserting a new key in a 2-3 tree is done as follows. First of all, we always insert a new key K at a leaf except for the empty tree. The appropriate leaf is found by performing a search for K . If the leaf in question is a 2-node, we insert K there as either the first or the second key, depending on whether K is smaller or larger than the node’s old key. If the leaf is a 3-node, we split the leaf in two: the smallest

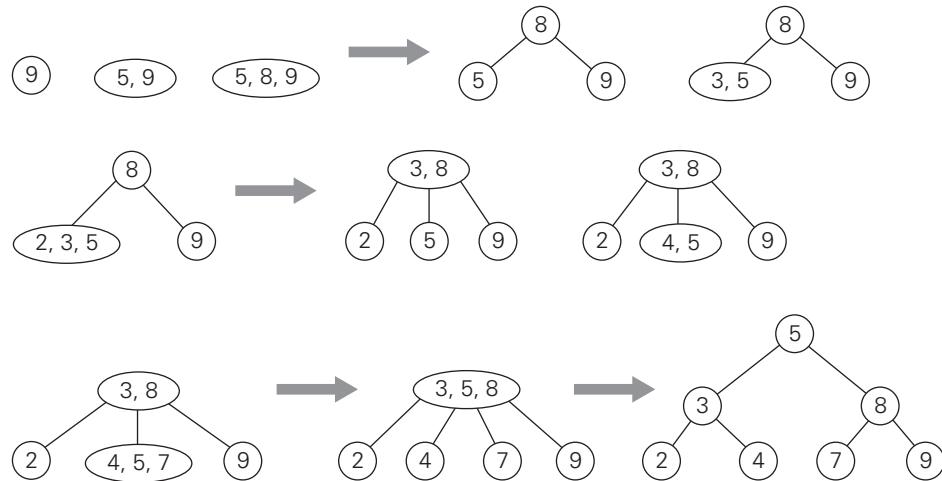


FIGURE 6.8 Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

of the three keys (two old ones and the new key) is put in the first leaf, the largest key is put in the second leaf, while the middle key is promoted to the old leaf's parent. (If the leaf happens to be the tree's root, a new root is created to accept the middle key.) Note that promotion of a middle key to its parent can cause the parent's overflow (if it was a 3-node) and hence can lead to several node splits along the chain of the leaf's ancestors.

An example of a 2-3 tree construction is given in Figure 6.8.

As for any search tree, the efficiency of the dictionary operations depends on the tree's height. So let us first find an upper bound for it. A 2-3 tree of height h with the smallest number of keys is a full tree of 2-nodes (such as the final tree in Figure 6.8 for $h = 2$). Therefore, for any 2-3 tree of height h with n nodes, we get the inequality

$$n \geq 1 + 2 + \dots + 2^h = 2^{h+1} - 1,$$

and hence

$$h \leq \log_2(n + 1) - 1.$$

On the other hand, a 2-3 tree of height h with the largest number of keys is a full tree of 3-nodes, each with two keys and three children. Therefore, for any 2-3 tree with n nodes,

$$n \leq 2 \cdot 1 + 2 \cdot 3 + \dots + 2 \cdot 3^h = 2(1 + 3 + \dots + 3^h) = 3^{h+1} - 1,$$

and hence

$$h \geq \log_3(n + 1) - 1.$$

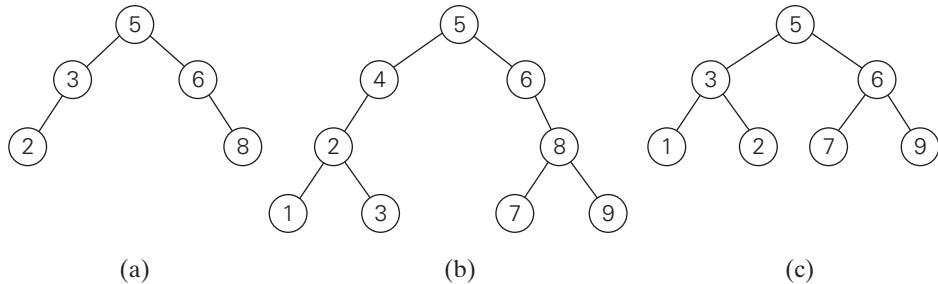
These lower and upper bounds on height h ,

$$\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1,$$

imply that the time efficiencies of searching, insertion, and deletion are all in $\Theta(\log n)$ in both the worst and average case. We will consider a very important generalization of 2-3 trees, called B -trees, in Section 7.4.

Exercises 6.3

- 1.** Which of the following binary trees are AVL trees?



Exercises 6.3

- 2. a.** For $n = 1, 2, 3, 4$, and 5 , draw all the binary trees with n nodes that satisfy the balance requirement of AVL trees.
b. Draw a binary tree of height 4 that can be an AVL tree and has the smallest number of nodes among all such trees.
- 3.** Draw diagrams of the single L -rotation and of the double RL -rotation in their general form.
- 4.** For each of the following lists, construct an AVL tree by inserting their elements successively, starting with the empty tree:
- 1, 2, 3, 4, 5, 6
 - 6, 5, 4, 3, 2, 1
 - 3, 6, 5, 1, 2, 4
- 5. a.** For an AVL tree containing real numbers, design an algorithm for computing the range (i.e., the difference between the largest and smallest numbers) in the tree and determine its worst-case efficiency.
b. True or false: The smallest and the largest keys in an AVL tree can always be found on either the last level or the next-to-last level.
- 6.** Write a program for constructing an AVL tree for a given list of n distinct integers.

7. a. Construct a 2-3 tree for the list C, O, M, P, U, T, I, N, G. (Use the alphabetical order of the letters and insert them successively starting with the empty tree.)
- b. Assuming that the probabilities of searching for each of the keys (i.e., the letters) are the same, find the largest number and the average number of key comparisons for successful searches in this tree.
8. Let T_B and T_{2-3} be, respectively, a classical binary search tree and a 2-3 tree constructed for the same list of keys inserted in the corresponding trees in the same order. True or false: Searching for the same key in T_{2-3} always takes fewer or the same number of key comparisons as searching in T_B .
9. For a 2-3 tree containing real numbers, design an algorithm for computing the range (i.e., the difference between the largest and smallest numbers) in the tree and determine its worst-case efficiency.
10. Write a program for constructing a 2-3 tree for a given list of n integers.

6.4 Heaps and Heapsort

The data structure called the “heap” is definitely not a disordered pile of items as the word’s definition in a standard dictionary might suggest. Rather it is a clever, partially ordered data structure that is especially suitable for implementing priority queues. Recall that a ***priority queue*** is a set of items with an orderable characteristic called an item’s ***priority***, with the following operations:

- Finding an item with the highest (i.e., largest) priority
- Deleting an item with the highest priority
- Adding a new item to the set

It is primarily an efficient implementation of these operations that makes the heap both interesting and useful. The heap is also the data structure that serves as a cornerstone of a theoretically important sorting algorithm called heapsort. We discuss this algorithm after we define the heap and investigate its basic properties.

Notion of the Heap

DEFINITION A ***heap*** can be defined as a binary tree with keys assigned to its nodes (one key per node) provided the following two conditions are met:

1. The tree’s shape requirement—The binary tree is ***essentially complete*** (or simply ***complete***), that is, all its levels are full except possibly the last level, where only some rightmost leaves may be missing.

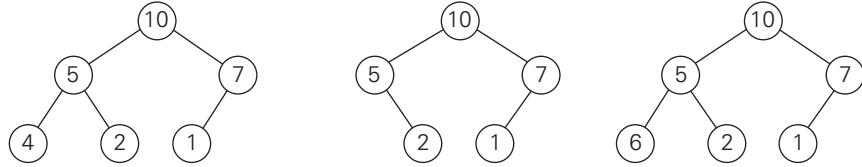


FIGURE 6.9 Illustration of the definition of “heap”: only the leftmost tree is a heap.

2. The parental dominance requirement—The key at each node is greater than or equal to the keys at its children. (This condition is considered automatically satisfied for all leaves.)⁵

For example, consider the trees of Figure 6.9. The first tree there is a heap, the second one is not a heap because the tree’s shape requirement is violated, and the third one is not a heap because the parental dominance requirement fails for the node with key 5.

Note that key values in a heap are ordered top down; that is, a sequence of values on any path from the root to a leaf is decreasing (nonincreasing, if equal keys are allowed). However, there is no left-to-right order in key values; that is, there is no relationship among key values for nodes either on the same level of the tree or, more generally, in the left and right subtrees of the same node.

Here is a list of important properties of heaps, which are not difficult to prove (check these properties for the heap of Figure 6.10, as an example).

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap’s elements in positions 1 through n of such an array, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,
 - a. the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions.
 - b. the children of a key in the array’s parental position i ($1 \leq i \leq \lfloor n/2 \rfloor$) will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i ($2 \leq i \leq n$) will be in position $\lfloor i/2 \rfloor$.

5. Some authors require the key at each node to be *less* than or equal to the keys at its children. We call this variation a **min-heap**.

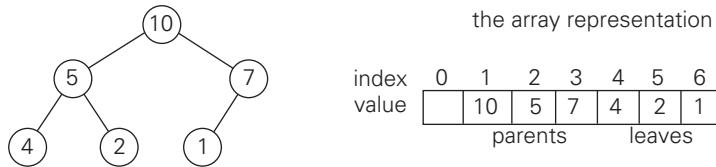


FIGURE 6.10 Heap and its array representation

As an example, you can check these properties for the heap of Figure 6.10.

Thus, we could also define a heap as an array $H[1..n]$ in which every element in position i in the first half of the array is greater than or equal to the elements in positions $2i$ and $2i + 1$, i.e.,

$$H[i] \geq \max\{H[2i], H[2i + 1]\} \quad \text{for } i = 1, \dots, \lfloor n/2 \rfloor.$$

(Of course, if $2i + 1 > n$, just $H[i] \geq H[2i]$ needs to be satisfied.) While the ideas behind the majority of algorithms dealing with heaps are easier to understand if we think of heaps as binary trees, their actual implementations are usually much simpler and more efficient with arrays.

How can we construct a heap for a given list of keys? There are two principal alternatives for doing that. The first is the so-called ***bottom-up heap construction*** algorithm (illustrated in Figure 6.11). It initializes the essentially complete binary tree with n nodes by placing keys in the order given and then “heapifies” the tree as follows. Starting with the last parental node and ending with the root, the algorithm checks whether the parental dominance holds for the key at this node. If it does not, the algorithm exchanges the node’s key K with the larger key of its children and checks whether the parental dominance holds for K in its new position. This process continues until the parental dominance requirement for K is satisfied. (Eventually it has to because it holds automatically for any key in a leaf.) After completing the “heapification” of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node’s immediate predecessor. The algorithm stops after this is done for the tree’s root.

Before we introduce a pseudocode of the bottom-up heap construction algorithm, we should make one more remark. Since the value of a node’s key does not change during the process of sifting it down the tree, there is no need to involve it in intermediate swaps. You can think of this improvement as swapping the empty node with larger keys in its children until a final position is reached where it accepts the “erased” value again.

ALGORITHM *HeapBottomUp*($H[1..n]$)

```
//Constructs a heap from the elements of a given array
// by the bottom-up algorithm
//Input: An array  $H[1..n]$  of orderable items
```

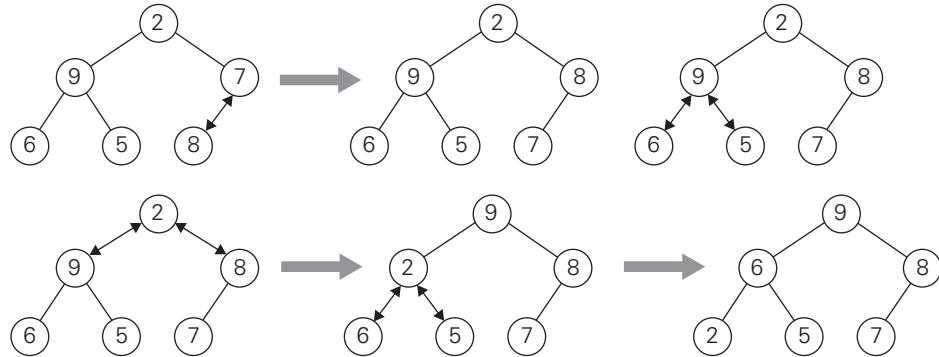


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8

```

//Output: A heap  $H[1..n]$ 
for  $i \leftarrow \lfloor n/2 \rfloor$  downto 1 do
     $k \leftarrow i$ ;  $v \leftarrow H[k]$ 
     $heap \leftarrow \text{false}$ 
    while not  $heap$  and  $2 * k \leq n$  do
         $j \leftarrow 2 * k$ 
        if  $j < n$  //there are two children
            if  $H[j] < H[j + 1]$   $j \leftarrow j + 1$ 
            if  $v \geq H[j]$ 
                 $heap \leftarrow \text{true}$ 
            else  $H[k] \leftarrow H[j]$ ;  $k \leftarrow j$ 
         $H[k] \leftarrow v$ 

```

How efficient is this algorithm in the worst case? Assume, for simplicity, that $n = 2^k - 1$ so that a heap's tree is full, i.e., the largest number of nodes occur on each level. Let h be the height of the tree; according to the first property of heaps in the list at the beginning of the section, $h = \lfloor \log_2 n \rfloor$ (or just $\lceil \log_2(n + 1) \rceil - 1 = k - 1$ for the specific values of n we are considering). Each key on level i of the tree will travel to the leaf level h in the worst case of the heap construction algorithm. Since moving to the next level down requires two comparisons—one to find the larger child and the other to determine whether the exchange is required—the total number of key comparisons involving a key on level i will be $2(h - i)$. So the total number of key comparisons in the worst case will be

$$C_{\text{worst}}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h - i) = \sum_{i=0}^{h-1} 2(h - i)2^i = 2(n - \log_2(n + 1)),$$

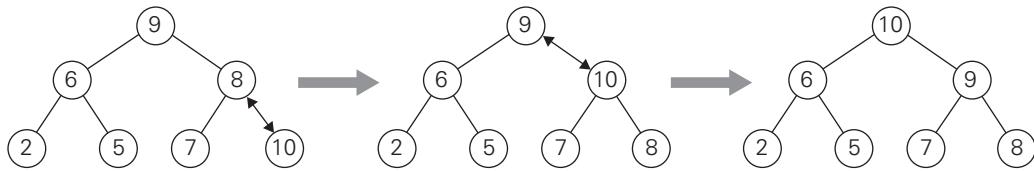


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

where the validity of the last equality can be proved either by using the closed-form formula for the sum $\sum_{i=1}^h i2^i$ (see Appendix A) or by mathematical induction in h . Thus, with this bottom-up algorithm, a heap of size n can be constructed with fewer than $2n$ comparisons.

The alternative (and less efficient) algorithm constructs a heap by successive insertions of a new key into a previously constructed heap; some people call it the **top-down heap construction** algorithm. So how can we insert a new key K into a heap? First, attach a new node with key K in it after the last leaf of the existing heap. Then sift K up to its appropriate place in the new heap as follows. Compare K with its parent's key: if the latter is greater than or equal to K , stop (the structure is a heap); otherwise, swap these two keys and compare K with its new parent. This swapping continues until K is not greater than its last parent or it reaches the root (illustrated in Figure 6.12). In this algorithm, too, we can sift up an empty node until it reaches its proper position, where it will get K 's value.

Obviously, this insertion operation cannot require more key comparisons than the heap's height. Since the height of a heap with n nodes is about $\log_2 n$, the time efficiency of insertion is in $O(\log n)$.

How can we delete an item from a heap? We consider here only the most important case of deleting the root's key, leaving the question about deleting an arbitrary key in a heap for the exercises. (Authors of textbooks like to do such things to their readers, do they not?) So deleting the root's key from a heap can be done with the following algorithm (illustrated in Figure 6.13).

Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 “Heapify” the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for K : if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

The efficiency of deletion is determined by the number of key comparisons needed to “heapify” the tree after the swap has been made and the size of the tree is decreased by 1. Since it cannot require more key comparisons than twice the heap's height, the time efficiency of deletion is in $O(\log n)$ as well.

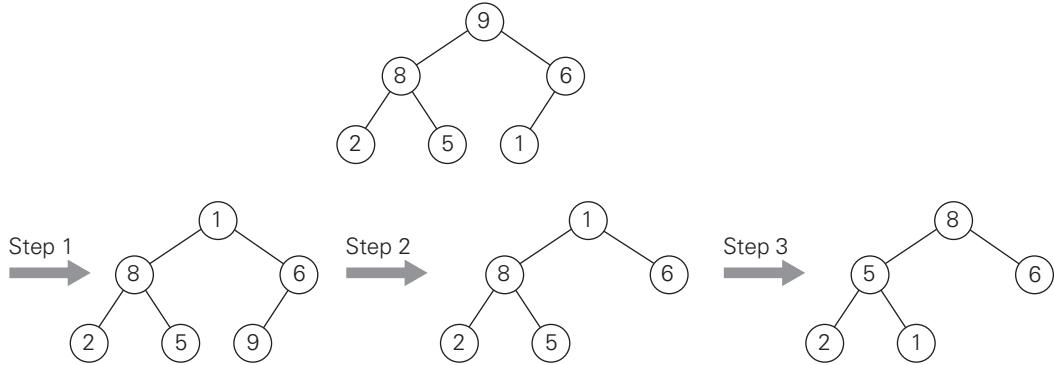


FIGURE 6.13 Deleting the root’s key from a heap. The key to be deleted is swapped with the last key, after which the smaller tree is “heapified” by exchanging the new key at its root with the larger key at its children until the parental dominance requirement is satisfied.

Heapsort

Now we can describe **heapsort**—an interesting sorting algorithm discovered by J. W. J. Williams [Wil64]. This is a two-stage algorithm that works as follows.

Stage 1 (heap construction): Construct a heap for a given array.

Stage 2 (maximum deletions): Apply the root-deletion operation $n - 1$ times to the remaining heap.

As a result, the array elements are eliminated in decreasing order. But since under the array implementation of heaps, an element being deleted is placed last, the resulting array will be exactly the original array sorted in ascending order. Heapsort is traced on a specific input in Figure 6.14. (The same input as the one of Figure 6.11 is intentionally used so that you can compare the tree and array implementations of the bottom-up heap construction algorithm.)

Since we already know that the heap construction stage of the algorithm is in $O(n)$, we need to investigate just the time efficiency of the second stage. For the number of key comparisons, $C(n)$, needed for eliminating the root keys from the heaps of diminishing sizes from n to 2, we get the following inequality:

$$\begin{aligned} C(n) &\leq 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \dots + 2\lfloor \log_2 1 \rfloor \leq 2 \sum_{i=1}^{n-1} \log_2 i \\ &\leq 2 \sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1) \log_2(n-1) \leq 2n \log_2 n. \end{aligned}$$

This means that $C(n) \in O(n \log n)$ for the second stage of heapsort. For both stages, we get $O(n) + O(n \log n) = O(n \log n)$. A more detailed analysis shows that, in

Stage 1 (heap construction)	Stage 2 (maximum deletions)
2 9 7 6 5 8	9 6 8 2 5 7
2 9 8 6 5 7	7 6 8 2 5 19
2 9 8 6 5 7	8 6 7 2 5
9 2 8 6 5 7	5 6 7 2 18
9 6 8 2 5 7	7 6 5 2
	2 6 5 17
	6 2 5
	5 2 16
	5 2
	2 15
	2

FIGURE 6.14 Sorting the array 2, 9, 7, 6, 5, 8 by heapsort

fact, the time efficiency of heapsort is in $\Theta(n \log n)$ in both the worst and average cases. Thus, heapsort's time efficiency falls in the same class as that of mergesort; and, unlike the latter, it is in place, i.e., it does not require any extra storage. Timing experiments on random files show that heapsort runs more slowly than quicksort but is competitive with mergesort.

Exercises 6.4

1.
 - a. Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by the bottom-up algorithm.
 - b. Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by successive key insertions (top-down algorithm).
 - c. Is it always true that the bottom-up and top-down algorithms yield the same heap for the same input?
2. Outline an algorithm for checking whether an array $H[1..n]$ is a heap and determine its time efficiency.
3.
 - a. Find the smallest and the largest number of keys that a heap of height h can contain.
 - b. Prove that the height of a heap with n nodes is equal to $\lfloor \log_2 n \rfloor$.

4. Prove the following equality used in Section 6.4

$$\sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)), \text{ where } n = 2^{h+1} - 2.$$

5. a. Design an efficient algorithm for finding and deleting an element of the smallest value in a heap and determine its time efficiency.
 b. Design an efficient algorithm for finding and deleting an element of a given value v in a given heap H and determine its time efficiency.
6. Sort the following lists by heapsort by using the array representation of heaps:
 a. 1, 2, 3, 4, 5 (in increasing order)
 b. 5, 4, 3, 2, 1 (in increasing order)
 c. S, O, R, T, I, N, G (in alphabetical order)
7. Is heapsort a stable sorting algorithm?
8. What variety of the transform-and-conquer technique does heapsort represent?
9. Implement three advanced sorting algorithms—mergesort, quicksort, and heapsort—in the language of your choice and investigate their performance on arrays of sizes $n = 10^2, 10^3, 10^4, 10^5$ and 10^6 . For each of these sizes, consider
 a. randomly generated files of integers in the range $[1..n]$.
 b. increasing files of integers $1, 2, \dots, n$.
 c. decreasing files of integers $n, n-1, \dots, 1$.
10. Imagine a handful of uncooked spaghetti, individual rods whose lengths represent numbers that need to be sorted.
 a. Outline a “spaghetti sort”—a sorting algorithm that takes advantage of this unorthodox representation.
 b. What does this example of computer science folklore (see [Dew93]) have to do with the topic of this chapter in general and heapsort in particular?



Exercises 6.4

6.5 Horner's Rule and Binary Exponentiation

In this section we discuss the problem of computing the value of a polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 \quad (6.1)$$

at a given point x and its important special case of computing x^n . Polynomials constitute the most important class of functions because they possess a wealth of good properties on the one hand and can be used for approximating other types of functions on the other. The problem of manipulating polynomials efficiently

has been important for several centuries; new discoveries were still being made in the last 50 years. By far the most important of them was the *fast Fourier transform (FFT)*. The practical importance of this remarkable algorithm, which is based on representing a polynomial by its values at specially chosen points, was such that some people consider it one of the most important algorithmic discoveries of all times. Because of its relative complexity, we do not discuss the FFT algorithm in this book. An interested reader will find a wealth of literature on the subject including reasonably accessible treatments in such textbooks as [Sed88] and [CLRS01].

Horner's Rule

Horner's rule is an old but very elegant and efficient algorithm for evaluating a polynomial. It is named after the British mathematician W. G. Horner, who published it in the early 19th century. But according to Knuth ([KnuII], p. 486), the method was used by Isaac Newton 150 years before Horner. You will appreciate this method much more if you first design an algorithm for the polynomial evaluation problem by yourself and investigate its efficiency (see Problems 1 and 2 in the exercises to this section).

Horner's rule is a good example of the representation change technique since it is based on representing $p(x)$ by a formula different from (6.1). This new formula is obtained from (6.1) by successively taking x as a common factor in the remaining polynomials of diminishing degrees:

$$p(x) = (\dots (a_n x + a_{n-1}) x + \dots) x + a_0. \quad (6.2)$$

For example, for the polynomial $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$, we get

$$\begin{aligned} p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 \\ &= x(2x^3 - x^2 + 3x + 1) - 5 \\ &= x(x(2x^2 - x + 3) + 1) - 5 \\ &= x(x(x(2x - 1) + 3) + 1) - 5. \end{aligned} \quad (6.3)$$

It is in formula (6.2) that we will substitute a value of x at which the polynomial needs to be evaluated. It is hard to believe that this is a way to an efficient algorithm, but the unpleasant appearance of formula (6.2) is just that, an appearance. As we shall see, there is no need to go explicitly through transformation leading to it: all we need is an original list of the polynomial's coefficients.

The pen-and-pencil calculation can be conveniently organized with a two-row table. The first row contains the polynomial's coefficients (including all the coefficients equal to zero, if any) listed from the highest a_n to the lowest a_0 . The second row is used to store intermediate results except for its first entry which is a_n . After this initialization, the next table's entry is computed as the x 's value times the last entry in the second row plus the next coefficient from the first row. The final entry computed in this fashion is the value being sought.

EXAMPLE 1 Evaluate $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ at $x = 3$.

coefficients	2	-1	3	1	-5
$x = 3$	2	$3 \cdot 2 + (-1) = 5$	$3 \cdot 5 + 3 = 18$	$3 \cdot 18 + 1 = 55$	$3 \cdot 55 + (-5) = 160$

So $p(3) = 160$. (On comparing the table's entries with formula (6.3), you will see that $3 \cdot 2 + (-1) = 5$ is the value of $2x - 1$ at $x = 3$, $3 \cdot 5 + 3 = 18$ is the value of $x(2x - 1) + 3$ at $x = 3$, $3 \cdot 18 + 1 = 55$ is the value of $x(x(2x - 1) + 3) + 1$ at $x = 3$, and finally, $3 \cdot 55 + (-5) = 160$ is the value of $x(x(x(2x - 1) + 3) + 1) - 5 = p(x)$ at $x = 3$.) ■

The pseudocode for this algorithm is the shortest one imaginable for a non-trivial algorithm:

ALGORITHM *Horner($P[0..n]$, x)*

```
//Evaluates a polynomial at a given point by Horner's rule
//Input: An array  $P[0..n]$  of coefficients of a polynomial of degree  $n$ 
//        (stored from the lowest to the highest) and a number  $x$ 
//Output: The value of the polynomial at  $x$ 
 $p \leftarrow a_n$ 
for  $i \leftarrow n - 1$  downto 0 do
     $p \leftarrow x * p + a_i$ 
return  $p$ 
```

The number of multiplications and the number of additions are given by the same sum:

$$M(n) = A(n) = \sum_{i=0}^{n-1} 1 = n.$$

To appreciate how efficient Horner's rule is, consider only the first term of a polynomial of degree n : $a_n x^n$. Just computing this single term by the brute-force algorithm would require n multiplications, whereas Horner's rule computes, in addition to this term, $n - 1$ other terms, and it still uses the same number of multiplications! It is not surprising that Horner's rule is an optimal algorithm for polynomial evaluation without preprocessing the polynomial's coefficients. But it took scientists 150 years after Horner's publication to come to the realization that such a question was worth investigating.

Horner's rule also has some useful by-products. The intermediate numbers generated by the algorithm in the process of evaluating $p(x)$ at some point x_0 turn out to be the coefficients of the quotient of the division of $p(x)$ by $x - x_0$, while the

final result, in addition to being $p(x_0)$, is equal to the remainder of this division. Thus, according to the example, the quotient and the remainder of the division of $2x^4 - x^3 + 3x^2 + x - 5$ by $x - 3$ are $2x^3 + 5x^2 + 18x + 55$ and 150, respectively. This division algorithm, known as *synthetic division*, is more convenient than so-called “long division.” (However, unlike the long division, it is applicable only to divisions by $x - c$ where c is some constant.)

Binary Exponentiation

The amazing efficiency of Horner’s rule fades if the method is applied to computing a^n , which is the value of x^n at $x = a$. In fact, it degenerates to the brute-force multiplication of a by itself, with wasteful additions of zeros in between. Since computing a^n (actually, $a^n \bmod m$) is an essential operation in several important primality-testing and encryption methods, we consider now two algorithms for computing a^n that are based on the representation change idea. They both exploit the binary representation of exponent n , but one of them processes this binary string left to right whereas the second does it right to left.

Let

$$n = b_I \dots b_i \dots b_0$$

be the bit string representing a positive integer n in the binary number system. This means that the value of n can be computed as the value of the polynomial

$$p(x) = b_I x^I + \dots + b_i x^i + \dots + b_0 \quad (6.4)$$

at $x = 2$. For example, if $n = 13$, its binary representation is 1101 and

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0.$$

Let us now compute the value of this polynomial by applying Horner’s rule and see what the method’s operations imply for computing the power

$$a^n = a^{p(2)} = a^{b_I 2^I + \dots + b_i 2^i + \dots + b_0}.$$

Horner’s rule for the binary polynomial $p(2)$	Implications for $a^n = a^{p(2)}$
$p \leftarrow 1$ //the leading digit is always 1 for $n \geq 1$	$a^p \leftarrow a^1$
for $i \leftarrow I - 1$ downto 0 do	for $i \leftarrow I - 1$ downto 0 do
$p \leftarrow 2p + b_i$	$a^p \leftarrow a^{2p+b_i}$

But

$$a^{2p+b_i} = a^{2p} \cdot a^{b_i} = (a^p)^2 \cdot a^{b_i} = \begin{cases} (a^p)^2 & \text{if } b_i = 0 \\ (a^p)^2 \cdot a & \text{if } b_i = 1 \end{cases}.$$

Thus, after initializing the accumulator's value to a , we can scan the bit string representing the exponent to always square the last value of the accumulator and, if the current binary digit is 1, also to multiply it by a . These observations lead to the following ***left-to-right binary exponentiation*** method of computing a^n .

ALGORITHM *LeftRightBinaryExponentiation(a, b(n))*

```
//Computes  $a^n$  by the left-to-right binary exponentiation algorithm
//Input: A number  $a$  and a list  $b(n)$  of binary digits  $b_1, \dots, b_0$ 
//      in the binary expansion of a positive integer  $n$ 
//Output: The value of  $a^n$ 
product ← a
for i ← I - 1 downto 0 do
    product ← product * product
    if  $b_i = 1$  product ← product * a
return product
```

EXAMPLE Compute a^{13} by the left-to-right binary exponentiation algorithm. Here, $n = 13 = 1101_2$. So we have

binary digits of n	1	1	0	1
product accumulator	a	$a^2 \cdot a = a^3$	$(a^3)^2 = a^6$	$(a^6)^2 \cdot a = a^{13}$

■

Since the algorithm makes one or two multiplications on each repetition of its only loop, the total number of multiplications $M(n)$ made by it in computing a^n is

$$(b - 1) \leq M(n) \leq 2(b - 1),$$

where b is the length of the bit string representing the exponent n . Taking into account that $b - 1 = \lfloor \log_2 n \rfloor$, we can conclude that the efficiency of the left-to-right binary exponentiation is logarithmic. Thus, this algorithm is in a better efficiency class than the brute-force exponentiation, which always requires $n - 1$ multiplications.

The ***right-to-left binary exponentiation*** uses the same binary polynomial $p(2)$ (see (6.4)) yielding the value of n . But rather than applying Horner's rule to it as the previous method did, this one exploits it differently:

$$a^n = a^{b_I 2^I + \dots + b_i 2^i + \dots + b_0} = a^{b_I 2^I} \cdot \dots \cdot a^{b_i 2^i} \cdot \dots \cdot a^{b_0}.$$

Thus, a^n can be computed as the product of the terms

$$a^{b_i 2^i} = \begin{cases} a^{2^i} & \text{if } b_i = 1 \\ 1 & \text{if } b_i = 0 \end{cases},$$

that is, the product of consecutive terms a^{2^i} , skipping those for which the binary digit b_i is zero. In addition, we can compute a^{2^i} by simply squaring the same term we computed for the previous value of i since $a^{2^i} = (a^{2^{i-1}})^2$. So we compute all such powers of a from the smallest to the largest (from right to left), but we include in the product accumulator only those whose corresponding binary digit is 1. Here is a pseudocode of this algorithm.

ALGORITHM *RightLeftBinaryExponentiation(a, b(n))*

```
//Computes  $a^n$  by the right-to-left binary exponentiation algorithm
//Input: A number  $a$  and a list  $b(n)$  of binary digits  $b_1, \dots, b_0$ 
//       in the binary expansion of a nonnegative integer  $n$ 
//Output: The value of  $a^n$ 
term  $\leftarrow a$  //initializes  $a^{2^0}$ 
if  $b_0 = 1$  product  $\leftarrow a$ 
else product  $\leftarrow 1$ 
for  $i \leftarrow 1$  to  $I$  do
    term  $\leftarrow$  term * term
    if  $b_i = 1$  product  $\leftarrow$  product * term
return product
```

EXAMPLE 3 Compute a^{13} by the right-to-left binary exponentiation method. Here, $n = 13 = 1101_2$. So we have the following table filled in from right to left.

1	1	0	1	binary digits of n
a^8	a^4	a^2	a	terms a^{2^i}
$a^5 \cdot a^8 = a^{13}$	$a \cdot a^4 = a^5$		a	product accumulator

Obviously, the algorithm's efficiency is also logarithmic for the same reason the left-to-right binary multiplication is. The usefulness of both binary exponentiation algorithms is reduced somewhat by their reliance on availability of the explicit binary expansion of exponent n . Problem 8 in the exercises asks you to design an algorithm that does not have this shortcoming.

Exercises 6.5

1. Consider the following brute-force algorithm for evaluating a polynomial.

ALGORITHM *BruteForcePolynomialEvaluation($P[0..n]$, x)*

```
//The algorithm computes the value of polynomial  $P$  at a given point  $x$ 
//by the “highest to lowest term” brute-force algorithm
//Input: An array  $P[0..n]$  of the coefficients of a polynomial of degree  $n$ ,
//       stored from the lowest to the highest and a number  $x$ 
//Output: The value of the polynomial at the point  $x$ 
 $p \leftarrow 0.0$ 
for  $i \leftarrow n$  downto 0 do
     $power \leftarrow 1$ 
    for  $j \leftarrow 1$  to  $i$  do
         $power \leftarrow power * x$ 
         $p \leftarrow p + P[i] * power$ 
return  $p$ 
```

Find the total number of multiplications and the number of additions made by this algorithm.

2. Write a pseudocode for the brute-force polynomial evaluation that stems from substituting a given value of the variable into the polynomial’s formula and evaluating it from the lowest term to the highest one. Determine the number of multiplications and the number of additions made by this algorithm.
3. a. Estimate how much faster Horner’s rule is compared to the “lowest to highest term” brute-force algorithm of Problem 2 if (i) the time of one multiplication is significantly larger than the time of one addition; (ii) the time of one multiplication is about the same as the time of one addition.
b. Is Horner’s rule more time efficient at the expense of being less space efficient than the brute-force algorithm?
4. a. Apply Horner’s rule to evaluate the polynomial

$$p(x) = 3x^4 - x^3 + 2x + 5 \text{ at } x = -2.$$

- b. Use the results of Horner’s rule application to find the quotient and remainder of the division of $p(x)$ by $x + 2$.
5. Compare the number of multiplications and additions/subtractions needed by the “long division” of a polynomial $p(x) = a_nx^n + a_{n-1}x^{n-1} + \dots + a_0$ by $x - c$ where c is some constant with the number of these operations in “synthetic division.”

6. a. Apply the left-to-right binary exponentiation algorithm to compute a^{17} .
- b. Is it possible to extend the left-to-right binary exponentiation algorithm to work for every nonnegative integer exponent?
7. Apply the right-to-left binary exponentiation algorithm to compute a^{17} .
8. Design a nonrecursive algorithm for computing a^n that mimics the right-to-left binary exponentiation but does not explicitly use the binary representation of n .
9. Is it a good idea to use a general-purpose polynomial evaluation algorithm such as Horner's rule to evaluate the polynomial $p(x) = x^n + x^{n-1} + \dots + x + 1$?
10. According to the corollary of the Fundamental Theorem of Algebra, every polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$$

can be represented in the form

$$p(x) = a_n(x - x_1)(x - x_2) \dots (x - x_n)$$

where x_1, x_2, \dots, x_n are the roots of the polynomial (generally, complex and not necessarily distinct). Discuss which of the two representations is more convenient for each of the following operations:

- a. Polynomial evaluation at a given point
- b. Addition of two polynomials
- c. Multiplication of two polynomials

6.6 Problem Reduction

Here is my version of a well-known joke about mathematicians. Professor X, a noted mathematician, noticed that when his wife wanted to boil water for their tea, she took their kettle from a kitchen cabinet, filled it with water, and put it on the stove. Once, when his wife was away (if you have to know, she was signing her best-seller in a local bookstore), the professor had to boil water by himself. He saw that the kettle was sitting on the kitchen counter. What did Professor X do? He put the kettle in the cupboard first and then proceeded to follow his wife's routine.

The way Professor X approached his task is an example of an important problem-solving strategy called **problem reduction**. If you need to solve a problem, reduce it to another problem that you know how to solve (Figure 6.15).

The joke about the professor notwithstanding, the idea of problem reduction plays a central role in theoretical computer science where it is used to classify

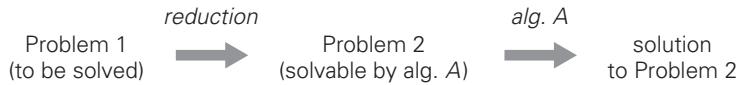


FIGURE 6.15 Problem reduction strategy

problems according to their complexity. We will touch on this classification in Chapter 10. But the strategy can be used for actual problem solving, too. The practical difficulty in applying it lies, of course, in finding a problem to which the problem at hand should be reduced. Moreover, if we want our efforts to be of practical value, we need our reduction-based algorithm to be more efficient than solving the original problem directly.

Note that we have already encountered this technique earlier in the book. In Section 6.5, for example, we mentioned the so-called synthetic division done by applying Horner's rule for polynomial evaluation. In Section 4.6, we used the following fact from analytical geometry: if $p_1 = (x_1, y_1)$, $p_2 = (x_1, y_1)$, and $p_3 = (x_3, y_3)$ are three arbitrary points in the plane, then the determinant

$$\det \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3$$

is positive if and only if the point p_3 is to the left of the directed line $\overrightarrow{p_1 p_2}$ through points p_1 and p_2 . In other words, we reduced a geometric question about the relative locations of three points to a question about the sign of a determinant. In fact, the entire idea of analytical geometry is based on reducing geometric problems to algebraic ones. And the vast majority of geometric algorithms take advantage of this historic insight by René Descartes (1596–1650). In this section, we give a few more examples of algorithms based on the problem reduction strategy.

Computing the Least Common Multiple

Recall that the **least common multiple** of two positive integers m and n , denoted $\text{lcm}(m, n)$, is defined as the smallest integer that is divisible by both m and n . For example, $\text{lcm}(24, 60) = 120$, and $\text{lcm}(11, 5) = 55$. The least common multiple is one of the most important notions in elementary arithmetic and algebra; perhaps you remember the following middle-school method for computing it. Given the prime factorizations of m and n , $\text{lcm}(m, n)$ can be computed as the product of all the common prime factors of m and n times the product of m 's prime factors that are not in n times n 's prime factors that are not in m . For example,

$$\begin{aligned} 24 &= 2 \cdot 2 \cdot 2 \cdot 3 \\ 60 &= 2 \cdot 2 \cdot 3 \cdot 5 \\ \text{lcm}(24, 60) &= (2 \cdot 2 \cdot 3) \cdot 2 \cdot 5 = 120. \end{aligned}$$

As a computational procedure, this algorithm has the same drawbacks as the middle-school algorithm for computing the greatest common divisor discussed in Section 1.1: it is inefficient and requires a list of consecutive primes.

A much more efficient algorithm for computing the least common multiple can be designed by using problem reduction. After all, there is a very efficient algorithm (Euclid's algorithm) for finding the greatest common divisor, which is a product of all the common prime factors of m and n . Can we find a formula relating $\text{lcm}(m, n)$ and $\text{gcd}(m, n)$? It is not difficult to see that the product of $\text{lcm}(m, n)$ and $\text{gcd}(m, n)$ includes every factor of m and n exactly once and hence is simply equal to the product of m and n . This observation leads to the formula

$$\text{lcm}(m, n) = \frac{m \cdot n}{\text{gcd}(m, n)},$$

where $\text{gcd}(m, n)$ can be computed very efficiently by Euclid's algorithm.

Counting Paths in a Graph

As our next example, we will consider the problem of counting paths between two vertices in a graph. It is not difficult to prove by mathematical induction that the number of different paths of length $k > 0$ from the i th vertex to the j th vertex of a graph (undirected or directed) equals the (i, j) th element of A^k where A is the adjacency matrix of the graph. (Incidentally, the exponentiation algorithms we discussed before for computing powers of numbers are applicable to matrices as well.) Thus, the problem of counting a graph's paths can be solved with an algorithm for computing an appropriate power of its adjacency matrix.

As a specific example, consider the graph of Figure 6.16. Its adjacency matrix A and its square A^2 indicate the number of paths of length 1 and 2, respectively, between the corresponding vertices of the graph. In particular, there are three paths of length 2 that start and end at vertex a : $a - b - a$, $a - c - a$, and $a - d - a$ but only one path of length 2 from a to c : $a - d - c$.

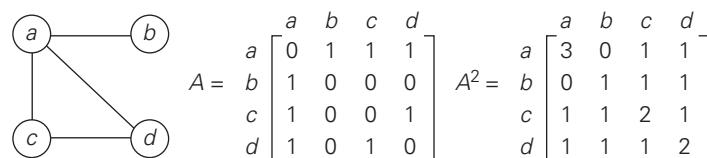


FIGURE 6.16 A graph, its adjacency matrix A , and its square A^2 . The elements of A and A^2 indicate the number of paths of lengths 1 and 2, respectively.

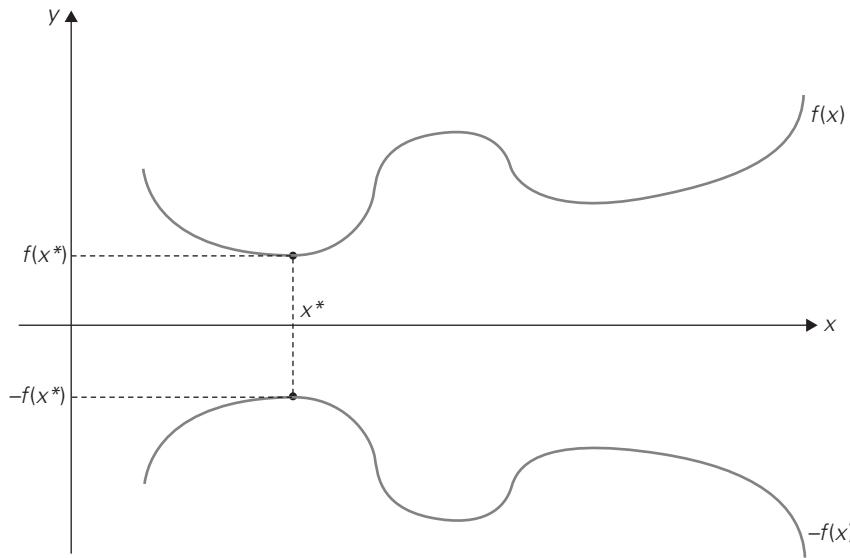


FIGURE 6.17 Relationship between minimization and maximization problems:
 $\min f(x) = -\max[-f(x)]$

Reduction of Optimization Problems

Our next example deals with solving optimization problems. If a problem seeks to find a maximum of some function, it is said to be a **maximization problem**; if it seeks to find a function's minimum, it is said to be a **minimization problem**. Suppose now that you have to find a minimum of some function $f(x)$ and you know an algorithm for maximizing the function. How can you take advantage of the latter? The answer lies in the simple formula

$$\min f(x) = -\max[-f(x)].$$

In other words, to minimize a function we can maximize its negative instead and, to get a correct minimal value of the function itself, change the sign of the answer. This property is illustrated for a function of one real variable in Figure 6.17.

Of course, the formula

$$\max f(x) = -\min[-f(x)]$$

is valid as well; it shows how a maximization problem can be reduced to an equivalent minimization problem.

This relationship between minimization and maximization problems is very general: it holds for functions defined on any domain D . In particular, we can

apply it to functions of several variables subject to additional constraints. A very important class of such problems is introduced in the next portion of this section.

Now that we are on the topic of function optimization, it is worth pointing out that the standard calculus procedure for finding extremum points of a function is, in fact, also based on problem reduction. Indeed, it suggests finding the function's derivative $f'(x)$ and then solving the equation $f'(x) = 0$ to find the function's critical points. In other words, the optimization problem is reduced to the problem of solving an equation as the principal part of finding extremum points. Note that we are not calling the calculus procedure an algorithm because it is not clearly defined. In fact, there is no general method for solving equations. A little secret of calculus textbooks is that problems are carefully selected so that critical points can always be found without difficulty. This makes the lives of both students and instructors easier but, in the process, may unintentionally create a wrong impression in students' minds.

Linear Programming

Many problems of optimal decision making can be reduced to an instance of the ***linear programming*** problem, which is a problem of optimizing a linear function of several variables subject to constraints in the form of linear equations and linear inequalities.

EXAMPLE 1 Consider a university endowment that needs to invest \$100 million. This sum must be split between three types of investments: stocks, bonds, and cash. The endowment managers expect an annual return of 10%, 7%, and 3% for their stock, bond, and cash investments, respectively. Since stocks are more risky than bonds, the endowment rules require an amount invested in stocks to be no more than one third of the moneys invested in bonds. In addition, at least 25% of the total amount invested in stocks and bonds must be invested in cash. How should the managers invest the money to maximize the return?

Let us create a mathematical model of this problem. Let x , y , and z be the amounts (in millions of dollars) invested in stocks, bonds, and cash, respectively. By using these variables, we can pose the following optimization problem:

$$\text{maximize } 0.10x + 0.07y + 0.03z$$

$$\text{subject to } x + y + z = 100$$

$$x \leq \frac{1}{3}y$$

$$z \geq 0.25(x + y)$$

$$x \geq 0, y \geq 0, z \geq 0.$$

Although this particular problem is both small and simple, it does show how a problem of optimal decision making can be reduced to an instance of the general linear programming problem

$$\begin{aligned} \text{maximize (or minimize)} \quad & c_1x_1 + \cdots + c_nx_n \\ \text{subject to} \quad & a_{i1}x_1 + \cdots + a_{in}x_n \leq (\text{or } \geq \text{ or } =) b_i \quad \text{for } i = 1, \dots, m \\ & x_1 \geq 0, \dots, x_n \geq 0. \end{aligned}$$

(The last group of constraints—the so-called nonnegativity constraints—are, strictly speaking, unnecessary because they are special cases of more general constraints $a_{i1}x_1 + \cdots + a_{in}x_n \geq b_i$, but it is convenient to treat them separately.)

Linear programming has proved to be flexible enough to model a wide variety of important applications, such as airline crew scheduling, transportation and communication network planning, oil exploration and refining, and industrial production optimization. In fact, linear programming is considered by many as one of the most important achievements in the history of applied mathematics. The classic algorithm for this problem is called the **simplex method**; it was discovered by the U.S. mathematician George Dantzig in the 1940s [Dan63]. Although the worst-case efficiency of this algorithm is known to be exponential, it performs very well on typical inputs. The efforts of many computer scientists over the last 50 years have polished the algorithm and its computer implementations to a point where problems with tens if not hundreds of thousands of variables and constraints can be solved in a reasonable amount of time. Moreover, a few other algorithms for solving the general linear programming problem have been discovered relatively recently; the best-known of them is by Narendra Karmarkar [Kar84]. The theoretical advantage of these newer algorithms lies in their proven polynomial worst-case efficiency; Karmarkar's algorithm has also been found to be competitive with the simplex method in empirical tests.

It is important to stress, however, that the simplex method and Karmarkar's algorithm can successfully handle only linear programming problems that do not limit its variables to integer values. When variables of a linear programming problem are required to be integers, the linear programming problem is said to be an **integer linear programming** problem. Integer linear programming problems are known to be much more difficult. There is no known polynomial-time algorithm for solving an arbitrary instance of the general integer linear programming problem and, as we shall see in Chapter 10, such an algorithm quite possibly does not exist. Other approaches such as the branch-and-bound technique discussed in Section 11.2 are typically used for solving integer linear programming problems.

EXAMPLE 2 Let us see how the knapsack problem can be reduced to a linear programming problem. Recall from Section 3.4 that the knapsack problem can be posed as follows. Given a knapsack of capacity W and n items of weights w_1, \dots, w_n and values v_1, \dots, v_n , find the most valuable subset of the items that fits into the knapsack. We consider first the so-called **continuous** or **fractional** version of the

problem in which any fraction of any item given can be taken into the knapsack. Let x_j , $j = 1, \dots, n$, be a variable representing a fraction of item j taken into the knapsack. Obviously, x_j must satisfy the inequality $0 \leq x_j \leq 1$. Then the total weight of the items selected can be expressed by the sums $\sum_{j=1}^n w_j x_j$ and the total value of the items selected by the sum $\sum_{j=1}^n v_j x_j$. Thus, the continuous version of the knapsack problem can be posed as the following linear programming problem:

$$\begin{aligned} & \text{maximize} && \sum_{j=1}^n v_j x_j \\ & \text{subject to} && \sum_{j=1}^n w_j x_j \leq W \\ & && 0 \leq x_j \leq 1 \text{ for } j = 1, \dots, n. \end{aligned}$$

There is no need to apply a general method for solving linear programming problems here: this particular problem can be solved by a simple special algorithm that will be introduced in Section 11.3. (But why wait? Try to discover it on your own now.) This reduction of the knapsack problem to an instance of the linear programming problem is still useful, though, to prove the correctness of the algorithm in question.

In the so called **discrete** or **0-1** version of the knapsack problem, we are allowed either to take an item in its entirety or not to take it at all. Hence, we have the following integer linear programming problem for this version:

$$\begin{aligned} & \text{maximize} && \sum_{j=1}^n v_j x_j \\ & \text{subject to} && \sum_{j=1}^n w_j x_j \leq W \\ & && x_j \in \{0, 1\} \text{ for } j = 1, \dots, n. \end{aligned}$$

This seemingly minor modification makes a drastic difference for the complexity of this and similar problems constrained to take only discrete values in their potential ranges. Despite the fact that the 0-1 version might seem to be easier because it can ignore any subset of the continuous version that has a fractional value of an item, the 0-1 version is, in fact, much more complicated than its continuous counterpart. The reader interested in specific algorithms for solving this problem will find a wealth of literature on the subject including the monograph by Martello and Toth [MT90].

Reduction to Graph Problems

As we mentioned in Section 1.3, many problems can be solved by a reduction to one of the standard graph problems. This is true in particular for a variety of puzzles and games. In these applications, vertices of a graph typically represent possible states of the problem in question while edges indicate permitted transitions among such states. One of the graph's vertices represents an initial state, while another represents a goal state of the problem. (There might be several vertices of the latter kind.) Such a graph is called a ***state-space graph***. Thus, the transformation just described reduces the problem to the question about a path from the initial-state vertex to a goal-state vertex.

EXAMPLE As a specific example, let us revisit the classic river-crossing puzzle of Problem 1 in the exercises to Section 1.2. A peasant finds himself on a river bank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river in his boat. However, the boat has room only for the peasant himself and one other item (either the wolf, the goat, or the cabbage). In his absence, the wolf would eat the goat, and the goat would eat the cabbage. Find a way for the peasant to solve his problem or prove that it has no solution.

The state-space graph for this problem is given in Figure 6.18. Its vertices are labeled to indicate the states they represent: P, w, g, c stand for the peasant, the wolf, the goat, and the cabbage, respectively; the two bars || denote the river; for convenience, we also label the edges by indicating the boat's occupants for

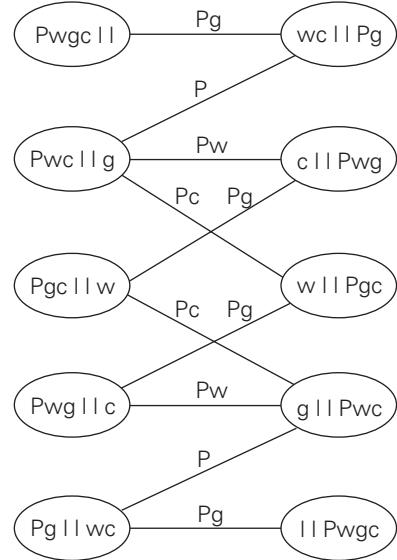


FIGURE 6.18 State-space graph for the peasant, wolf, goat, and cabbage puzzle

each crossing. In terms of this graph, we are interested in finding a path from the initial-state vertex labeled $\text{Pwgc} \parallel$ to the final-state vertex labeled $\parallel \text{Pwgc}$.

It is easy to see that there exist two distinct simple paths from the initial-state vertex to the final state vertex (what are they?). If we find them by applying breadth-first search, we get a formal proof that these paths have the smallest number of edges possible. Hence, this puzzle has two solutions, each of which requires seven river crossings. ■

Our success in solving this simple puzzle should not lead you to believe that generating and investigating state-space graphs is always a straightforward task. To get a better appreciation of state-space graphs, consult books on artificial intelligence (AI), the branch of computer science in which such problems are a principal subject. In this book, we will deal with an important special case of state-space graphs in Sections 11.1 and 11.2.

Exercises 6.6

Exercises 6.6

- 1. a.** Prove the equality

$$\text{lcm}(m, n) = \frac{m \cdot n}{\text{gcd}(m, n)}$$

that underlies the algorithm for computing $\text{lcm}(m, n)$.

- b.** Euclid's algorithm is known to be in $O(\log n)$. If it is the algorithm that is used for computing $\text{gcd}(m, n)$, what is the efficiency of the algorithm for computing $\text{lcm}(m, n)$?
- 2.** You are given a list of numbers for which you need to construct a min-heap. (A min-heap is a complete binary tree in which every key is less than or equal to the keys in its children.) How would you use an algorithm for constructing a max-heap (a heap as defined in Section 6.4) to construct a min-heap?
- 3.** Prove that the number of different paths of length $k > 0$ from the i th vertex to the j th vertex of a graph (undirected or directed) equals the (i, j) th element of A^k where A is the adjacency matrix of the graph.
- 4. a.** Design an algorithm with a time efficiency better than cubic for checking whether a graph with n vertices contains a cycle of length 3 [Man89].
- b.** Consider the following algorithm for the same problem. Starting at an arbitrary vertex, traverse the graph by depth-first search and check whether its depth-first search forest has a vertex with a back edge leading to its grandparent. If it does, the graph contains a triangle; if it does not, the graph does not contain a triangle as its subgraph. Is this algorithm correct?
- 5.** Given $n > 3$ points $P_1 = (x_1, y_1), \dots, P_n = (x_n, y_n)$ in the coordinate plane, design an algorithm to check whether all the points lie within a triangle with

its vertices at three of the points given. (You can either design an algorithm from scratch or reduce the problem to another one with a known algorithm.)

6. Consider the problem of finding, for a given positive integer n , the pair of integers whose sum is n and whose product is as large as possible. Design an efficient algorithm for this problem and indicate its efficiency class.
7. The assignment problem, introduced in Section 3.4, can be stated as follows. There are n people who need to be assigned to execute n jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the i th person is assigned to the j th job is a known quantity $C[i, j]$ for each pair $i, j = 1, \dots, n$. The problem is to assign the people to the jobs to minimize the total cost of the assignment. Express the assignment problem as a 0-1 linear programming problem.

8. Solve the instance of the linear programming problem given in Section 6.6

$$\begin{aligned} & \text{maximize} && 0.10x + 0.07y + 0.03z \\ & \text{subject to} && x + y + z = 100 \\ & && x \leq \frac{1}{3}y \\ & && z \geq 0.25(x + y) \\ & && x \geq 0, y \geq 0, z \geq 0. \end{aligned}$$

Exercises 6.6

9. The graph-coloring problem is usually stated as the vertex-coloring problem: assign the smallest number of colors to vertices of a given graph so that no two adjacent vertices are the same color. Consider the **edge-coloring** problem: assign the smallest number of colors possible to edges of a given graph so that no two edges with the same end point are the same color. Explain how the edge-coloring problem can be reduced to a vertex-coloring problem.



10. **Jealous-Husbands Puzzle** There are n ($n \geq 2$) married couples who need to cross a river. They have a boat that can hold no more than two people at a time. To complicate matters, all the husbands are jealous and will not agree on any crossing procedure that would put a wife on the same bank of the river with another woman's husband without the wife's husband being there too, even if there are other people on the same bank. Can they cross the river under such constraints?
 - a. Solve the problem for $n = 2$.
 - b. Solve the problem for $n = 3$, which is the classical version of this problem.
 - c. Does the problem have a solution for every $n \geq 4$? If it does, indicate how many river crossings it will take; if it does not, explain why.

SUMMARY

- *Transform-and-conquer* is the fourth general algorithm design (and problem-solving) strategy discussed in the book. It is, in fact, a group of techniques based on the idea of transformation to a problem that is easier to solve.
- There are three principal varieties of the transform-and-conquer strategy: *instance simplification*, *representation change*, and *problem reduction*.
- *Instance simplification* is a technique of transforming an instance of a problem to an instance of the same problem with some special property that makes the problem easier to solve. List presorting, Gaussian elimination, and AVL trees are good examples of this technique.
- *Representation change* implies changing one representation of a problem's instance into another representation of the same instance. Examples discussed in this chapter include representation of a set by a 2-3 tree, heaps for representing priority queues, Horner's rule for polynomial evaluation, and two binary exponentiation algorithms.
- *Problem reduction* calls for transforming a problem given to another problem that can be solved by a known algorithm. Among examples of applying this idea to algorithmic problem solving (see Section 6.6), reductions to linear programming and reductions to graph problems are especially important.
- Some examples used to illustrate the transform-and-conquer techniques happen to be very important data structures and algorithms. They are: heaps and heapsort, AVL and 2-3 trees, Gaussian elimination, and Horner's rule.
- A *heap* is an essentially complete binary tree with keys (one per node) satisfying the parental dominance requirement. Though defined as binary trees, heaps are normally implemented as arrays. Heaps are most important for the efficient implementation of priority queues; they also underlie heapsort.
- *Heapsort* is a theoretically important sorting algorithm based on arranging elements of an array in a heap and then successively removing the largest element from a remaining heap. The algorithm's running time is in $\Theta(n \log n)$ both in the worst case and in the average case; in addition, it is in place.
- *AVL trees* are binary search trees that are always balanced to the extent possible for a binary tree. The balance is maintained by transformations of four types called *rotations*. All basic operations on AVL trees are in $\Theta(\log n)$; it eliminates the bad worst-case efficiency of classic binary search trees.

- 2-3 trees achieve a perfect balance in a search tree by allowing a node to contain up to two ordered keys and have up to three children. This idea can be generalized to yield very important *B*-trees, discussed later in the book.
- *Gaussian elimination*—an algorithm for solving systems of linear equation—is a principal algorithm in linear algebra. It solves a system by transforming it to an equivalent system with an upper-triangular coefficient matrix, which is easy to solve by backward substitutions. Gaussian elimination requires about $\frac{1}{3}n^3$ multiplications.
- *Horner's rule* is an optimal algorithm for polynomial evaluation without coefficient preprocessing. It requires only n multiplications and n additions. It also has a few useful by-products such as the synthetic division algorithm.
- Two *binary exponentiation* algorithms for computing a^n were introduced in Section 6.5. Both of them exploit the binary representation of the exponent n , but they process it in the opposite directions: left to right and right to left.
- *Linear programming* concerns optimizing a linear function of several variables subject to constraints in the form of linear equations and linear inequalities. There are efficient algorithms capable of solving very large instances of this problem with many thousands of variables and constraints, provided the variables are not required to be integers. The latter, called *integer linear programming* problems, constitute a much more difficult class of problems.

Space and Time Tradeoffs

Things which matter most must never be at the mercy of things which matter less.

—Johann Wolfgang von Goethe (1749–1832)

Space and time tradeoffs in algorithm design are a well-known issue for both theoreticians and practitioners of computing. Consider, as an example, the problem of computing values of a function at many points in its domain. If it is time that is at a premium, we can precompute the function's values and store them in a table. This is exactly what human computers had to do before the advent of electronic computers, in the process burdening libraries with thick volumes of mathematical tables. Though such tables have lost much of their appeal with the widespread use of electronic computers, the underlying idea has proven to be quite useful in the development of several important algorithms for other problems. In somewhat more general terms, the idea is to preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward. We call this approach *input enhancement*¹ and discuss the following algorithms based on it:

- counting methods for sorting (Section 7.1)
- Boyer-Moore algorithm for string matching and its simplified version suggested by Horspool (Section 7.2)

The other type of technique that exploits space-for-time tradeoffs simply uses extra space to facilitate faster and/or more flexible access to the data. We call this approach *prestructuring*. This name highlights two facets of this variation of the space-for-time tradeoff: some processing is done before a problem in question

1. The standard terms used synonymously for this technique are *preprocessing* and *preconditioning*. Confusingly, these terms can also be applied to methods that use the idea of preprocessing but do not use extra space (see Chapter 6). Thus, in order to avoid confusion, we use “input enhancement” as a special name for the space-for-time tradeoff technique being discussed here.

is actually solved but, unlike the input-enhancement variety, it deals with access structuring. We illustrate this approach by

- hashing (Section 7.3)
- indexing with B-trees (Section 7.4).

There is one more algorithm design technique related to the space-for-time tradeoff idea: **dynamic programming**. This strategy is based on recording solutions to overlapping subproblems of a given problem in a table from which a solution to the problem in question is then obtained. We discuss this well-developed technique separately, in the next chapter of the book.

Two final comments about the interplay between time and space in algorithm design need to be made. First, the two resources—time and space—do not have to compete with each other in all design situations. In fact, they can align to bring an algorithmic solution that minimizes both the running time and the space consumed. Such a situation arises, in particular, when an algorithm uses a space-efficient data structure to represent a problem's input, which leads, in turn, to a faster algorithm. Consider, as an example, the problem of traversing graphs. Recall that the time efficiency of the two principal traversal algorithms—depth-first search and breadth-first search—depends on the data structure used for representing graphs: it is $\Theta(n^2)$ for the adjacency matrix representation and $\Theta(n + m)$ for the adjacency list representation, where n and m are the numbers of vertices and edges, respectively. If input graphs are sparse, i.e., have few edges relative to the number of vertices (say, $m \in O(n)$), the adjacency list representation may well be more efficient from both the space and the running-time points of view. The same situation arises in the manipulation of sparse matrices and sparse polynomials: if the percentage of zeros in such objects is sufficiently high, we can save both space and time by ignoring zeros in the objects' representation and processing.

Second, one cannot discuss space-time tradeoffs without mentioning the hugely important area of data compression. Note, however, that in data compression, size reduction is the goal rather than a technique for solving another problem. We discuss just one data compression algorithm, in the next chapter. The reader interested in this topic will find a wealth of algorithms in such books as [Say00].

7.1 Sorting by Counting

As a first example of applying the input enhancement technique, we discuss its application to the sorting problem. One rather obvious idea is to count, for each element of a list to be sorted, the total number of elements smaller than this element and record the results in a table. These numbers will indicate the positions of the elements in the sorted list: e.g., if the count is 10 for some element, it should be in the 11th position (with index 10, if we start counting with 0) in the sorted array. Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list. This algorithm is called **comparison counting sort** (Figure 7.1).

Array A[0..5]		62	31	84	96	19	47
Initially	Count []	0	0	0	0	0	0
After pass $i = 0$	Count []	3	0	1	1	0	0
After pass $i = 1$	Count []		1	2	2	0	1
After pass $i = 2$	Count []			4	3	0	1
After pass $i = 3$	Count []				5	0	1
After pass $i = 4$	Count []					0	2
Final state	Count []	3	1	4	5	0	2
Array S[0..5]		19	31	47	62	84	96

FIGURE 7.1 Example of sorting by comparison counting**ALGORITHM** *ComparisonCountingSort(A[0..n - 1])*

```

//Sorts an array by comparison counting
//Input: An array A[0..n - 1] of orderable elements
//Output: Array S[0..n - 1] of A's elements sorted in nondecreasing order
for  $i \leftarrow 0$  to  $n - 1$  do  $Count[i] \leftarrow 0$ 
for  $i \leftarrow 0$  to  $n - 2$  do
    for  $j \leftarrow i + 1$  to  $n - 1$  do
        if  $A[i] < A[j]$ 
             $Count[j] \leftarrow Count[j] + 1$ 
        else  $Count[i] \leftarrow Count[i] + 1$ 
for  $i \leftarrow 0$  to  $n - 1$  do  $S[Count[i]] \leftarrow A[i]$ 
return S

```

What is the time efficiency of this algorithm? It should be quadratic because the algorithm considers all the different pairs of an n -element array. More formally, the number of times its basic operation, the comparison $A[i] < A[j]$, is executed is equal to the sum we have encountered several times already:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}.$$

Since the algorithm makes the same number of key comparisons as selection sort and in addition uses a linear amount of extra space, it can hardly be recommended for practical use.

But the counting idea does work productively in a situation in which elements to be sorted belong to a known small set of values. Assume, for example, that we have to sort a list whose values can be either 1 or 2. Rather than applying a general sorting algorithm, we should be able to take advantage of this additional information about values to be sorted. Indeed, we can scan the list to compute the number of 1's and the number of 2's in it and then, on the second pass, simply make the appropriate number of the first elements equal to 1 and the remaining elements equal to 2. More generally, if element values are integers between some

lower bound l and upper bound u , we can compute the frequency of each of those values and store them in array $F[0..u-l]$. Then the first $F[0]$ positions in the sorted list must be filled with l , the next $F[1]$ positions with $l+1$, and so on. All this can be done, of course, only if we can overwrite the given elements.

Let us consider a more realistic situation of sorting a list of items with some other information associated with their keys so that we cannot overwrite the list's elements. Then we can copy elements into a new array $S[0..n-1]$ to hold the sorted list as follows. The elements of A whose values are equal to the lowest possible value l are copied into the first $F[0]$ elements of S , i.e., positions 0 through $F[0]-1$, the elements of value $l+1$ are copied to positions from $F[0]$ to $(F[0]+F[1])-1$, and so on. Since such accumulated sums of frequencies are called a distribution in statistics, the method itself is known as ***distribution counting***.

EXAMPLE Consider sorting the array

13	11	12	13	12	12
----	----	----	----	----	----

whose values are known to come from the set {11, 12, 13} and should not be overwritten in the process of sorting. The frequency and distribution arrays are as follows:

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

Note that the distribution values indicate the proper positions for the last occurrences of their elements in the final sorted array. If we index array positions from 0 to $n-1$, the distribution values must be reduced by 1 to get corresponding element positions.

It is more convenient to process the input array right to left. For the example, the last element is 12, and, since its distribution value is 4, we place this 12 in position $4-1=3$ of the array S that will hold the sorted list. Then we decrease the 12's distribution value by 1 and proceed to the next (from the right) element in the given array. The entire processing of this example is depicted in Figure 7.2. ■

Here is a pseudocode of this algorithm.

ALGORITHM *DistributionCounting(A[0..n-1], l, u)*

```
//Sorts an array of integers from a limited range by distribution counting
//Input: An array A[0..n-1] of integers between l and u (l ≤ u)
//Output: Array S[0..n-1] of A's elements sorted in nondecreasing order
for j ← 0 to u - l do D[j] ← 0 //initialize frequencies
```

	D[0..2]			S[0..5]		
A [5] = 12	1	4	6		12	
A [4] = 12	1	3	6			13
A [3] = 13	1	2	6			
A [2] = 12	1	2	5	12		
A [1] = 11	1	1	5	11		
A [0] = 13	0	1	5			13

FIGURE 7.2 Example of sorting by distribution counting. The distribution values being decremented are shown in bold.

```

for  $i \leftarrow 0$  to  $n - 1$  do  $D[A[i] - l] \leftarrow D[A[i] - l] + 1$  //compute frequencies
for  $j \leftarrow 1$  to  $u - l$  do  $D[j] \leftarrow D[j - 1] + D[j]$  //reuse for distribution
for  $i \leftarrow n - 1$  downto 0 do
     $j \leftarrow A[i] - l$ 
     $S[D[j] - 1] \leftarrow A[i]$ 
     $D[j] \leftarrow D[j] - 1$ 
return  $S$ 

```

Assuming that the range of array values is fixed, this is obviously a linear algorithm because it makes just two consecutive passes through its input array A . This is a better time-efficiency class than that of the most efficient sorting algorithms—mergesort, quicksort, and heapsort—we have encountered. It is important to remember, however, that this efficiency is obtained by exploiting the specific nature of input lists on which sorting by distribution counting works, in addition to trading space for time.

Exercises 7.1

1. Write a pseudocode to exchange values of two variables, say u and v , with and without using extra storage.
2. Will the comparison counting algorithm work correctly for arrays with equal values?
3. Assuming that the set of possible list values is $\{a, b, c, d\}$, sort the following list in alphabetical order by the distribution counting algorithm:

$b, c, d, c, b, a, a, b.$

4. Is the distribution counting algorithm stable?
5. Design an algorithm for the multiplying corresponding numbers from two arrays of size n , whose values are n distinct integers from 1 to n .
6. The *ancestry problem* asks to determine whether a vertex u is an ancestor of vertex v in a given binary (or, more generally, rooted ordered) tree of n vertices. Design a $O(n)$ input enhancement algorithm that provides sufficient

information to solve this problem for any pair of the tree's vertices in constant time.

7. The following technique, known as *virtual initialization*, provides a time-efficient way to initialize just some elements of a given array $A[0..n - 1]$ so that for each of its elements, we can say in constant time whether it has been initialized and, if it has been, with which value. This is done by utilizing a variable *counter* for the number of initialized elements in A and two auxiliary arrays of the same size, say $B[0..n - 1]$ and $C[0..n - 1]$, defined as follows. $B[0], \dots, B[counter - 1]$ contain the indices of the elements of A that were initialized: $B[0]$ contains the index of the element initialized first, $B[1]$ contains the index of the element initialized second, and so on. Furthermore, if $A[i]$ was the k th element ($0 \leq k \leq counter - 1$) to be initialized, $C[i]$ contains k .
 - a. Sketch the state of arrays $A[0..7]$, $B[0..7]$, and $C[0..7]$ after the three assignments

$$A[3] \leftarrow x; A[7] \leftarrow z; A[1] \leftarrow y.$$

- b. In general, how can we check with this scheme whether $A[i]$ has been initialized and, if it has been, with which value?
8. a. Write a program for the addition of two sparse matrices, a p -by- q matrix A and a p -by- q matrix B .
 b. Write a pseudocode for multiplication of two sparse matrices, a p -by- q matrix A and a p -by- q matrix B .
 c. Write a program for multiplication and addition of two polynomials $p(x)$ and $q(x)$ of degrees m and n , respectively.
9. *Tic-tac-toe* Write a program that plays the game of tic-tac-toe with the human user by storing all possible positions on the game's 3-by-3 board along with the best move for each of them.



7.2 Input Enhancement in String Matching

In this section, we see how the technique of input enhancement can be applied to the problem of string matching. Recall that the problem of string matching requires finding an occurrence of a given string of m characters (called the *pattern*) in a longer string of n characters (called the *text*). We discussed the brute-force algorithm for this problem in Section 3.2: it simply matches corresponding pairs of characters in the pattern and the text left to right and, if a mismatch occurs, shifts the pattern one position to the right for the next trial. Since the maximum number of such trials is $n - m + 1$ and, in the worst case, m comparisons need to be made on each of them, the worst-case number of character comparisons is $m(n - m + 1)$. This puts the worst-case performance of the brute-force algorithm in the $\Theta(nm)$ class. On average, however, we should expect just a few comparisons before a pattern's shift, and for random natural-language texts, the average-case efficiency indeed turns out to be in $\Theta(n)$.

The good average-case efficiency of the brute-force algorithm can be considered both good and bad news. It is good news from a practical standpoint because it makes the brute-force solution a passable candidate for practical applications (especially for short patterns). It is bad news for a theoretician who would like to invent a faster algorithm. Nevertheless, several better algorithms have been discovered. Most of them exploit the input enhancement idea: preprocess the pattern to get some information about it, store this information in a table, and then use this information during an actual search for the pattern in a given text. This is exactly the idea behind the two best-known algorithms of this type: the Knuth-Morris-Pratt algorithm [Knu77] and the Boyer-Moore algorithm [Boy77].

The principal difference between these two algorithms lies in the way they compare characters of a pattern with their counterparts in a text: the Knuth-Morris-Pratt algorithm does it left to right, while the Boyer-Moore algorithm does it right to left. Since the latter idea leads to simpler algorithms, it is the only one we will pursue here. (Note that the Boyer-Moore algorithm starts by aligning the pattern against the beginning characters of the text; if the first trial fails, it shifts the pattern to the right. It is comparisons within a trial that the algorithm does right to left, starting with the last one in the pattern.)

Although the underlying idea of the Boyer-Moore algorithm is simple, its actual implementation in a working method is less so. Therefore, we start our discussion with a simplified version of the Boyer-Moore algorithm suggested by R. Horspool [Hor80]. In addition to being simpler, Horspool's algorithm is not necessarily less efficient than the Boyer-Moore algorithm on random strings.

Horspool's Algorithm

Consider, as an example, searching for the pattern BARBER in some text:

$s_0 \dots$	$c \dots s_{n-1}$
B A R B E R	

Starting with the last R of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and the text. If all the pattern's characters match successfully, a matching substring is found. (Then the search can be either stopped altogether or continued if another occurrence of the same pattern is desired.) If, however, we encounter a mismatch, we need to shift the pattern to the right. Clearly, we would like to make as large a shift as possible without risking the possibility of missing a matching substring in the text. Horspool's algorithm determines the size of such a shift by looking at the character c of the text that was aligned against the last character of the pattern. In general, the following four possibilities can occur.

Case 1 If there are no c 's in the pattern—e.g., c is letter S in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character c that is known not to be in the pattern):

must be 1. (This condition is satisfied automatically if m itself is prime.) Some functions recommended in the literature are $s(k) = m - 2 - k \bmod (m - 2)$ and $s(k) = 8 - (k \bmod 8)$ for small tables and $s(k) = k \bmod 97 + 1$ for larger ones (see [Sed88], [Sed98]). Mathematical analysis of double hashing has proved to be quite difficult. Some partial results and considerable practical experience with the method suggest that with good hashing functions—both primary and secondary—double hashing is superior to linear probing. But its performance also deteriorates when the table gets close to being full. A natural solution in such a situation is *rehashing*: the current table is scanned, and all its keys are relocated into a larger table.

Since its discovery in the 1950s by IBM researchers, hashing has found many important applications. In particular, it has become a standard technique for storing a symbol table—a table of a computer program's symbols generated during compilation. With some modifications, it has also proved to be useful for storing very large dictionaries on disks; this variation of hashing is called *extendible hashing*. Since disk access is expensive compared with probes performed in the main memory, it is preferable to make many more probes than disk accesses. Accordingly, a location computed by a hash function in extendible hashing indicates a disk address of a *bucket* that can hold up to b keys. When a key's bucket is identified, all its keys are read into main memory and then searched for the key in question. In the next section, we discuss *B-trees*, a principal alternative for storing large dictionaries.

Exercises 7.3

1. For the inputs 50, 10, 59, 75, 21, 18 and hash function $h(k) = k \bmod 11$
 - a. Construct the open and closed hash table.
 - b. Find the largest number of key comparisons in a successful search for both tables.
2. For the input 30, 20, 56, 75, 31, 19 and hash function $h(K) = K \bmod 11$
 - a. construct the closed hash table.
 - b. find the largest number of key comparisons in a successful search in this table.
 - c. find the average number of key comparisons in a successful search in this table.
3. Write a pseudocode using the hashing method to implement the dictionary, *i.e.*, a set with the operations:
 - find
 - insert
 - delete
4. Consider the input as student records, key = SSN and hash function $h(K) = K \bmod m$, where m is an integer (typically, prime). If $m = 1000$, where is a record with SSN = 314159265 stored?



5. *Birthday paradox* The birthday paradox asks how many people should be in a room so that the chances are better than even that two of them will have the same birthday (month and day). Find the quite unexpected answer to this problem. What implication for hashing does this result have?
6. Answer the following questions for the separate-chaining version of hashing.
 - a. Where would you insert keys if you knew that all the keys in the dictionary are distinct? Which dictionary operations, if any, would benefit from this modification?
 - b. We could keep keys of the same linked list sorted. Which of the dictionary operations would benefit from this modification? How could we take advantage of this if all the keys stored in the entire table need to be sorted?
7. Explain how hashing can be applied to check whether all elements of a list are distinct. What is the time efficiency of this application?
8. Fill in the following table with the average-case efficiency classes for the five implementations of the ADT dictionary:

	unordered array	ordered array	binary search tree	separate chaining	linear probing
search					
insertion					
deletion					

9. We have discussed hashing in the context of techniques based on space-time tradeoffs. But it also takes advantage of another general strategy. Which one?
10. Write a computer program that uses hashing for the following problem. Given a natural language text, generate a list of distinct words with the number of occurrences of each word in the text. Insert appropriate counters in the program to compare the empirical efficiency of hashing with the corresponding theoretical results.

7.4 B-Trees

The idea of using extra space to facilitate faster access to a given data set is particularly important if the data set in question contains a very large number of records that need to be stored on a disk. A principal device in organizing such data sets is an *index*, which provides some information about the location of records with indicated key values. For data sets of structured records (as opposed to “unstruc-

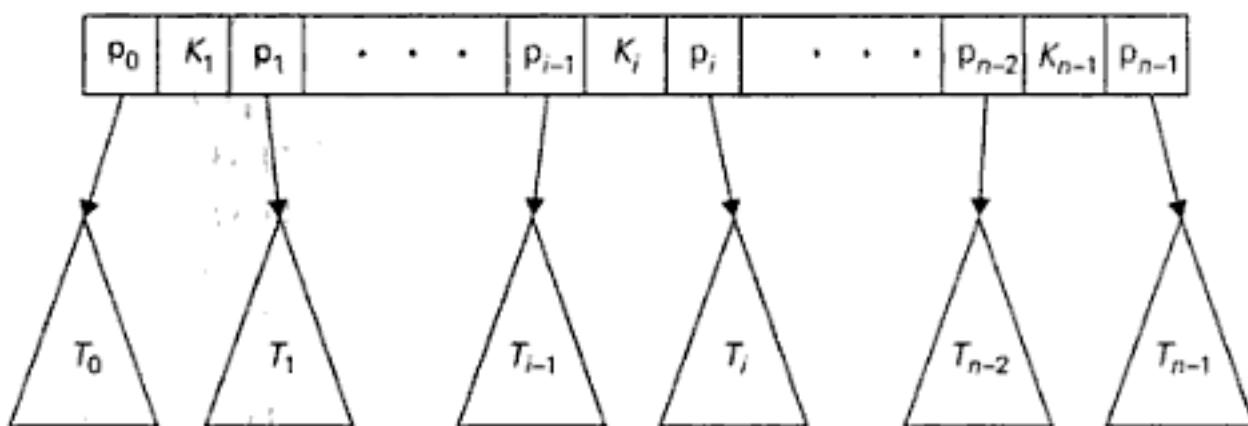


FIGURE 7.7 Parental node of a B-tree

tured” data such as text, images, sound, and video), the most important index organization is the **B-tree**, introduced by R. Bayer and E. McCreight [Bay72]. It extends the idea of the 2-3 tree (see Section 6.3) by permitting more than a single key in the same node of a search tree.

In the B-tree version we consider here, all data records (or record keys) are stored at the leaves, in increasing order of the keys. The parental nodes are used for indexing. Specifically, each parental node contains $n - 1$ ordered keys $K_1 < \dots < K_{n-1}$ assumed, for the sake of simplicity, to be distinct. The keys are interposed with n pointers to the node’s children so that all the keys in subtree T_0 are smaller than K_1 , all the keys in subtree T_1 are greater than or equal to K_1 and smaller than K_2 with K_1 being equal to the smallest key in T_1 , and so on through the last subtree T_{n-1} whose keys are greater than or equal to K_{n-1} with K_{n-1} being equal to the smallest key in T_{n-1} (Figure 7.7). The node depicted in Figure 7.7 is called the **n-node**. Thus, all the nodes in a classic binary search tree are 2-nodes; a 2-3 tree introduced in Section 6.3 comprises 2-nodes and 3-nodes.

In addition, a B-tree of order $m \geq 2$ must satisfy the following structural properties.

- The root is either a leaf or has between 2 and m children.
- Each node, except for the root and the leaves, has between $\lceil m/2 \rceil$ and m children (and hence between $\lceil m/2 \rceil - 1$ and $m - 1$ keys).
- The tree is (perfectly) balanced; i.e., all its leaves are at the same level.

An example of a B-tree of order 4 is given in Figure 7.8.

Searching in a B-tree is very similar to searching in the binary search tree, and even more so in the 2-3 tree. Starting at the root, we follow a chain of pointers to the leaf that may contain the search key. Then we search for the search key among the keys of that leaf. Note that since keys are stored in sorted order, at both parental nodes and leaves, we can use binary search if the number of keys at a node is large enough to make it worthwhile.

It is not the number of key comparisons, however, that we should be concerned about in a typical application of this data structure. When used for storing a large data file on a disk, the nodes of a B-tree normally correspond to the disk pages.

and a few typical values of the tree's order m :

order m	50	100	250
h 's upper bound	6	5	4

Keep in mind that the table's entries are upper estimates for the number of disk accesses. In actual applications, this number rarely exceeds 3, with the B-tree's root and sometimes first-level nodes stored in the fast memory to minimize the number of disk accesses.

The operations of insertion and deletion are less straightforward than searching, but both can also be done in $O(\log n)$ time. Here we outline an insertion algorithm only; a deletion algorithm can be found in the references (e.g., [Aho83], [Cor01]).

The most straightforward algorithm for inserting a new record into a B-tree is quite similar to the algorithm for insertion into a 2-3 tree outlined in Section 6.3. First, we apply the search procedure to the new record's key K to find the appropriate leaf for the new record. If there is room for the record in that leaf, we place it there (in an appropriate position so that the keys remain sorted) and we are done. If there is no room for the record, the leaf is split in half by sending the second half of the records to a new node. After that, the smallest key K' in the new node and the pointer to it have to be inserted into the old leaf's parent (immediately after the key and pointer to the old leaf). This recursive procedure may percolate up to the tree's root. If the root is already full too, a new root is created with the two halves of the old root's keys split between two children of the new root. As an example, Figure 7.9 shows the result of inserting 65 into the B-tree in Figure 7.8 under the restriction that the tree's leaves cannot contain more than three items.

You should be aware that there are other algorithms for implementing insertions into a B-tree. For example, we can avoid the possibility of recursive node splits if we split full nodes as we are searching for an appropriate leaf for the new record. Another possibility is to avoid some node splits by moving a key to the node's sibling. For example, inserting 65 into the B-tree in Figure 7.8 can be done by moving 60, the smallest key of the full leaf, to its sibling with keys 51 and 55, and

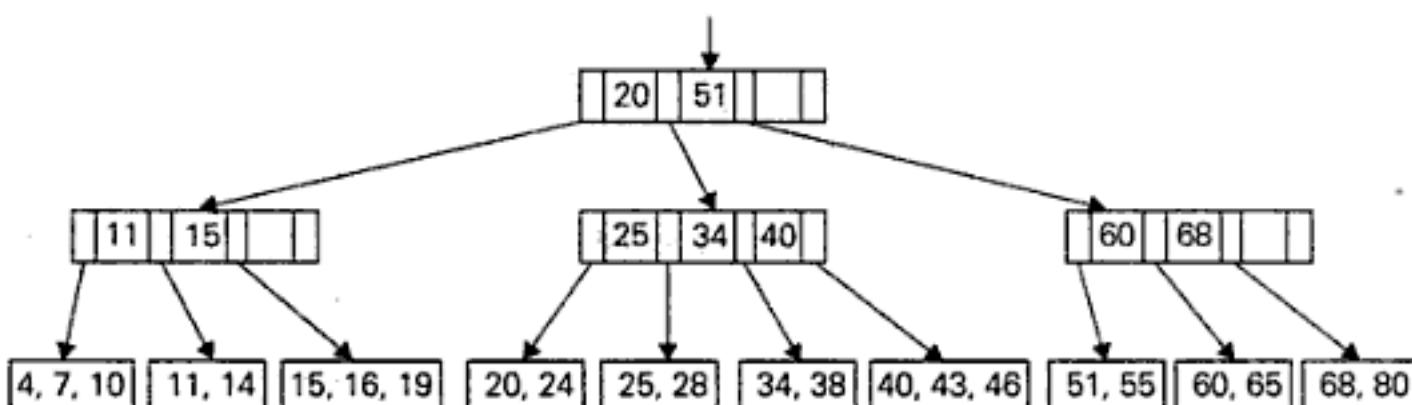


FIGURE 7.9 B-tree obtained after inserting 65 into the B-tree in Figure 7.8

replacing the key value of their parent by 65, the new smallest value in the second child. This modification tends to save some space at the expense of a slightly more complicated algorithm.

A B-tree does not have to be always associated with the indexing of a large file, and it can be considered as one of several varieties of the search tree. As with other types of search trees—such as binary search trees, AVL trees, and 2-3 trees—a B-tree can be constructed by successive insertions of data records into the initially empty tree. (The empty tree is considered to be a B-tree, too.) When all keys reside in the leaves and the upper levels are organized as a B-tree comprising an index, the entire structure is usually called, in fact, a *B⁺-tree*.

Exercises 7.4

1. Give examples of using an index in real-life applications that do not involve computers.
2. a. Prove the equality

$$1 + \sum_{i=1}^{h-1} 2\lceil m/2 \rceil^{i-1} (\lceil m/2 \rceil - 1) + 2\lceil m/2 \rceil^{h-1} = 4\lceil m/2 \rceil^{h-1} - 1,$$

which was used in the derivation of upper bound (7.7) for the height of a B-tree.

- b. Complete the derivation of inequality (7.7).
3. Find the minimum order of the B-tree that guarantees that the number of disk accesses in searching in a file of 100 million records does not exceed 3. Assume that the root's page is stored in main memory.
4. Explain B-tree of order 4 with an example and then show the insertion of keys into the tree.
5. Write an algorithm for the second largest key and the smallest key in a B-tree.
6. a. A *top-down 2-3-4 tree* is a B-tree of order 4 with the following modification of the *insert* operation. Whenever a search for a leaf for a new key encounters a full node (i.e., a node with three keys), the node is split into two nodes by sending its middle key to the node's parent (or, if the full node happens to be the root, the new root for the middle key is created). Construct a top-down 2-3-4 tree by inserting the following list of keys in the initially empty tree:

10, 6, 15, 31, 20, 27, 50, 44, 18.

- b. What is the principal advantage of this insertion procedure compared with the one used for 2-3 trees in Section 6.3? What is its disadvantage?
 7. Write a pseudocode for implementing a key into the B-tree.
-

8

Dynamic Programming

An idea, like a ghost... must be spoken to a little before it will explain itself.

—Charles Dickens (1812–1870)

Dynamic programming is an algorithm design technique with a rather interesting history. It was invented by a prominent U.S. mathematician, Richard Bellman, in the 1950s as a general method for optimizing multistage decision processes. Thus, the word “programming” in the name of this technique stands for “planning” and does not refer to computer programming. After proving its worth as an important tool of applied mathematics, dynamic programming has eventually come to be considered, at least in computer science circles, as a general algorithm design technique that does not have to be limited to special types of optimization problems. It is from this point of view that we will consider this technique here.

Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which we can then obtain a solution to the original problem.

This technique can be illustrated by revisiting the Fibonacci numbers discussed in Section 2.5. (If you have not read that section, you will be able to follow the discussion anyway. But it is a beautiful topic, so if you feel a temptation to read it, do succumb to it.) The Fibonacci numbers are the elements of the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots,$$

which can be defined by the simple recurrence

$$F(n) = F(n - 1) + F(n - 2) \quad \text{for } n \geq 2 \tag{8.1}$$

and two initial conditions

$$F(0) = 0, F(1) = 1. \quad (8.2)$$

If we try to use recurrence (8.1) directly to compute the n th Fibonacci number $F(n)$, we would have to recompute the same values of this function many times (see Figure 2.6 for a specific example). Note that the problem of computing $F(n)$ is expressed in terms of its smaller and overlapping subproblems of computing $F(n - 1)$ and $F(n - 2)$. So we can simply fill elements of a one-dimensional array with the $n + 1$ consecutive values of $F(n)$ by starting, in view of initial conditions (8.2), with 0 and 1 and using equation (8.1) as the rule for producing all the other elements. Obviously, the last element of this array will contain $F(n)$. A single-loop pseudocode for this very simple algorithm can be found in Section 2.5.

Note that we can, in fact, avoid using an extra array to accomplish this task by recording the values of just the last two elements of the Fibonacci sequence (see Problem 6 in Exercises 2.5). This phenomenon is not unusual, and we shall encounter it in a few more examples in this chapter. Thus, although a straightforward application of dynamic programming can be interpreted as a special variety of space-for-time tradeoff, a dynamic programming algorithm can sometimes be refined to avoid using extra space.

Certain algorithms compute the n th Fibonacci number without computing all the preceding elements of this sequence (see Section 2.5). It is typical of an algorithm based on the classic bottom-up dynamic programming approach, however, to solve *all* smaller subproblems of a given problem. One variation of the dynamic programming approach seeks to avoid solving unnecessary subproblems. This technique, illustrated in Section 8.4, exploits so-called memory functions and can be considered a top-down variation of dynamic programming.

Whether we use the classical bottom-up version of dynamic programming or its top-down variation, the crucial step in designing such an algorithm remains the same: namely, deriving a recurrence relating a solution to the problem's instance to solutions of its smaller (and overlapping) subinstances. The immediate availability of equation (8.1) for computing the n th Fibonacci number is one of the few exceptions to this rule.

In the sections and exercises of this chapter are a few standard examples of dynamic programming algorithms. (Some of them, in fact, were invented before or independent of the discovery of dynamic programming and only later came to be viewed as examples of this technique's applications.) Numerous other applications range from the optimal way of breaking text into lines (e.g., [Baa00]) to an optimal triangulation of a polygon (e.g., [Ski98]) to a variety of applications to sophisticated engineering problems (e.g., [Bel62], [Ber01]).

8.1 Computing a Binomial Coefficient

Computing a binomial coefficient is a standard example of applying dynamic programming to a nonoptimization problem. You may recall from your studies of elementary combinatorics that the **binomial coefficient**, denoted $C(n, k)$ or $\binom{n}{k}$,

ALGORITHM *Binomial(n, k)*

```

//Computes  $C(n, k)$  by the dynamic programming algorithm
//Input: A pair of nonnegative integers  $n \geq k \geq 0$ 
//Output: The value of  $C(n, k)$ 
for  $i \leftarrow 0$  to  $n$  do
    for  $j \leftarrow 0$  to  $\min(i, k)$  do
        if  $j = 0$  or  $j = i$ 
             $C[i, j] \leftarrow 1$ 
        else  $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$ 
return  $C[n, k]$ 

```

What is the time efficiency of this algorithm? Clearly, the algorithm's basic operation is addition, so let $A(n, k)$ be the total number of additions made by this algorithm in computing $C(n, k)$. Note that computing each entry by formula (8.3) requires just one addition. Also note that because the first $k + 1$ rows of the table form a triangle while the remaining $n - k$ rows form a rectangle, we have to split the sum expressing $A(n, k)$ into two parts:

$$\begin{aligned}
A(n, k) &= \sum_{i=1}^k \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^n \sum_{j=1}^k 1 = \sum_{i=1}^k (i-1) + \sum_{i=k+1}^n k \\
&= \frac{(k-1)k}{2} + k(n-k) \in \Theta(nk).
\end{aligned}$$

You are asked to ascertain whether this is an efficient algorithm by comparing it with the running times of a few other algorithms for this problem in the exercises. Another problem in the exercises is to analyze whether or not the extra space used by the dynamic programming algorithm is actually necessary.

Exercises 8.1

1. **a.** What does dynamic programming have in common with divide-and-conquer?
- b.** What is a principal difference between the two techniques?
2. Compute the Fibonacci series using dynamic programming.
3. Prove the following assertion made in the text while investigating the time efficiency of the dynamic programming algorithm for computing $C(n, k)$:

$$\frac{(k-1)k}{2} + k(n-k) \in \Theta(nk).$$

4. **a.** Write a pseudocode for computing $C(n, k)$.
- b.** What is the space and time complexity of a dynamic programming algorithm for computing the binomial coefficient $C(n, k)$?

5. a. Find the order of growth of the following functions:
- $C(n, 1)$
 - $C(n, 2)$
 - $C(n, n/2)$ for even n 's
- b. What major implication for computing $C(n, k)$ do the answers to the questions in part (a) have?
6. Find the exact number of additions made by the following recursive algorithm based directly on formulas (8.3) and (8.4).

ALGORITHM *BinomCoeff(n, k)*

```
if  $k = 0$  or  $k = n$  return 1
else return BinomCoeff( $n - 1, k - 1$ ) + BinomCoeff( $n - 1, k$ )
```

7. Which of the following algorithms for computing a binomial coefficient is most efficient?

- a. Use the formula

$$C(n, k) = \frac{n!}{k!(n-k)!}.$$

- b. Use the formula

$$C(n, k) = \frac{n(n-1)\dots(n-k+1)}{k!}.$$

- c. Apply recursively the formula

$$C(n, k) = C(n-1, k-1) + C(n-1, k) \quad \text{for } n > k > 0,$$

$$C(n, 0) = C(n, n) = 1.$$

- d. Apply the dynamic programming algorithm.

8. Design a DP algorithm for the knapsack problem.



9. *Shortest path counting* A chess rook can move horizontally or vertically to any square in the same row or in the same column of a chessboard. Find the number of shortest paths by which a rook can move from one corner of a chessboard to the diagonally opposite corner. [Gar78], p. 10

- a. by a dynamic programming algorithm.

- b. by using elementary combinatorics.

10. *World Series odds* Consider two teams, A and B , playing a series of games until one of the teams wins n games. Assume that the probability of A winning a game is the same for each game and equal to p , and the probability of A losing a game is $q = 1 - p$. (Hence, there are no ties.) Let $P(i, j)$ be the probability of A winning the series if A needs i more games to win the series and B needs j more games to win the series.

- a. Set up a recurrence relation for $P(i, j)$ that can be used by a dynamic programming algorithm.

Since this method traverses the same digraph several times, we should hope that a better algorithm can be found. Indeed, such an algorithm exists. It is called **Warshall's algorithm** after S. Warshall [War62]. Warshall's algorithm constructs the transitive closure of a given digraph with n vertices through a series of n -by- n boolean matrices:

$$R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}. \quad (8.5)$$

Each of these matrices provides certain information about directed paths in the digraph. Specifically, the element $r_{ij}^{(k)}$ in the i th row and j th column of matrix $R^{(k)}$ ($k = 0, 1, \dots, n$) is equal to 1 if and only if there exists a directed path (of a positive length) from the i th vertex to the j th vertex with each intermediate vertex, if any, numbered not higher than k . Thus, the series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing else but the adjacency matrix of the digraph. (Recall that the adjacency matrix contains the information about one-edge paths, i.e., paths with no intermediate vertices.) $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate; thus, with more freedom, so to speak, it may contain more ones than $R^{(0)}$. In general, each subsequent matrix in series (8.5) has one more vertex to use as intermediate for its paths than its predecessor and hence may, but does not have to, contain more ones. The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing else but the digraph's transitive closure.

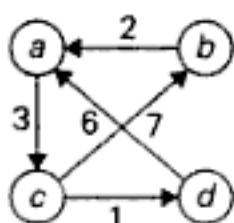
The central point of the algorithm is that we can compute all the elements of each matrix $R^{(k)}$ from its immediate predecessor $R^{(k-1)}$ in series (8.5). Let $r_{ij}^{(k)}$, the element in the i th row and j th column of matrix $R^{(k)}$, be equal to 1. This means that there exists a path from the i th vertex v_i to the j th vertex v_j with each intermediate vertex numbered not higher than k :

$$v_i, \text{ a list of intermediate vertices each numbered not higher than } k, v_j. \quad (8.6)$$

Two situations regarding this path are possible. In the first, the list of its intermediate vertices does not contain the k th vertex. Then this path from v_i to v_j has intermediate vertices numbered not higher than $k-1$, and therefore $r_{ij}^{(k-1)}$ is equal to 1 as well. The second possibility is that path (8.6) does contain the k th vertex v_k among the intermediate vertices. Without loss of generality, we may assume that v_k occurs only once in that list. (If it is not the case, we can create a new path from v_i to v_j with this property by simply eliminating all the vertices between the first and last occurrences of v_k in it.) With this caveat, path (8.6) can be rewritten as follows:

$$v_i, \text{ vertices numbered } \leq k-1, v_k, \text{ vertices numbered } \leq k-1, v_j.$$

The first part of this representation means that there exists a path from v_i to v_k with each intermediate vertex numbered not higher than $k-1$ (hence $r_{ik}^{(k-1)} = 1$), and the second part means that there exists a path from v_k to v_j with each intermediate vertex numbered not higher than $k-1$ (hence $r_{kj}^{(k-1)} = 1$).



(a)

$$W = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \left[\begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{matrix} \right] \end{matrix}$$

(b)

$$D = \begin{bmatrix} a & b & c & d \\ a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{bmatrix}$$

(c)

FIGURE 8.5 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

each vertex to all other vertices. It is convenient to record the lengths of shortest paths in an n -by- n matrix D called the *distance matrix*: the element d_{ij} in the i th row and the j th column of this matrix indicates the length of the shortest path from the i th vertex to the j th vertex ($1 \leq i, j \leq n$). For an example, see Figure 8.5.

We can generate the distance matrix with an algorithm that is very similar to Warshall's algorithm. It is called ***Floyd's algorithm***, after its inventor R. Floyd [Flo62]. It is applicable to both undirected and directed weighted graphs provided that they do not contain a cycle of a negative length. (Of course, in the case of a directed graph, by a path or cycle we mean a directed path or a directed cycle.)

Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of n -by- n matrices:

$$D^{(0)}, \dots, D^{(k-1)}, D^{(k)}, \dots, D^{(n)}. \quad (8.8)$$

Each of these matrices contains the lengths of shortest paths with certain constraints on the paths considered for the matrix in question. Specifically, the element $d_{ij}^{(k)}$ in the i th row and the j th column of matrix $D^{(k)}$ ($k = 0, 1, \dots, n$) is equal to the length of the shortest path among all paths from the i th vertex to the j th vertex with each intermediate vertex, if any, numbered not higher than k . In particular, the series starts with $D^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $D^{(0)}$ is nothing but the weight matrix of the graph. The last matrix in the series, $D^{(n)}$, contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate and hence is nothing but the distance matrix being sought.

As in Warshall's algorithm, we can compute all the elements of each matrix $D^{(k)}$ from its immediate predecessor $D^{(k-1)}$ in series (8.8). Let $d_{ij}^{(k)}$ be the element in the i th row and the j th column of matrix $D^{(k)}$. This means that $d_{ij}^{(k)}$ is equal to the length of the shortest path among all paths from the i th vertex v_i to the j th vertex v_j with their intermediate vertices numbered not higher than k :

v_i , a list of intermediate vertices each numbered not higher than k , v_j . (8.9)

We can partition all such paths into two disjoint subsets: those that do not use the k th vertex v_k as intermediate and those that do. Since the paths of the first subset

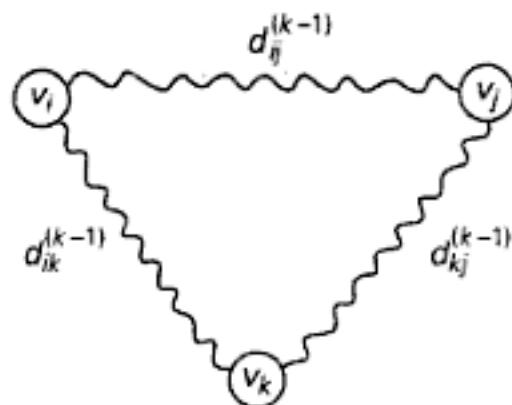


FIGURE 8.6 Underlying idea of Floyd's algorithm

have their intermediate vertices numbered not higher than $k - 1$, the shortest of them is, by definition of our matrices, of length $d_{ij}^{(k-1)}$.

What is the length of the shortest path in the second subset? If the graph does not contain a cycle of a negative length, we can limit our attention only to the paths in the second subset that use vertex v_k as their intermediate vertex exactly once (because visiting v_k more than once can only increase the path's length). All such paths have the following form:

$$v_i, \text{vertices numbered } \leq k-1, v_k, \text{vertices numbered } \leq k-1, v_j.$$

In other words, each of the paths is made up of a path from v_i to v_k with each intermediate vertex numbered not higher than $k - 1$ and a path from v_k to v_j with each intermediate vertex numbered not higher than $k - 1$. The situation is depicted symbolically in Figure 8.6.

Since the length of the shortest path from v_i to v_k among the paths that use intermediate vertices numbered not higher than $k - 1$ is equal to $d_{ik}^{(k-1)}$ and the length of the shortest path from v_k to v_j among the paths that use intermediate vertices numbered not higher than $k - 1$ is equal to $d_{kj}^{(k-1)}$, the length of the shortest path among the paths that use the k th vertex is equal to $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$. Taking into account the lengths of the shortest paths in both subsets leads to the following recurrence:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1, \quad d_{ij}^{(0)} = w_{ij}. \quad (8.10)$$

To put it another way, the element in the i th row and the j th column of the current distance matrix $D^{(k-1)}$ is replaced by the sum of the elements in the same row i and the k th column and in the same column j and the k th column if and only if the latter sum is smaller than its current value.

The application of Floyd's algorithm to the graph in Figure 8.5 is illustrated in Figure 8.7.

Here is a pseudocode of Floyd's algorithm. It takes advantage of the fact that the next matrix in sequence (8.8) can be written over its predecessor.

somewhat different from its original formulation, it says that an optimal solution to any instance of an optimization problem is composed of optimal solutions to its subinstances. The principle of optimality holds more often than not. (To give a rather rare example, it fails for finding longest simple paths.) Although its applicability to a particular problem needs to be verified, of course, such a verification is usually not a principal difficulty in developing a dynamic programming algorithm. The challenge typically lies in figuring out what smaller subinstances need to be considered and in deriving an equation relating a solution to any instance to solutions to its smaller subinstances. We consider a few more examples in the remaining sections of this chapter and their exercises.

Exercises 8.2

1. Write an algorithm to find the adjacency matrix and its transitive closure.
2.
 - a. Prove that the time efficiency of Warshall's algorithm is cubic.
 - b. Explain why the time efficiency of Warshall's algorithm is inferior to that of the traversal-based algorithm for sparse graphs represented by their adjacency lists.
3. Explain how to implement Warshall's algorithm without using extra memory for storing elements of the algorithm's intermediate matrices.
4. Explain how to restructure the innermost loop of the algorithm *Warshall* to make it run faster at least on some inputs.
5. Rewrite the pseudocode of Warshall's algorithm assuming that the matrix rows are represented by bit strings on which the bitwise *or* operation can be performed.
6. Write a pseudocode to find the weight matrix and the distance matrix for a digraph. Explain with an example.
7. Compare matrix generation for Warshall's algorithm and Floyd's algorithm, giving an example for each.
8. Prove that the next matrix in sequence (8.8) of Floyd's algorithm can be written over its predecessor.
9. Give an example of a graph or a digraph with negative weights for which Floyd's algorithm does not yield the correct result.
10. Enhance Floyd's algorithm so that shortest paths themselves, not just their lengths, can be found.
11. **Jack Straws** In the game of Jack Straws, a number of plastic or wooden "straws" are dumped on the table and players try to remove them one-by-one without disturbing the other straws. Here, we are only concerned with whether various pairs of straws are connected by a path of touching straws. Given a list of the endpoints for $n > 1$ straws (as if they were dumped on a large piece of graph paper), determine all the pairs of straws that are connected. Note that touching is connecting, but also that two straws can be connected



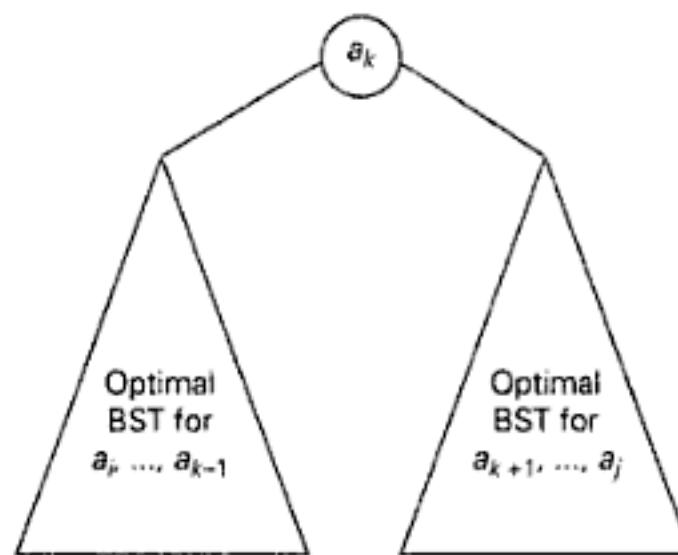


FIGURE 8.9 Binary search tree (BST) with root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^j

Thus, following the classic dynamic programming approach, we will find values of $C[i, j]$ for all smaller instances of the problem, although we are interested just in $C[1, n]$. To derive a recurrence underlying the dynamic programming algorithm, we will consider all possible ways to choose a root a_k among the keys a_i, \dots, a_j . For such a binary search tree (Figure 8.9), the root contains key a_k , the left subtree T_i^{k-1} contains keys a_i, \dots, a_{k-1} optimally arranged, and the right subtree T_{k+1}^j contains keys a_{k+1}, \dots, a_j also optimally arranged. (Note how we are taking advantage of the principle of optimality here.)

If we count tree levels starting with 1 (to make the comparison numbers equal the keys' levels), the following recurrence relation is obtained:

$$\begin{aligned}
 C[i, j] &= \min_{i \leq k \leq j} \left\{ p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1) \right. \\
 &\quad \left. + \sum_{s=k+1}^j p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^j + 1) \right\} \\
 &= \min_{i \leq k \leq j} \left\{ p_k + \sum_{s=i}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=i}^{k-1} p_s \right. \\
 &\quad \left. + \sum_{s=k+1}^j p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^j + \sum_{s=k+1}^j p_s \right\} \\
 &= \min_{i \leq k \leq j} \left\{ \sum_{s=i}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=k+1}^j p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^j + \sum_{s=i}^j p_s \right\} \\
 &= \min_{i \leq k \leq j} \left\{ C[i, k - 1] + C[k + 1, j] \right\} + \sum_{s=i}^j p_s.
 \end{aligned}$$

indices of the roots of the optimal subtrees, which makes it possible to reconstruct an optimal tree for the entire set given.

EXAMPLE Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The initial tables look like this:

main table					root table				
0	1	2	3	4	0	1	2	3	4
1	0	0.1			1	1			
2		0	0.2		2		2		
3			0	0.4	3			3	
4				0	4				4
5					5				

Let us compute $C[1, 2]$:

$$C[1, 2] = \min \begin{cases} k=1: C[1, 0] + C[2, 2] + \sum_{s=1}^2 p_s = 0 + 0.2 + 0.3 = 0.5 \\ k=2: C[1, 1] + C[3, 2] + \sum_{s=1}^2 p_s = 0.1 + 0 + 0.3 = 0.4 \end{cases} = 0.4.$$

Thus, out of two possible binary trees containing the first two keys, A and B , the root of the optimal tree has index 2 (i.e., it contains B), and the average number of comparisons in a successful search in this tree is 0.4.

We will ask you to finish the computations in the exercises. You should arrive at the following final tables:

main table					root table				
0	1	2	3	4	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7	1	1	2	3
2		0	0.2	0.8	1.4	2		2	3
3			0	0.4	1.0	3			3
4				0	0.3	4			4
5					0	5			

Thus, the average number of key comparisons in the optimal tree is equal to 1.7. Since $R[1, 4] = 3$, the root of the optimal tree contains the third key, i.e., C . Its left subtree is made up of keys A and B , and its right subtree contains just key D (why?). To find the specific structure of these subtrees, we find first their roots by consulting the root table again as follows. Since $R[1, 2] = 2$, the root of the optimal tree containing A and B is B , with A being its left child (and the root of the one-node tree: $R[1, 1] = 1$). Since $R[4, 4] = 4$, the root of this one-node optimal tree is its only key D . Figure 8.11 presents the optimal tree in its entirety. ■

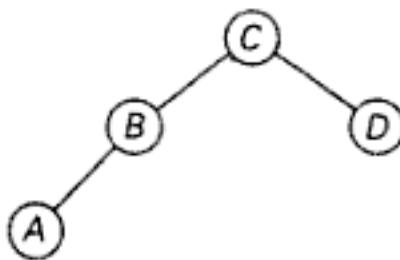


FIGURE 8.11 Optimal binary search tree for the example

Here is a pseudocode of the dynamic programming algorithm.

ALGORITHM *OptimalBST($P[1..n]$)*

```

//Finds an optimal binary search tree by dynamic programming
//Input: An array  $P[1..n]$  of search probabilities for a sorted list of  $n$  keys
//Output: Average number of comparisons in successful searches in the
//        optimal BST and table  $R$  of subtrees' roots in the optimal BST
for  $i \leftarrow 1$  to  $n$  do
     $C[i, i - 1] \leftarrow 0$ 
     $C[i, i] \leftarrow P[i]$ 
     $R[i, i] \leftarrow i$ 
     $C[n + 1, n] \leftarrow 0$ 
for  $d \leftarrow 1$  to  $n - 1$  do //diagonal count
    for  $i \leftarrow 1$  to  $n - d$  do
         $j \leftarrow i + d$ 
         $minval \leftarrow \infty$ 
        for  $k \leftarrow i$  to  $j$  do
            if  $C[i, k - 1] + C[k + 1, j] < minval$ 
                 $minval \leftarrow C[i, k - 1] + C[k + 1, j]; kmin \leftarrow k$ 
             $R[i, j] \leftarrow kmin$ 
             $sum \leftarrow P[i];$  for  $s \leftarrow i + 1$  to  $j$  do  $sum \leftarrow sum + P[s]$ 
             $C[i, j] \leftarrow minval + sum$ 
return  $C[1, n], R$ 
  
```

The algorithm's space efficiency is clearly quadratic; the time efficiency of this version of the algorithm is cubic (why?). A more careful analysis shows that entries in the root table are always nondecreasing along each row and column. This limits values for $R[i, j]$ to the range $R[i, j - 1], \dots, R[i + 1, j]$ and makes it possible to reduce the running time of the algorithm to $\Theta(n^2)$.

Exercises 8.3

1. Finish the computations started in the section's example of constructing an optimal binary search tree.

2. a. Compute the time complexity of algorithm *OptimalBST*.
b. Compute the space complexity of algorithm *OptimalBST*.
3. Write the DP for the *OptimalBST* problem.
4. Devise a way to compute the sums $\sum_{s=i}^j p_s$, which are used in the dynamic programming algorithm for constructing an optimal binary search tree, in constant time (per sum).
5. Write a C code for a linear-time algorithm, that generates the optimal binary search tree for the root table.
6. How would you construct an optimal binary search tree for a set of n keys if all the keys are equally likely to be searched for? What will be the average number of comparisons in a successful search in such a tree if $n = 2^k$?
7. a. Show that the number of distinct binary search trees $b(n)$ that can be constructed for a set of n orderable keys satisfies the recurrence relation

$$b(n) = \sum_{k=0}^{n-1} b(k)b(n-1-k) \quad \text{for } n > 0, \quad b(0) = 1.$$

- b. It is known that the solution to this recurrence is given by the Catalan numbers. Verify this assertion for $n = 1, 2, \dots, 5$.
- c. Find the order of growth of $b(n)$. What implication does the answer to this question have for the exhaustive-search algorithm for constructing an optimal binary search tree?
8. Design a $\Theta(n^2)$ algorithm for finding an optimal binary search tree.
9. Generalize the optimal binary search algorithm by taking into account unsuccessful searches.
10. **Matrix chain multiplication** Consider the problem of minimizing the total number of multiplications made in computing the product of n matrices

$$A_1 \cdot A_2 \cdot \dots \cdot A_n$$

whose dimensions are d_0 by d_1 , d_1 by d_2, \dots, d_{n-1} by d_n , respectively. Assume that all intermediate products of two matrices are computed by the brute-force (definition-based) algorithm.

- a. Give an example of three matrices for which the number of multiplications in $(A_1 \cdot A_2) \cdot A_3$ and $A_1 \cdot (A_2 \cdot A_3)$ differ at least by a factor 1000.
- b. How many different ways are there to compute the chained product of n matrices?
- c. Design a dynamic programming algorithm for finding an optimal order of multiplying n matrices.

8.4 The Knapsack Problem and Memory Functions

We start this section with designing the dynamic programming algorithm for the knapsack problem: given n items of known weights w_1, \dots, w_n and values v_1, \dots, v_n and a knapsack of capacity W , find the most valuable subset of the items that fit into the knapsack. (This problem was introduced in Section 3.4, where we discussed solving it by an exhaustive-search algorithm.) We assume here that all the weights and the knapsack's capacity are positive integers; the item values do not have to be integers.

To design a dynamic programming algorithm, we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances. Let us consider an instance defined by the first i items, $1 \leq i \leq n$, with weights w_1, \dots, w_i , values v_1, \dots, v_i , and knapsack capacity j , $1 \leq j \leq W$. Let $V[i, j]$ be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j . We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the i th item and those that do. Note the following:

1. Among the subsets that do not include the i th item, the value of an optimal subset is, by definition, $V[i - 1, j]$.
2. Among the subsets that do include the i th item (hence, $j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i - 1$ items that fit into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + V[i - 1, j - w_i]$.

Thus, the value of an optimal solution among all feasible subsets of the first i items is the maximum of these two values. Of course, if the i th item does not fit into the knapsack, the value of an optimal subset selected from the first i items is the same as the value of an optimal subset selected from the first $i - 1$ items. These observations lead to the following recurrence:

$$V[i, j] = \begin{cases} \max\{V[i - 1, j], v_i + V[i - 1, j - w_i]\} & \text{if } j - w_i \geq 0 \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases} \quad (8.12)$$

It is convenient to define the initial conditions as follows:

$$V[0, j] = 0 \text{ for } j \geq 0 \text{ and } V[i, 0] = 0 \text{ for } i \geq 0. \quad (8.13)$$

Our goal is to find $V[n, W]$, the maximal value of a subset of the n given items that fit into the knapsack of capacity W , and an optimal subset itself.

Figure 8.12 illustrates the values involved in equations (8.12) and (8.13). For $i, j > 0$, to compute the entry in the i th row and the j th column, $V[i, j]$, we compute the maximum of the entry in the previous row and the same column and the sum of v_i and the entry in the previous row and w_i columns to the left. The table can be filled either row by row or column by column.

Memory Functions

As we discussed at the beginning of this chapter and illustrated in subsequent sections, dynamic programming deals with problems whose solutions satisfy a recurrence relation with overlapping subproblems. The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient (typically, exponential or worse). The classic dynamic programming approach, on the other hand, works bottom-up: it fills a table with solutions to *all* smaller subproblems, but each of them is solved only once. An unsatisfying aspect of this approach is that solutions to some of these smaller subproblems are often not necessary for getting a solution to the problem given. Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches. The goal is to get a method that solves only subproblems which are necessary and does it only once. Such a method exists; it is based on using *memory functions* [Bra96].

This method solves a given problem in the top-down manner but, in addition, maintains a table of the kind that would have been used by a bottom-up dynamic programming algorithm. Initially, all the table's entries are initialized with a special “null” symbol to indicate that they have not yet been calculated. Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first: if this entry is not “null,” it is simply retrieved from the table; otherwise, it is computed by the recursive call whose result is then recorded in the table.

The following algorithm implements this idea for the knapsack problem. After initializing the table, the recursive function needs to be called with $i = n$ (the number of items) and $j = W$ (the capacity of the knapsack).

ALGORITHM *MFKnapsack(i,j)*

```

//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//       items being considered and a nonnegative integer j indicating
//       the knapsack's capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table V[0..n, 0..W] whose entries are initialized with -1's except for
//row 0 and column 0 initialized with 0's
if V[i,j] < 0
    if j < Weights[i]
        value  $\leftarrow$  MFKnapsack(i - 1, j)
    else
        value  $\leftarrow$  max(MFKnapsack(i - 1, j),
                           Values[i] + MFKnapsack(i - 1, j - Weights[i]))
    V[i,j]  $\leftarrow$  value
return V[i,j]

```

	capacity j					
i	0	1	2	3	4	5
$w_1 = 2, v_1 = 12$	0	0	0	0	0	0
$w_2 = 1, v_2 = 10$	1	0	0	12	12	12
$w_3 = 3, v_3 = 20$	2	0	-	12	22	-
$w_4 = 2, v_4 = 15$	3	0	-	-	22	-
	4	0	-	-	-	37

FIGURE 8.14 Example of solving an instance of the knapsack problem by the memory function algorithm

EXAMPLE 2 Let us apply the memory function method to the instance considered in Example 1. Figure 8.14 gives the results. Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) have been computed. Just one nontrivial entry, $V[1, 2]$, is retrieved rather than being recomputed. For larger instances, the proportion of such entries can be significantly larger. ■

In general, we cannot expect more than a constant-factor gain in using the memory function method for the knapsack problem because its time efficiency class is the same as that of the bottom-up algorithm (why?). A more significant improvement can be expected for dynamic programming algorithms in which a computation of one value takes more than constant time. You should also keep in mind that a memory function method may be less space-efficient than a space-efficient version of a bottom-up algorithm.

Exercises 8.4

1. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value	
1	7	\$42	
2	3	\$12	capacity $W = 10$
3	4	\$40	
4	5	\$25	

2. a. Write a pseudocode for constructing a table for solving the knapsack problem by DP.
 b. Explain the above by considering an example.
3. For the bottom-up dynamic programming algorithm for the knapsack problem, prove that
- a. its time efficiency is in $\Theta(nW)$.
 - b. its space efficiency is in $\Theta(nW)$.

- c. the time needed to find the composition of an optimal subset from a filled dynamic programming table is in $O(n + W)$.
4. a. True or false: A sequence of values in a row of the dynamic programming table for an instance of the knapsack problem is always nondecreasing?
b. True or false: A sequence of values in a column of the dynamic programming table for an instance of the knapsack problem is always nondecreasing?
5. Apply the memory function method to the instance of the knapsack problem given in Problem 1. Indicate the entries of the dynamic programming table that are: (i) never computed by the memory function method on this instance; (ii) retrieved without a recomputation.
6. Why is the time efficiency class of the memory function method for the knapsack problem the same as that of the bottom-up algorithm?
7. Write a pseudocode of a memory function for the optimal binary search tree problem. (You may limit your function to finding the smallest number of key comparisons in a successful search.)
8. Give two reasons why the memory function approach is unattractive for the problem of computing a binomial coefficient.
9. Design a dynamic programming algorithm for the *change-making problem*: given an amount n and unlimited quantities of coins of each of the denominations d_1, d_2, \dots, d_m , find the smallest number of coins that add up to n or indicate that the problem does not have a solution.
10. Write a research report on one of the following well-known applications of dynamic programming:
 - a. finding the longest common subsequence in two sequences
 - b. optimal string editing
 - c. minimal triangulation of a polygon

SUMMARY

- *Dynamic programming* is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type. Dynamic programming suggests solving each smaller subproblem once and recording the results in a table from which a solution to the original problem can be then obtained.
- Applicability of dynamic programming to an optimization problem requires the problem to satisfy the *principle of optimality*: an optimal solution to any of its instances must be made up of optimal solutions to its subinstances.

- Computing a binomial coefficient via constructing the Pascal triangle can be viewed as an application of the dynamic programming technique to a nonoptimization problem.
- *Warshall's algorithm* for finding the transitive closure and *Floyd's algorithm* for the all-pairs shortest-paths problem are based on the idea that can be interpreted as an application of the dynamic programming technique.
- Dynamic programming can be used for constructing an *optimal binary search tree* for a given set of keys and known probabilities of searching for them.
- Solving a knapsack problem by a dynamic programming algorithm exemplifies an application of this technique to difficult problems of combinatorial optimization.
- The *memory function* technique seeks to combine strengths of the top-down and bottom-up approaches to solving problems with overlapping subproblems. It does this by solving, in the top-down fashion but only once, just necessary subproblems of a given problem and recording their solutions in a table.

9

Greedy Technique

Greed, for lack of a better word, is good! Greed is right! Greed works!

—Michael Douglas, U.S. actor in the role of Gordon Gecko,
in the film *Wall Street*, 1987

Let us start with the *change-making problem* faced by millions of cashiers all over the world (at least subconsciously): give change for a specific amount n with the least number of coins of the denominations $d_1 > d_2 > \dots > d_m$ used in that locale. For example, the widely used coin denominations in the United States are $d_1 = 25$ (quarter), $d_2 = 10$ (dime), $d_3 = 5$ (nickel), and $d_4 = 1$ (penny). How would you give change with coins of these denominations of, say, 48 cents? If you came up with the answer 1 quarter, 2 dimes, and 3 pennies, you followed—consciously or not—a logical strategy of making a sequence of best choices among the currently available alternatives. Indeed, in the first step, you could have given one coin of any of the four denominations. “Greedy” thinking leads to giving one quarter because it reduces the remaining amount the most, namely, to 23 cents. In the second step, you had the same coins at your disposal, but you could not give a quarter because it would have violated the problem’s constraints. So your best selection in this step was one dime, reducing the remaining amount to 13 cents. Giving one more dime left you with 3 cents to be given with three pennies.

Is this solution to the instance of the change-making problem optimal? Yes, it is. In fact, it is possible to prove that the greedy algorithm yields an optimal solution for every positive integer amount with these coin denominations. At the same time, it is easy to give an example of “weird” coin denominations—e.g., $d_1 = 7$, $d_2 = 5$, $d_3 = 1$ —that may not yield an optimal solution for some amounts. (It is the reason you were asked to develop a dynamic programming algorithm for this problem in Exercises 8.4: that algorithm works for any set of coin denominations by either returning an optimal solution or reporting that no solution exists.)

The approach applied in the opening paragraph to the change-making problem is called *greedy*. Computer scientists consider it a general design technique despite the fact that it is applicable to optimization problems only. The greedy approach suggests constructing a solution through a sequence of steps, each

expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step—and this is the central point of this technique—the choice made must be

- *feasible*, i.e., it has to satisfy the problem's constraints
- *locally optimal*, i.e., it has to be the best local choice among all feasible choices available on that step
- *irrevocable*, i.e., once made, it cannot be changed on subsequent steps of the algorithm

These requirements explain the technique's name: on each step, it suggests a “greedy” grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem. We refrain from a philosophical discussion of whether greed is good or bad. (If you have not seen the movie from which the chapter's epigraph is taken, its hero did not end up well.) From our algorithmic perspective, the question is whether a greedy strategy works or not. As we shall see, there are problems for which a sequence of locally optimal choices does yield an optimal solution for every instance of the problem in question. However, there are others for which this is not the case; for such problems, a greedy algorithm can still be of value if we are interested in or have to be satisfied with an approximate solution.

In the first two sections of the chapter, we discuss two classic algorithms for the minimum spanning tree problem: Prim's algorithm and Kruskal's algorithm. What is remarkable about these algorithms is the fact that they solve the same problem by applying the greedy approach in two different ways, and both of them always yield an optimal solution. In Section 9.3, we introduce another classic algorithm—Dijkstra's algorithm for the shortest-path problem in a weighted graph. Section 9.4 is devoted to Huffman trees and their principal application, Huffman codes—an important data compression method that can be interpreted as an application of the greedy technique. Finally, a few examples of approximation algorithms based on the greedy approach are discussed in Section 12.3.

As a rule, greedy algorithms are both intuitively appealing and simple. Despite their apparent simplicity, there is a rather sophisticated theory behind the technique, which is based on the abstract combinatorial structure called “matroid.” We are not going to discuss it here; an interested reader can check such sources as [Cor01].

9.1 Prim's Algorithm

The following problem arises naturally in several practical situations: given n points, connect them in the cheapest possible way so that there will be a path between every pair of points. We can represent the points by vertices of a graph, possible connections by the graph's edges, and the connection costs by the edge weights. Then the question can be posed as the minimum spanning tree problem, defined formally as follows.

DEFINITION A *spanning tree* of a connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. A *minimum spanning*

The nature of Prim's algorithm makes it necessary to provide each vertex not in the current tree with the information about the shortest edge connecting the vertex to a tree vertex. We can provide such information by attaching two labels to a vertex: the name of the nearest tree vertex and the length (the weight) of the corresponding edge. Vertices that are not adjacent to any of the tree vertices can be given the ∞ label indicating their "infinite" distance to the tree vertices and a null label for the name of the nearest tree vertex. (Alternatively, we can split the vertices that are not in the tree into two sets, the "fringe" and the "unseen." The fringe contains only the vertices that are not in the tree but are adjacent to at least one tree vertex. These are the candidates from which the next tree vertex is selected. The unseen vertices are all the other vertices of the graph, called "unseen" because they are yet to be affected by the algorithm.) With such labels, finding the next vertex to be added to the current tree $T = (V_T, E_T)$ becomes a simple task of finding a vertex with the smallest distance label in the set $V - V_T$. Ties can be broken arbitrarily.

After we have identified a vertex u^* to be added to the tree, we need to perform two operations:

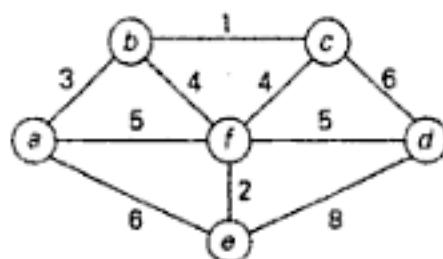
- Move u^* from the set $V - V_T$ to the set of tree vertices V_T .
- For each remaining vertex u in $V - V_T$ that is connected to u^* by a shorter edge than the u 's current distance label, update its labels by u^* and the weight of the edge between u^* and u , respectively.¹

Figure 9.2 demonstrates the application of Prim's algorithm to a specific graph.

Does Prim's algorithm always yield a minimum spanning tree? The answer to this question is yes. Let us prove by induction that each of the subtrees T_i , $i = 0, \dots, n - 1$, generated by Prim's algorithm is a part (i.e., a subgraph) of some minimum spanning tree. (This immediately implies, of course, that the last tree in the sequence, T_{n-1} , is a minimum spanning tree itself because it contains all n vertices of the graph.) The basis of the induction is trivial, since T_0 consists of a single vertex and hence must be a part of any minimum spanning tree. For the inductive step, let us assume that T_{i-1} is part of some minimum spanning tree T . We need to prove that T_i , generated from T_{i-1} by Prim's algorithm, is also a part of a minimum spanning tree. We prove this by contradiction by assuming that no minimum spanning tree of the graph can contain T_i . Let $e_i = (v, u)$ be the minimum weight edge from a vertex in T_{i-1} to a vertex not in T_{i-1} used by Prim's algorithm to expand T_{i-1} to T_i . By our assumption, e_i cannot belong to the minimum spanning tree T . Therefore, if we add e_i to T , a cycle must be formed (Figure 9.3).

In addition to edge $e_i = (v, u)$, this cycle must contain another edge (v', u') connecting a vertex $v' \in T_{i-1}$ to a vertex u' that is not in T_{i-1} . (It is possible that v' coincides with v or u' coincides with u but not both.) If we now delete the edge (v', u') from this cycle, we obtain another spanning tree of the entire graph whose weight is less than or equal to the weight of T since the weight of e_i is less than or

1. If the implementation with the fringe-unseen split is pursued, all the unseen vertices adjacent to u^* must also be moved to the fringe.



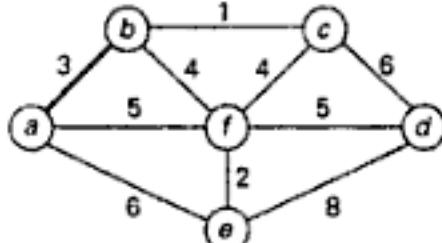
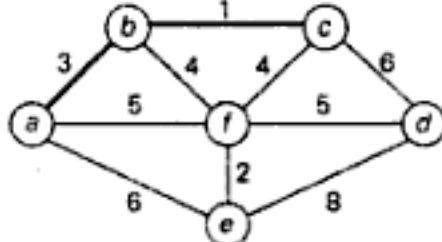
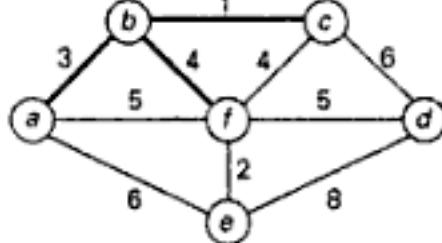
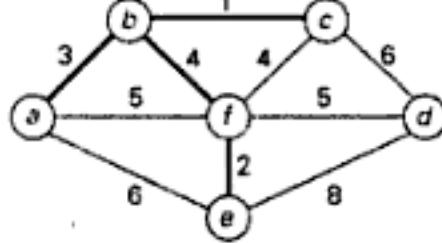
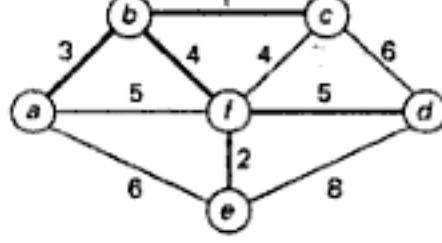
Tree vertices	Remaining vertices	Illustration
a(-, -)	b(a, 3) c(-, ∞) d(-, ∞) e(a, 6) f(a, 5)	
b(a, 3)	c(b, 1) d(-, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) f(b, 4)	
f(b, 4)	d(f, 5) e(f, 2)	
e(f, 2)	d(f, 5)	
d(f, 5)		

FIGURE 9.2 Application of Prim's algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.

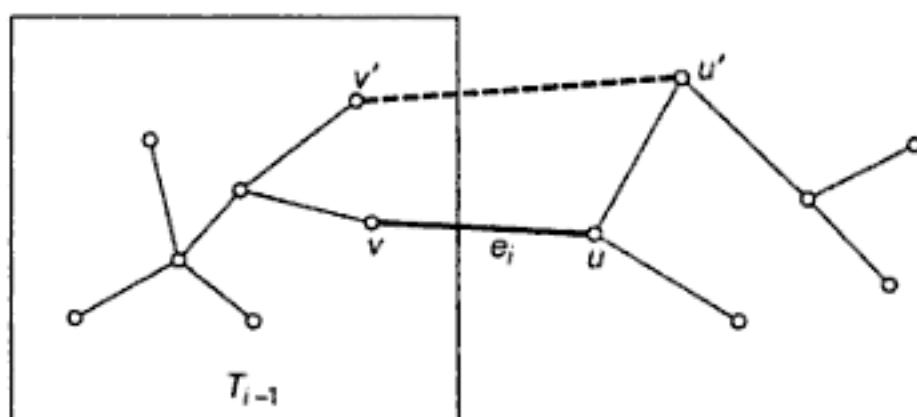


FIGURE 9.3 Correctness proof of Prim's algorithm

equal to the weight of (v', u') . Hence, this spanning tree is a minimum spanning tree, which contradicts the assumption that no minimum spanning tree contains T_i . This completes the correctness proof of Prim's algorithm.

How efficient is Prim's algorithm? The answer depends on the data structures chosen for the graph itself and for the priority queue of the set $V - V_T$ whose vertex priorities are the distances to the nearest tree vertices. (You may want to take another look at the example in Figure 9.2 to see that the set $V - V_T$ indeed operates as a priority queue.) In particular, if a graph is represented by its weight matrix and the priority queue is implemented as an unordered array, the algorithm's running time will be in $\Theta(|V|^2)$. Indeed, on each of the $|V| - 1$ iterations, the array implementing the priority queue is traversed to find and delete the minimum and then to update, if necessary, the priorities of the remaining vertices.

We can also implement the priority queue as a *min-heap*. A min-heap is a mirror image of the heap structure discussed in Section 6.4. (In fact, it can be implemented by constructing a heap after negating all the key values given.) Namely, a min-heap is a complete binary tree in which every element is less than or equal to its children. All the principal properties of heaps remain valid for min-heaps, with some obvious modifications. For example, the root of a min-heap contains the smallest rather than the largest element. Deletion of the smallest element from and insertion of a new element into a min-heap of size n are $O(\log n)$ operations, and so is the operation of changing an element's priority (see Problem 10).

If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is in $O(|E| \log |V|)$. This is because the algorithm performs $|V| - 1$ deletions of the smallest element and makes $|E|$ verifications and, possibly, changes of an element's priority in a min-heap of size not greater than $|V|$. Each of these operations, as noted earlier, is a $O(\log |V|)$ operation. Hence, the running time of this implementation of Prim's algorithm is in

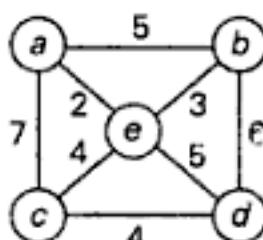
$$(|V| - 1 + |E|)O(\log |V|) = O(|E| \log |V|)$$

because, in a connected graph, $|V| - 1 \leq |E|$.

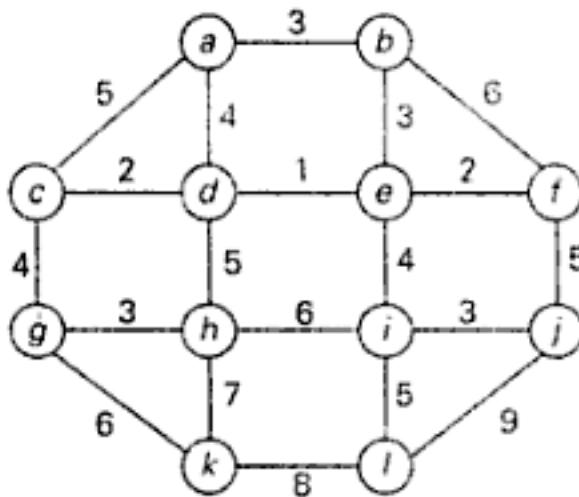
In the next section, you will find another greedy algorithm for the minimum spanning tree problem, which is “greedy” in a manner different from that of Prim's algorithm.

Exercises 9.1

1. Give an instance of the change-making problem for which the greedy algorithm does not yield an optimal solution.
2. Write a pseudocode of the greedy algorithm for the change-making problem with an amount n and coin denominations $d_1 > d_2 > \dots > d_m$ as its input. What is the time efficiency class of your algorithm?
3. Consider the problem of scheduling n jobs of known durations t_1, t_2, \dots, t_n for execution by a single processor. The jobs can be executed in any order, one job at a time. You want to find a schedule that minimizes the total time spent by all the jobs in the system. (The time spent by one job in the system is the sum of the time spent by this job in waiting plus the time spent on its execution.) Design a greedy algorithm for this problem. Does the greedy algorithm always yield an optimal solution?
4. Design a greedy algorithm for knapsack problem. We are given n objects and a knapsack or a bag. Object i has a weight w_i , and the knapsack has capacity m . If a fraction X_i ($0 \leq X_i \leq 1$) of object i is placed into the knapsack, profit of $P_i X_i$ is earned. The objective is to fill the knapsack in such a way as to maximize the total profit earned.
5. *Bridge crossing revisited* Consider the generalization of the bridge crossing puzzle (Problem 2 in Exercises 1.2) in which we have $n > 1$ people whose bridge crossing times are t_1, t_2, \dots, t_n . All the other conditions of the problem remain the same: only two people at the time can cross the bridge (and they move with the speed of the slower of the two) and they must carry with them the only flashlight the group has.
Design a greedy algorithm for this problem and find how long it will take to cross the bridge by using this algorithm. Does your algorithm yield a minimum crossing time for every instance of the problem? If it does—prove it; if it does not—find an instance with the smallest number of people for which this happens.
6. Apply the greedy method to the knapsack problem, where $n = 3$, $m = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$. (HINT: where n is the number of objects, m is the knapsack capacity, w is the weight and p is the profit.)
 - weights can be put only on the free cup of the scale.
 - weights can be put on both cups of the scale.
7. a. Apply Prim's algorithm to the following graph. Include in the priority queue all the vertices not already in the tree.



- b. Apply Prim's algorithm to the following graph. Include in the priority queue only the fringe vertices (the vertices not in the current tree which are adjacent to at least one tree vertex).



8. Prove by induction that Prim's construction actually yields the minimum spanning tree.
9. a. How can we use Prim's algorithm to find a spanning tree of a connected graph with no weights on its edges?
b. Is it a good algorithm for this problem?
10. Prove that any weighted connected graph with distinct weights has exactly one minimum spanning tree.
11. Outline an efficient algorithm for changing an element's value in a min-heap. What is the time efficiency of your algorithm?

9.2 Kruskal's Algorithm

In the previous section, we considered the greedy algorithm that “grows” a minimum spanning tree through a greedy inclusion of the nearest vertex to the vertices already in the tree. Remarkably, there is another greedy algorithm for the minimum spanning tree problem that also always yields an optimal solution. It is named **Kruskal's algorithm** [Kru56], after Joseph Kruskal, who discovered the algorithm when he was a second-year graduate student. Kruskal's algorithm looks at a minimum spanning tree for a weighted connected graph $G = (V, E)$ as an acyclic subgraph with $|V| - 1$ edges for which the sum of the edge weights is the smallest. (It is not difficult to prove that such a subgraph must be a tree.) Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs, which are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

ALGORITHM *Kruskal(G)*

```

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph  $G = (V, E)$ 
//Output:  $E_T$ , the set of edges composing a minimum spanning tree of  $G$ 
sort  $E$  in nondecreasing order of the edge weights  $w(e_{i_1}) \leq \dots \leq w(e_{i_{|E|}})$ 
 $E_T \leftarrow \emptyset$ ;  $eCounter \leftarrow 0$  //initialize the set of tree edges and its size
 $k \leftarrow 0$  //initialize the number of processed edges
while  $eCounter < |V| - 1$  do
     $k \leftarrow k + 1$ 
    if  $E_T \cup \{e_{i_k}\}$  is acyclic
         $E_T \leftarrow E_T \cup \{e_{i_k}\}$ ;  $eCounter \leftarrow eCounter + 1$ 
return  $E_T$ 

```

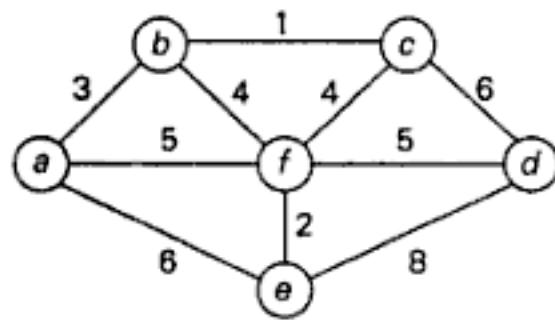
The correctness of Kruskal's algorithm can be proved by repeating the essential steps of the proof of Prim's algorithm given in the previous section. The fact that E_T is actually a tree in Prim's algorithm but generally just an acyclic subgraph in Kruskal's algorithm turns out to be an obstacle that can be overcome.

Figure 9.4 demonstrates the application of Kruskal's algorithm to the same graph we used for illustrating Prim's algorithm in Section 9.1. As you trace the algorithm's operations, note the disconnectedness of some of the intermediate graphs.

Applying Prim's and Kruskal's algorithms to the same small graph by hand may create an impression that the latter is simpler than the former. This impression is wrong because, on each of its iterations, Kruskal's algorithm has to check whether the addition of the next edge to the edges already selected would create a cycle. It is not difficult to see that a new cycle is created if and only if the new edge connects two vertices already connected by a path, i.e., if and only if the two vertices belong to the same connected component (Figure 9.5). Note also that each connected component of a subgraph generated by Kruskal's algorithm is a tree because it has no cycles.

In view of these observations, it is convenient to use a slightly different interpretation of Kruskal's algorithm. We can consider the algorithm's operations as a progression through a series of forests containing *all* the vertices of a given graph and *some* of its edges. The initial forest consists of $|V|$ trivial trees, each comprising a single vertex of the graph. The final forest consists of a single tree, which is a minimum spanning tree of the graph. On each iteration, the algorithm takes the next edge (u, v) from the sorted list of the graph's edges, finds the trees containing the vertices u and v , and, if these trees are not the same, unites them in a larger tree by adding the edge (u, v) .

Fortunately, there are efficient algorithms for doing so, including the crucial check whether two vertices belong to the same tree. They are called *union-find* algorithms. We discuss them in the following subsection. With an efficient union-find algorithm, the running time of Kruskal's algorithm will be dominated by the time needed for sorting the edge weights of a given graph. Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in $O(|E| \log |E|)$.



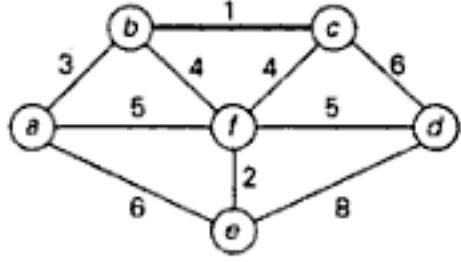
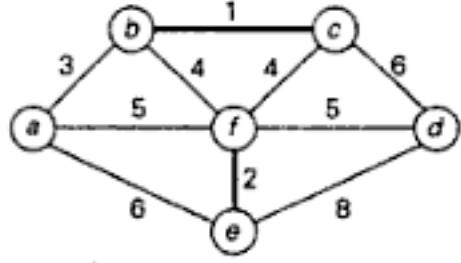
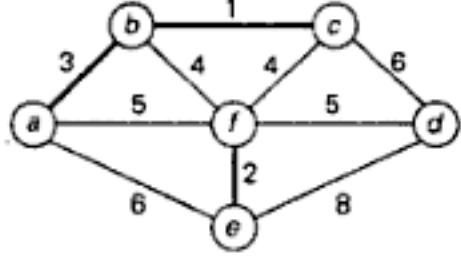
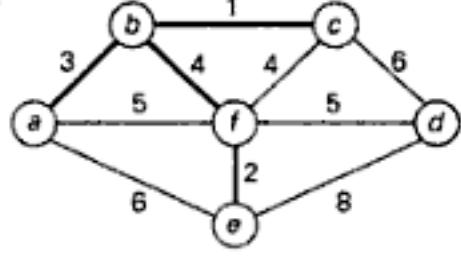
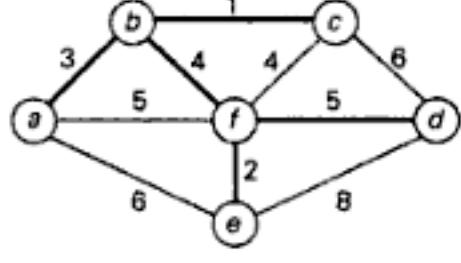
Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 4 5 5 6 6 8	
bc	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
df		

FIGURE 9.4 Application of Kruskal's algorithm. Selected edges are shown in bold.

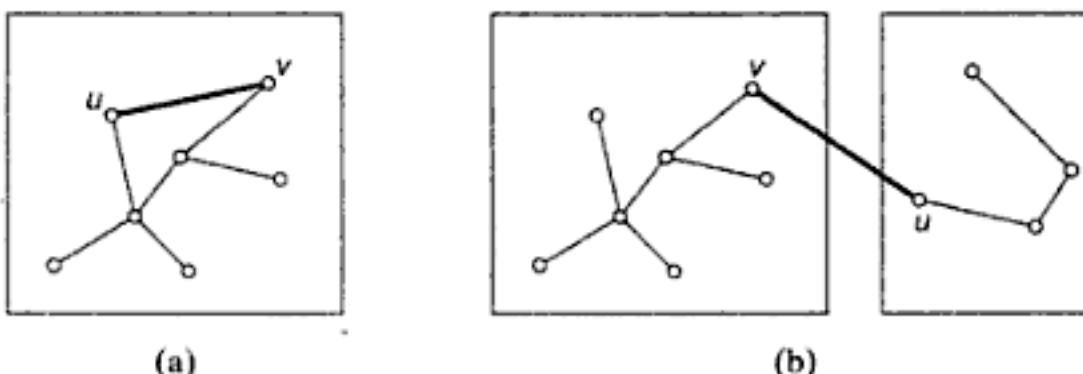


FIGURE 9.5 New edge connecting two vertices may (a) or may not (b) create a cycle

Disjoint Subsets and Union-Find Algorithms

Kruskal's algorithm is one of a number of applications that require a dynamic partition of some n -element set S into a collection of disjoint subsets S_1, S_2, \dots, S_k . After being initialized as a collection of n one-element subsets, each containing a different element of S , the collection is subjected to a sequence of intermixed union and find operations. (Note that the number of union operations in any such sequence must be bounded above by $n - 1$ because each union increases a subset's size at least by 1 and there are only n elements in the entire set S .) Thus, we are dealing here with an abstract data type of a collection of disjoint subsets of a finite set with the following operations:

makeset(x) creates a one-element set $\{x\}$. It is assumed that this operation can be applied to each of the elements of set S only once;

find(x) returns a subset containing x ;

union(x, y) constructs the union of the disjoint subsets S_x and S_y containing x and y , respectively, and adds it to the collection to replace S_x and S_y , which are deleted from it.

For example, let $S = \{1, 2, 3, 4, 5, 6\}$. Then *makeset(i)* creates the set $\{i\}$ and applying this operation six times initializes the structure to the collection of six singleton sets:

$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}.$

Performing *union(1, 4)* and *union(5, 2)* yields

$\{1, 4\}, \{5, 2\}, \{3\}, \{6\},$

and, if followed by *union(4, 5)* and then by *union(3, 6)*, we end up with the disjoint subsets

$\{1, 4, 5, 2\}, \{3, 6\}.$

Most implementations of this abstract data type use one element from each of the disjoint subsets in a collection as that subset's *representative*. Some implementations do not impose any specific constraints on such a representative; others do so by requiring, say, the smallest element of each subset to be used as the subset's representative. Also, it is usually assumed that set elements are (or can be mapped into) integers.

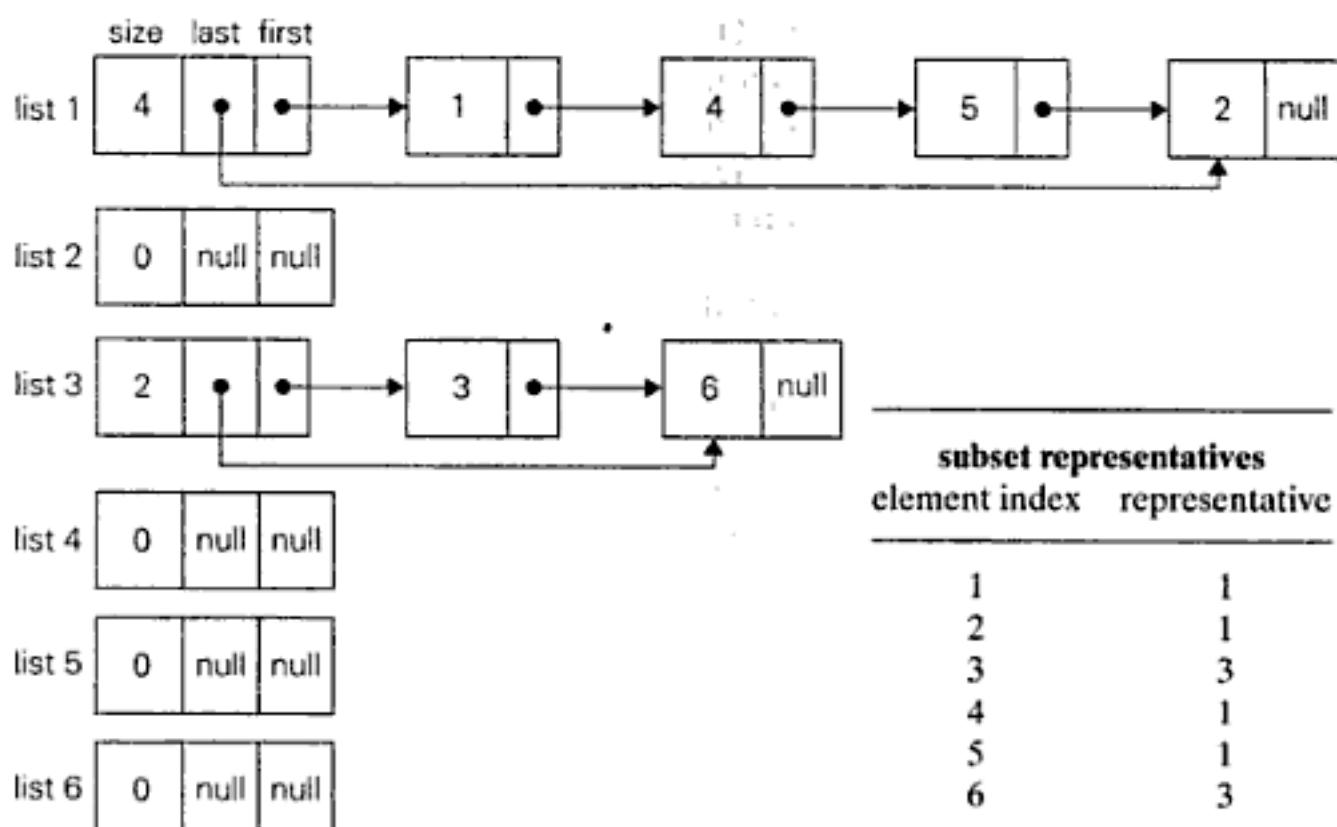


FIGURE 9.6 Linked-list representation of subsets {1, 4, 5, 2} and {3, 6} obtained by quick find after performing *union*(1,4), *union*(5,2), *union*(4,5), and *union*(3,6). The lists of size 0 are considered deleted from the collection.

There are two principal alternatives for implementing this data structure. The first one, called the *quick find*, optimizes the time efficiency of the find operation; the second one, called the *quick union*, optimizes the union operation.

The quick find uses an array indexed by the elements of the underlying set S ; the array's values indicate the representatives of the subsets containing those elements. Each subset is implemented as a linked list whose header contains the pointers to the first and last elements of the list along with the number of elements in the list (see Figure 9.6 for an example).

Under this scheme, the implementation of *makeset*(x) requires assigning the corresponding element in the representative array to x and initializing the corresponding linked list to a single node with the x value. The time efficiency of this operation is obviously in $\Theta(1)$, and hence the initialization of n singleton subsets is in $\Theta(n)$. The efficiency of *find*(x) is also in $\Theta(1)$: all we need to do is to retrieve the x 's representative in the representative array. Executing *union*(x, y) takes longer. A straightforward solution would simply append the y 's list to the end of the x 's list, update the information about their representative for all the elements in the y list, and then delete the y 's list from the collection. It is easy to verify, however, that with this algorithm the sequence of union operations

$$\textit{union}(2, 1), \textit{union}(3, 2), \dots, \textit{union}(i+1, i), \dots, \textit{union}(n, n-1)$$

runs in $\Theta(n^2)$ time, which is slow compared with several known alternatives.

A simple way to improve the overall efficiency of a sequence of union operations is to always append the shorter of the two lists to the longer one, with ties

broken arbitrarily. Of course, the size of each list is assumed to be available by, say, storing the number of elements in the list's header. This modification is called the ***union by size***. Though it does not improve the worst-case efficiency of a single application of the union operation (it is still in $\Theta(n)$), the worst-case running time of any legitimate sequence of union-by-size operations turns out to be in $O(n \log n)$.²

Here is a proof of this assertion. Let a_i be an element of set S whose disjoint subsets we manipulate and let A_i be the number of times a_i 's representative is updated in a sequence of union-by-size operations. How large can A_i get if set S has n elements? Each time a_i 's representative is updated, a_i must be in a smaller subset involved in computing the union whose size will be at least twice as large as the size of the subset containing a_i . Hence, when a_i 's representative is updated for the first time, the resulting set will have at least two elements; when it is updated for the second time, the resulting set will have at least four elements; and, in general, if it is updated A_i times, the resulting set will have at least 2^{A_i} elements. Since the entire set S has n elements, $2^{A_i} \leq n$ and hence $A_i \leq \log_2 n$. Therefore, the total number of possible updates of the representatives for all n elements in S will not exceed $n \log_2 n$.

Thus, for union by size, the time efficiency of a sequence of at most $n - 1$ unions and m finds is in $O(n \log n + m)$.

The ***quick union***—the second principal alternative for implementing disjoint subsets—represents each subset by a rooted tree. The nodes of the tree contain the subset's elements (one per node), with the root's element considered the subset's representative; the tree's edges are directed from children to their parents (Figure 9.7). (In addition, a mapping of the set elements to their tree nodes—implemented, say, as an array of pointers—is maintained. This mapping is not shown in Figure 9.7 for the sake of simplicity.)

For this implementation, *makeset(x)* requires the creation of a single-node tree, which is a $\Theta(1)$ operation; hence the initialization of n singleton subsets is in $\Theta(n)$. A *union(x, y)* is implemented by attaching the root of the *y*'s tree to the root

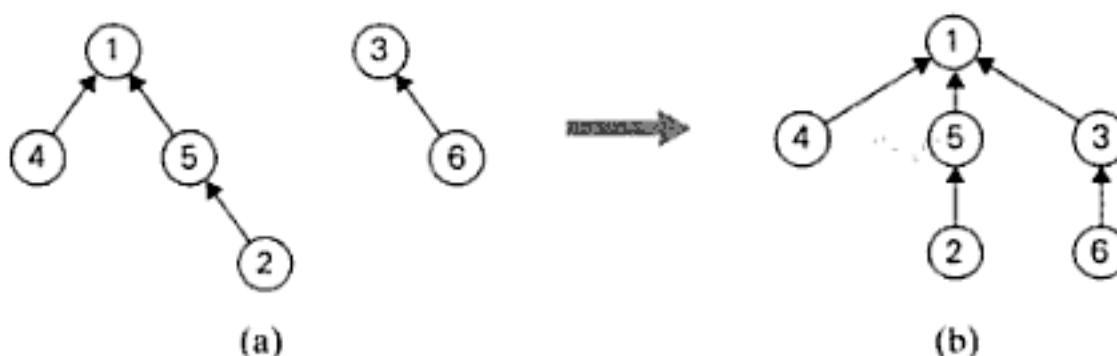


FIGURE 9.7 (a) Forest representation of subsets $\{1, 4, 5, 2\}$ and $\{3, 6\}$ used by quick union.
(b) Result of *union*(5, 6).

-
2. This is a specific example of the usefulness of the *amortized efficiency* we mentioned back in Chapter 2. The time efficiency of any sequence of n union-by-size operations is more efficient than the worst-case efficiency of its single application times n .

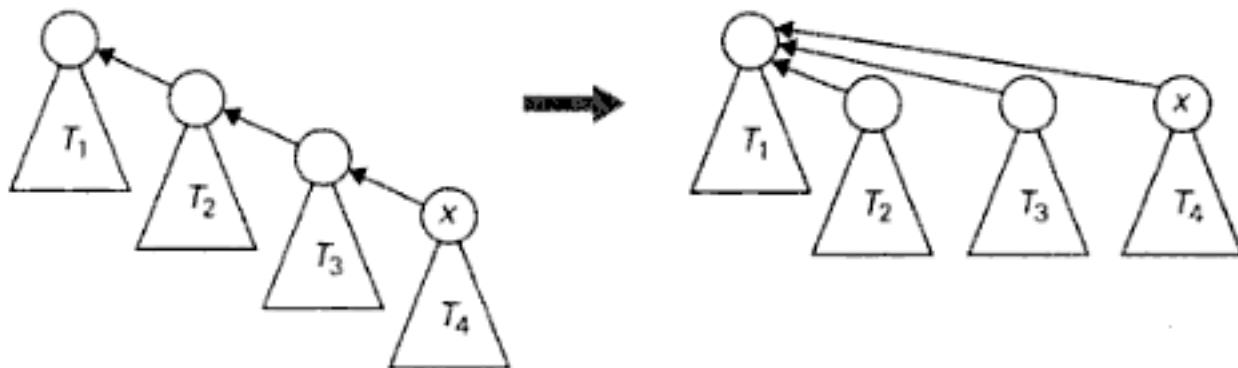


FIGURE 9.8 Path compression

of the x 's tree (and deleting the y 's tree from the collection by making the pointer to its root null). The time efficiency of this operation is clearly $\Theta(1)$. A $find(x)$ is performed by following the pointer chain from the node containing x to the tree's root (whose element is returned as the subset's representative). Accordingly, the time efficiency of a single $find$ operation is in $O(n)$ because a tree representing a subset can degenerate into a linked list with n nodes.

This time bound can be improved. The straightforward way for doing so is to always perform a union operation by attaching a smaller tree to the root of a larger one, with ties broken arbitrarily. The size of a tree can be measured either by the number of nodes (this version is called **union by size**) or by its height (this version is called **union by rank**). Of course, these options require storing, for each node of the tree, either the number of node descendants or the height of the subtree rooted at that node, respectively. One can easily prove that in either case the height of the tree will be logarithmic, making it possible to execute each $find$ in $O(\log n)$ time. Thus, for quick union, the time efficiency of a sequence of at most $n - 1$ unions and m finds is in $O(n + m \log n)$.

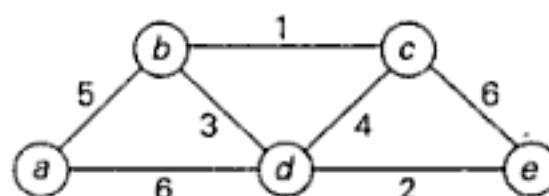
In fact, an even better efficiency can be obtained by combining either variety of quick union with **path compression**. This modification makes every node encountered during the execution of a $find$ operation point to the tree's root (Figure 9.8).

According to a quite sophisticated analysis that goes beyond the level of this book (see [Tar84]), this and similar techniques improve the efficiency of a sequence of at most $n - 1$ unions and m finds to only slightly worse than linear.

Exercises 9.2

1. Apply Kruskal's algorithm to find a minimum spanning tree of the following graphs.

a.



the book). The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common.

A variety of practical applications of the shortest-paths problem have made the problem a very popular object of study. There are several well-known algorithms for solving it, including Floyd's algorithm for the more general all-pairs shortest-paths problem discussed in Chapter 8. Here, we consider the best-known algorithm for the single-source shortest-paths problem, called **Dijkstra's algorithm**.³ This algorithm is applicable to graphs with nonnegative weights only. Since in most applications this condition is satisfied, the limitation has not impaired the popularity of Dijkstra's algorithm.

Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source. First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on. In general, before its i th iteration commences, the algorithm has already identified the shortest paths to $i - 1$ other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree T_i of the given graph (Figure 9.9). Since all the edge weights are nonnegative, the next vertex nearest to the source can be found among the vertices adjacent to the vertices of T_i . The set of vertices adjacent to the vertices in T_i can be referred to as "fringe vertices"; they are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source. (Actually, all the other vertices can be treated as fringe vertices connected to tree vertices by edges of infinitely large weights.) To identify the i th nearest vertex, the algorithm computes, for every fringe vertex u , the sum of the distance to the nearest tree vertex v (given by the

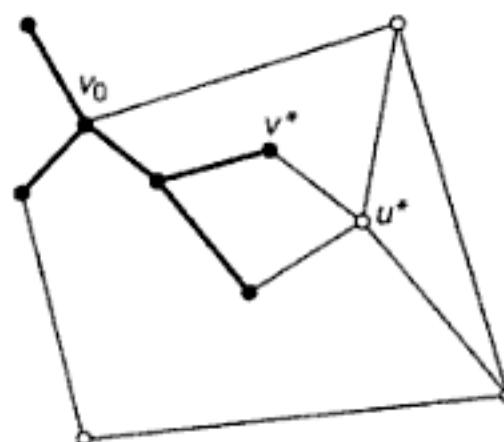


FIGURE 9.9 Idea of Dijkstra's algorithm. The subtree of the shortest paths already found is shown in bold. The next nearest to the source v_0 vertex, u^* , is selected by comparing the lengths of the subtree's paths increased by the distances to vertices adjacent to the subtree's vertices.

-
3. Edsger W. Dijkstra (1930–2002), a noted Dutch pioneer of the science and industry of computing, discovered this algorithm in the mid-1950s. Dijkstra said of his algorithm: "This was the first graph problem I ever posed myself and solved. The amazing thing was that I didn't publish it. It was not amazing at the time. At the time, algorithms were hardly considered a scientific topic."

weight of the edge (v, u)) and the length d_v of the shortest path from the source to v (previously determined by the algorithm) and then selects the vertex with the smallest such sum. The fact that it suffices to compare the lengths of such special paths is the central insight of Dijkstra's algorithm.

To facilitate the algorithm's operations, we label each vertex with two labels. The numeric label d indicates the length of the shortest path from the source to this vertex found by the algorithm so far; when a vertex is added to the tree, d indicates the length of the shortest path from the source to that vertex. The other label indicates the name of the next-to-last vertex on such a path, i.e., the parent of the vertex in the tree being constructed. (It can be left unspecified for the source s and vertices that are adjacent to none of the current tree vertices.) With such labeling, finding the next nearest vertex u^* becomes a simple task of finding a fringe vertex with the smallest d value. Ties can be broken arbitrarily.

After we have identified a vertex u^* to be added to the tree, we need to perform two operations:

- Move u^* from the fringe to the set of tree vertices.
- For each remaining fringe vertex u that is connected to u^* by an edge of weight $w(u^*, u)$ such that $d_{u^*} + w(u^*, u) < d_u$, update the labels of u by u^* and $d_{u^*} + w(u^*, u)$, respectively.

Figure 9.10 demonstrates the application of Dijkstra's algorithm to a specific graph.

The labeling and mechanics of Dijkstra's algorithm are quite similar to those used by Prim's algorithm (see Section 9.1). Both of them construct an expanding subtree of vertices by selecting the next vertex from the priority queue of the remaining vertices. It is important not to mix them up, however. They solve different problems and therefore operate with priorities computed in a different manner: Dijkstra's algorithm compares path lengths and therefore must add edge weights, while Prim's algorithm compares the edge weights as given.

Now we can give a pseudocode of Dijkstra's algorithm. It is spelled out—in more detail than Prim's algorithm was in Section 9.1—in terms of explicit operations on two sets of labeled vertices: the set V_T of vertices for which a shortest path has already been found and the priority queue Q of the fringe vertices. (Note that in the following pseudocode, V_T contains a given source vertex and the fringe contains the vertices adjacent to it *after* iteration 0 is completed.)

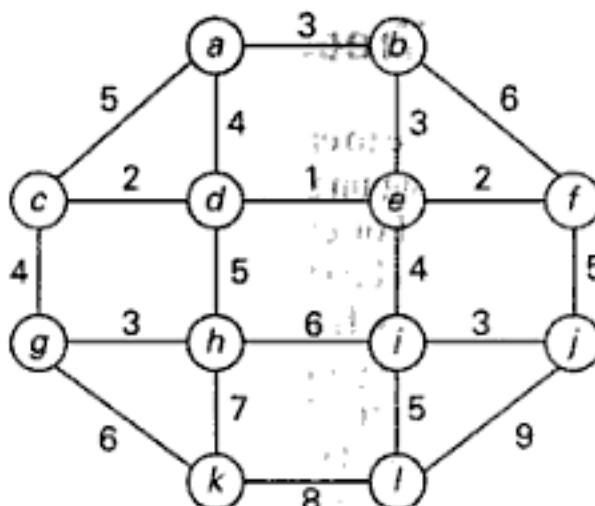
ALGORITHM *Dijkstra*(G, s)

```

//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph  $G = (V, E)$  with nonnegative weights
//      and its vertex  $s$ 
//Output: The length  $d_v$  of a shortest path from  $s$  to  $v$ 
//      and its penultimate vertex  $p_v$  for every vertex  $v$  in  $V$ 
Initialize( $Q$ ) //initialize vertex priority queue to empty
for every vertex  $v$  in  $V$  do
     $d_v \leftarrow \infty$ ;  $p_v \leftarrow \text{null}$ 
    Insert( $Q, v, d_v$ ) //initialize vertex priority in the priority queue

```

b.



3. Consider an example to explain Dijkstra's algorithm.
4. a. Does Dijkstra's algorithm work for graphs with negative weights?
b. Will it be applicable to both undirected and directed graphs?
5. Write a pseudocode of a simpler version of Dijkstra's algorithm that finds only the distances (i.e., the lengths of shortest paths but not shortest paths themselves) from a given vertex to all other vertices of a graph represented by its weight matrix.
6. Calculate the efficiency of Dijkstra's algorithm in two different representations.
7. Design a linear-time algorithm for solving the single-source shortest-paths problem for dags (directed acyclic graphs) represented by their adjacency lists.
8. Design an efficient algorithm for finding the length of a longest path in a dag. (This problem is important because it determines a lower bound on the total time needed for completing a project composed of precedence-constrained tasks.)
9. *Shortest-path modeling* Assume that you have a model of a weighted connected graph made of balls (representing the vertices) connected by strings of appropriate lengths (representing the edges).
 - a. Describe how you can solve the single-pair shortest-path problem with this model.
 - b. Describe how you can solve the single-source shortest-paths problem with this model.
10. Revisit Problem 6 in Exercises 1.3 about determining the best route for a subway passenger to take from one designated station to another in a well-developed subway system like those in Washington, DC, and London, UK. Write a program for this task.



9.4 Huffman Trees

Suppose we have to encode a text that comprises characters from some n -character alphabet by assigning to each of the text's characters some sequence of bits called the *codeword*. For example, we can use a *fixed-length encoding* that assigns to each character a bit string of the same length m ($m \geq \log_2 n$). This is exactly what the standard ASCII code does. One way of getting a coding scheme that yields a shorter bit string on the average is based on the old idea of assigning shorter codewords to more frequent characters and longer codewords to less frequent characters. (This idea was used, in particular, in the telegraph code invented in the mid-19th century by Samuel Morse. In that code, frequent letters such as *e* (·) and *a* (·-) are assigned short sequences of dots and dashes while infrequent letters such as *q* (---) and *z* (---..) have longer ones.)

Variable-length encoding, which assigns codewords of different lengths to different characters, introduces a problem that fixed-length encoding does not have. Namely, how can we tell how many bits of an encoded text represent the first (or, more generally, the i th) character? To avoid this complication, we can limit ourselves to *prefix-free* (or simply *prefix*) codes. In a prefix code, no codeword is a prefix of a codeword of another character. Hence, with such an encoding, we can simply scan a bit string until we get the first group of bits that is a codeword for some character, replace these bits by this character, and repeat this operation until the bit string's end is reached.

If we want to create a binary prefix code for some alphabet, it is natural to associate the alphabet's characters with leaves of a binary tree in which all the left edges are labeled by 0 and all the right edges are labeled by 1 (or vice versa). The codeword of a character can then be obtained by recording the labels on the simple path from the root to the character's leaf. Since there is no simple path to a leaf that continues to another leaf, no codeword can be a prefix of another codeword; hence, any such tree yields a prefix code.

Among the many trees that can be constructed in this manner for a given alphabet with known frequencies of the character occurrences, how can we construct a tree that would assign shorter bit strings to high-frequency characters and longer ones to low-frequency characters? It can be done by the following greedy algorithm, invented by David Huffman while he was a graduate student at MIT [Huf52].

Huffman's Algorithm

Step 1 Initialize n one-node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate the tree's *weight*. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in the exercises). Make them the left and right subtree

of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by the above algorithm is called a **Huffman tree**. It defines—in the manner described—a **Huffman code**.

EXAMPLE Consider the five-character alphabet {A, B, C, D, _} with the following occurrence probabilities:

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for this input is shown in Figure 9.11. The resulting codewords are as follows:

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Hence, DAD is encoded as 011101, and 10011011011101 is decoded as BAD_AD.

With the occurrence probabilities given and the codeword lengths obtained, the expected number of bits per character in this code is

$$2 \cdot 0.35 + 3 \cdot 0.1 + 2 \cdot 0.2 + 2 \cdot 0.2 + 3 \cdot 0.15 = 2.25.$$

Had we used a fixed-length encoding for the same alphabet, we would have to use at least three bits per each character. Thus, for this toy example, Huffman's code achieves the **compression ratio**—a standard measure of a compression algorithm's effectiveness—of $(3 - 2.25)/3 \cdot 100\% = 25\%$. In other words, we should expect that Huffman's encoding of a text will use 25% less memory than its fixed-length encoding. (Extensive experiments with Huffman codes have shown that the compression ratio for this scheme typically falls between 20% and 80%, depending on the characteristics of the text being compressed.) ■

Huffman's encoding is one of the most important file compression methods. In addition to its simplicity and versatility, it yields an optimal, i.e., minimal-length, encoding (provided the probabilities of character occurrences are independent and known in advance). The simplest version of Huffman compression calls, in fact, for a preliminary scanning of a given text to count the frequencies of character occurrences in it. Then these frequencies are used to construct a Huffman coding tree and encode the text as described above. This scheme makes it necessary, however, to include the information about the coding tree into the encoded text to make its decoding possible. This drawback can be overcome by using **dynamic Huffman encoding**, in which the coding tree is updated each time a new character is read from the source text (see, e.g., [Say00]).

It is important to note that applications of Huffman's algorithm are not limited to data compression. Suppose we have n positive numbers w_1, w_2, \dots, w_n that have

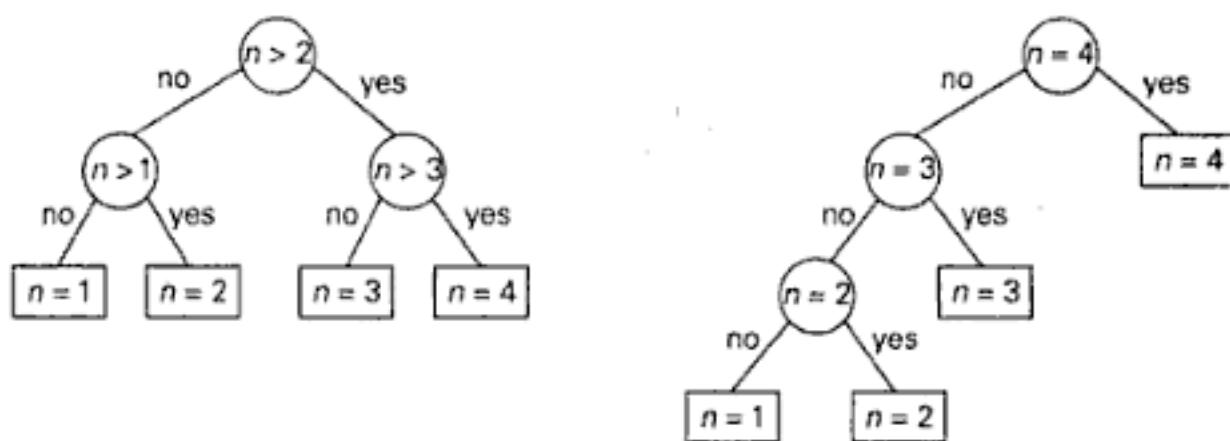


FIGURE 9.12 Two decision trees for guessing an integer between 1 and 4

questions answerable by yes or no. Different strategies for playing this game can be modeled by **decision trees**⁴ such as those depicted in Figure 9.12 for $n = 4$.

The length of the simple path from the root to a leaf in such a tree is equal to the number of questions needed to get to the chosen number represented by the leaf. If number i is chosen with probability p_i , the sum $\sum_{i=1}^n l_i p_i$, where l_i is the length of the simple path from the root to the i th leaf, indicates the average number of questions needed to “guess” the chosen number with a game strategy represented by its decision tree. If each of the numbers is chosen with the same probability of $1/n$, the best strategy is to successively eliminate half (or almost half) the candidates as binary search does. This may not be the case for arbitrary p_i ’s, however. (For example, if $n = 4$ and $p_1 = 0.1$, $p_2 = 0.2$, $p_3 = 0.3$, and $p_4 = 0.4$, the minimum weighted path tree is the rightmost one in Figure 9.12.) Thus, we need Huffman’s algorithm to solve this problem in its general case.

In conclusion, it is worthwhile to remember that this is the second time we are encountering the problem of constructing an optimal binary tree. In Section 8.3, we discussed the problem of constructing an optimal binary search tree with positive numbers (the search probabilities) assigned to every node of the tree. In this section, given numbers are assigned just to leaves. The latter problem turns out to be easier: it can be solved by the greedy algorithm whereas the former is solved by the more complicated dynamic programming algorithm.

Exercises 9.4

1. a. Construct a Huffman tree for the following data and obtain its Huffman code.

character	A	B	C	D	E	_
probability	0.5	0.35	0.5	0.1	0.4	0.2

4. Decision trees are discussed in more detail in Section 11.2.

- b. Encode the text DAD_BE using the code of question (a).
- c. Decode the text whose encoding is 1100110110 in the code of question (a).
2. For data transmission purposes, it is often desirable to have a code with a minimum variance of the codeword lengths (among codes of the same average length). Compute the average and variance of the codeword length in two Huffman codes that result from a different tie breaking during a Huffman code construction for the following data:

character	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4

3. Indicate whether each of the following properties are true for every Huffman code.
- The codewords of the two least frequent characters have the same length.
 - The codeword's length of a more frequent character is always smaller than or equal to the codeword's length of a less frequent one.
4. Write a program to calculate the maximal length of a codeword possible using Huffman encoding in a word of n alphabets.
5. a. Write a pseudocode for the Huffman-tree construction algorithm.
b. What is the time efficiency class of the algorithm for constructing a Huffman tree as a function of the alphabet's size?
6. For the given problem show that a Huffman tree can be constructed in linear time if the alphabets are given in a sorted order of their frequencies.

character	A	B	C	D	_
probability / frequency	0.1	0.15	0.15	0.2	0.4

7. Given a Huffman coding tree, which algorithm would you use to get the codewords for all the characters? What is its time-efficiency class as a function of the alphabet's size?
8. Explain how one can generate a Huffman code without an explicit generation of a Huffman coding tree.
9. a. Write a program that constructs a Huffman code for a given English text and encode it.
b. Write a program for decoding an English text that has been encoded with a Huffman code.
c. Experiment with your encoding program to find a range of typical compression ratios for Huffman's encoding of English texts of, say, 1000 words.
d. Experiment with your encoding program to find out how sensitive the compression ratios are to using standard estimates of frequencies instead of actual frequencies of character occurrences in English texts.



10. *Card guessing* Design a strategy that minimizes the expected number of questions asked in the following game [Gar94], #52. You have a deck of cards that consists of one ace of spades, two deuces of spades, three threes, and on up to nine nines, making 45 cards in all. Someone draws a card from the shuffled deck, which you have to identify by asking questions answerable with yes or no.

SUMMARY

- The *greedy technique* suggests constructing a solution to an optimization problem through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step, the choice made must be *feasible*, *locally optimal*, and *irrevocable*.
- *Prim's algorithm* is a greedy algorithm for constructing a minimum spanning tree of a weighted connected graph. It works by attaching to a previously constructed subtree a vertex closest to the vertices already in the tree.
- *Kruskal's algorithm* is another greedy algorithm for the minimum spanning tree problem. It constructs a minimum spanning tree by selecting edges in increasing order of their weights provided that the inclusion does not create a cycle. Checking the latter condition efficiently requires an application of one of the so-called *union-find algorithms*.
- *Dijkstra's algorithm* solves the single-source shortest-paths problem of finding shortest paths from a given vertex (the source) to all the other vertices of a weighted graph or digraph. It works as Prim's algorithm but compares path lengths rather than edge lengths. Dijkstra's algorithm always yields a correct solution for a graph with nonnegative weights.
- A *Huffman tree* is a binary tree that minimizes the weighted path length from the root to the leaves of predefined weights. The most important application of Huffman trees are Huffman codes.
- A *Huffman code* is an optimal prefix-free variable-length encoding scheme that assigns bit strings to characters based on their frequencies in a given text. This is accomplished by a greedy construction of a binary tree whose leaves represent the alphabet characters and whose edges are labeled with 0's and 1's.

Iterative Improvement

The most successful men in the end are those whose success is the result of steady accretion.

—Alexander Graham Bell (1835–1910)

The greedy strategy, considered in the preceding chapter, constructs a solution to an optimization problem piece by piece, always adding a locally optimal piece to a partially constructed solution. In this section, we discuss a different approach to designing algorithms for optimization problems. It starts with some feasible solution (a solution that satisfies all the constraints of the problem) and proceeds to improve it by repeated applications of some simple step. This step typically involves a small, localized change yielding a feasible solution with an improved value of the objective function. When no such change improves the value of the objective function, the algorithm returns the last feasible solution as optimal and stops.

There can be several obstacles to the successful implementation of this idea. First, we need an initial feasible solution. For some problems, we can always start with a trivial solution or use an approximate solution obtained by some other (e.g., greedy) algorithm. But for others, finding an initial solution may require as much effort as solving the problem after a feasible solution has been identified. Second, it is not always clear what changes should be allowed in a feasible solution so that we can check efficiently whether the current solution is locally optimal and, if not, replace it with a better one. Third—and this is the most fundamental difficulty—is an issue of local versus global extremum (maximum or minimum). Think about the problem of finding a highest point in a hilly area with no map on a foggy day. A logical thing to do would be to start walking “up the hill” from the point you are at until it becomes impossible to do so because no direction would lead up. You will have reached a local highest point, but because of a limited feasibility, there will be no simple way to tell whether the point is the highest (global maximum you are after) in the entire area.

Fortunately, there are important problems that can be solved by iterative-improvement algorithms. The most important of them is linear programming. We have already encountered this topic in Section 6.6. Here, in Section 10.1,

we introduce the simplex method, the classic algorithm for linear programming. Discovered by the U.S. mathematician George B. Dantzig in 1947, this algorithm has proved to be one of the most consequential achievements in the history of algorithmics.

In Section 10.2, we consider the important problem of maximizing the amount of flow that can be sent through a network with links of limited capacities. This problem is a special case of linear programming. However, its special structure makes it possible to solve the problem by algorithms that are more efficient than the simplex method. We outline the classic iterative improvement algorithm for this problem, discovered by the American mathematicians L. R. Ford, Jr., and D. R. Fulkerson in the 1950s.

The last two sections of the chapter deal with bipartite matching. This is a problem of finding an optimal pairing of elements taken from two disjoint sets. Examples include matching workers and jobs, high school graduates and colleges, and men and women for marriage. Section 10.3 deals with the problem of maximizing the number of matched pairs; Section 10.4 is concerned with the matching stability.

We also discuss several iterative improvement algorithms in Section 12.3, where we consider approximation algorithms for the traveling salesman and knapsack problems. Other examples of iterative improvement algorithms can be found in the algorithms textbook by Moret and Shapiro [Mor91], books on continuous and discrete optimization (e.g., [Nem89]), and the literature on heuristic search (e.g., [Mic00]).

10.1 The Simplex Method

We have already encountered linear programming (see Section 6.6)—the general problem of optimizing a linear function of several variables subject to a set of linear constraints:

$$\begin{aligned} \text{maximize (or minimize)} \quad & c_1x_1 + \dots + c_nx_n \\ \text{subject to} \quad & a_{i1}x_1 + \dots + a_{in}x_n \leq (\text{or } \geq \text{ or } =) b_i \text{ for } i = 1, \dots, m \\ & x_1 \geq 0, \dots, x_n \geq 0. \end{aligned} \tag{10.1}$$

We mentioned there that many important practical problems can be modeled as instances of linear programming. Two researchers, L. V. Kantorovich of the former Soviet Union and the Dutch-American T. C. Koopmans, were even awarded the Nobel Prize in 1975 for their contributions to linear programming theory and its applications to economics. Apparently because there is no Nobel Prize in mathematics, the Royal Swedish Academy of Sciences failed to honor the U.S. mathematician G. B. Dantzig, who is universally recognized as the father of linear programming in its modern form and the inventor of the simplex method, the classic algorithm for solving such problems.¹

1. George B. Dantzig (1914–2005) has received many honors, including the National Medal of Science presented by the president of the United States in 1976. The citation states that the National Medal was

Geometric Interpretation of Linear Programming

Before we introduce a general method for solving linear programming problems, let us consider a small example, which will help us to see the fundamental properties of such problems.

EXAMPLE 1 Consider the following linear programming problem in two variables:

$$\begin{aligned} & \text{maximize } 3x + 5y \\ & \text{subject to } x + y \leq 4 \\ & \quad x + 3y \leq 6 \\ & \quad x \geq 0, y \geq 0. \end{aligned} \tag{10.2}$$

By definition, a **feasible solution** to this problem is any point (x, y) that satisfies all the constraints of the problem; the problem's **feasible region** is the set of all its feasible points. It is instructive to sketch the feasible region in the Cartesian plane. Recall that any equation $ax + by = c$, where coefficients a and b are not both equal to zero, defines a straight line. Such a line divides the plane into two half-planes: for all the points in one of them, $ax + by < c$, while for all the points in the other, $ax + by > c$. (It is easy to determine which of the two half-planes is which: take any point (x_0, y_0) not on the line $ax + by = c$ and check which of the two inequalities hold, $ax_0 + by_0 > c$ or $ax_0 + by_0 < c$.) In particular, the set of points defined by inequality $x + y \leq 4$ comprises the points on and below the line $x + y = 4$, and the set of points defined by inequality $x + 3y \leq 6$ comprises the points on and below the line $x + 3y = 6$. Since the points of the feasible region must satisfy all the constraints of the problem, the feasible region is obtained by the intersection of these two half-planes and the first quadrant of the Cartesian plane defined by the nonnegativity constraints $x \geq 0, y \geq 0$ (see Figure 10.1). Thus, the feasible region for problem (10.2) is the convex polygon with the vertices $(0,0)$, $(4,0)$, $(0,2)$, and $(3,1)$. (The last point, which is the point of intersection of the lines $x + y = 4$ and $x + 3y = 6$, is obtained by solving the system of these two linear equations.) Our task is to find an **optimal solution**, a point in the feasible region with the largest value of the **objective function** $z = 3x + 5y$.

Are there feasible solutions for which the value of the objective function equals, say, 20? The points (x, y) for which the objective function $z = 3x + 5y$ is equal to 20 form the line $3x + 5y = 20$. Since this line does not have common points with the feasible region—see Figure 10.2—the answer to the posed question is no. On the other hand, there are infinitely many feasible points for which the objective function is equal to, say, 10: they are the intersection points of the line $3x + 5y = 10$ with the feasible region. Note that the lines $3x + 5y = 20$ and $3x + 5y = 10$ have the same slope, as would any line defined by equation $3x + 5y = z$

awarded "for inventing linear programming and discovering methods that led to wide-scale scientific and technical applications to important problems in logistics, scheduling, and network optimization, and to the use of computers in making efficient use of the mathematical theory."

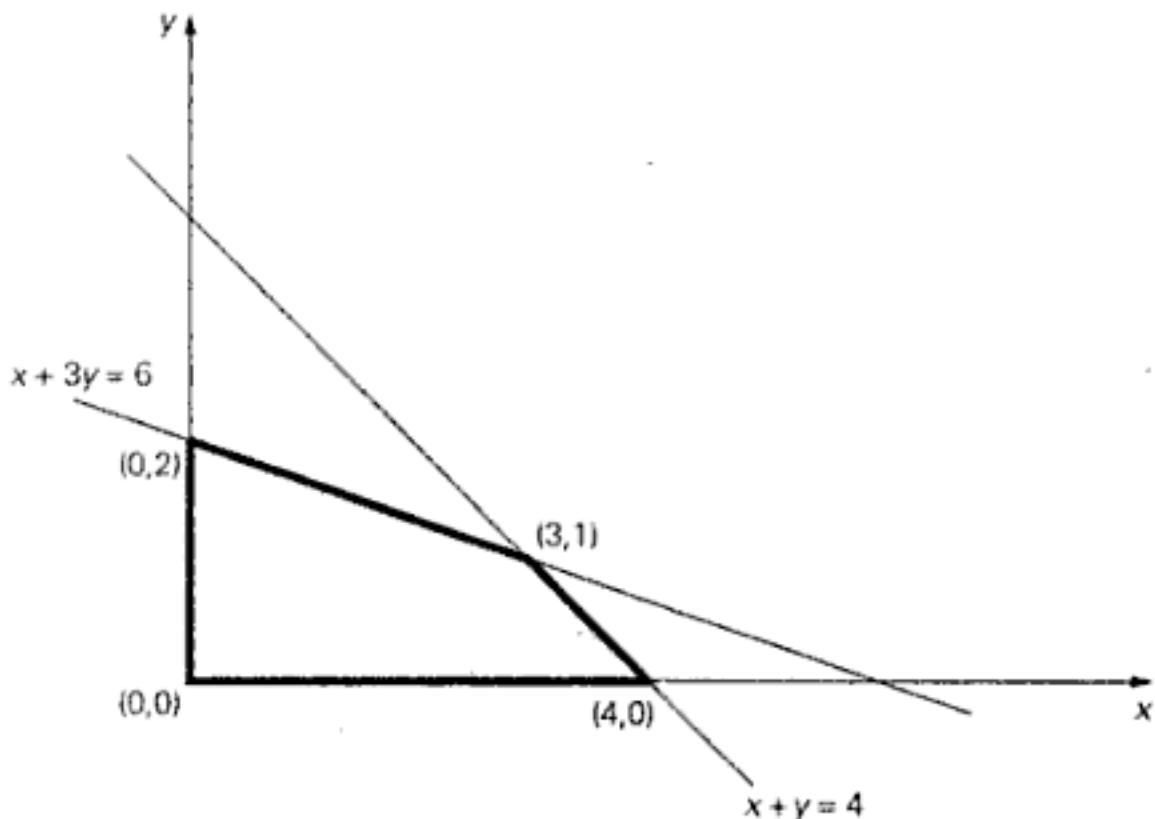


FIGURE 10.1 Feasible region of problem (10.2)

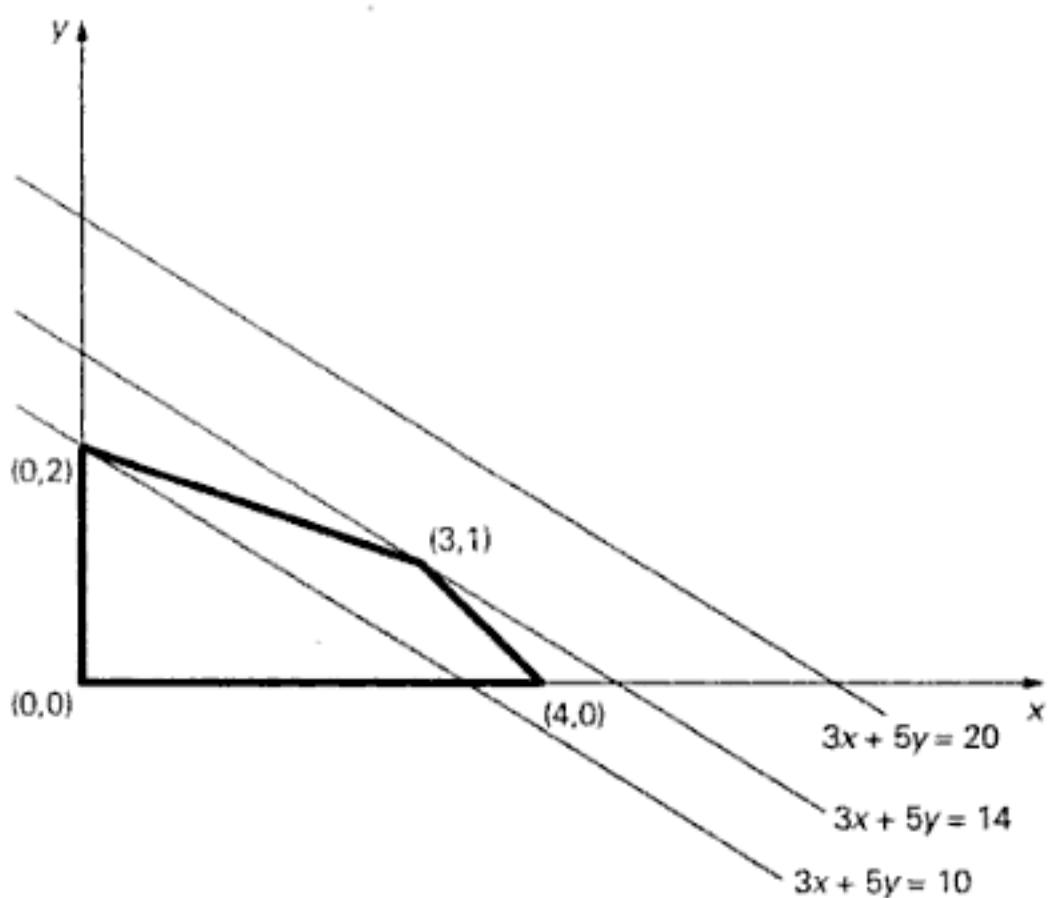


FIGURE 10.2 Solving a two-dimensional linear programming problem geometrically

where z is some constant. Such lines are called *level lines* of the objective function. Thus, our problem can be restated as finding the largest value of the parameter z for which the level line $3x + 5y = z$ has a common point with the feasible region.

We can find this line either by shifting, say, the line $3x + 5y = 20$ south-west (without changing its slope!) toward the feasible region until it hits the region for the first time or by shifting, say, the line $3x + 5y = 10$ north-east until it hits the feasible region for the last time. Either way, it will happen at the point $(3, 1)$ with the corresponding z value $3 \cdot 3 + 5 \cdot 1 = 14$. This means that the optimal solution to the linear programming problem in question is $x = 3, y = 1$, with the maximal value of the objective function equal to 14.

Note that if we had to maximize $z = 3x + 3y$ as the objective function in problem (10.2), the level line $3x + 3y = z$ for the largest value of z would coincide with the boundary line segment that has the same slope as the level lines (draw this line in Figure 10.2). Consequently, all the points of the line segment between vertices $(3, 1)$ and $(4, 0)$, including the vertices themselves, would be optimal solutions, yielding, of course, the same maximal value of the objective function. ■

Does every linear programming problem have an optimal solution that can be found at a vertex of its feasible region? Without appropriate qualifications, the answer to this question is no. To begin with, the feasible region of a linear programming problem can be empty. For example, if the constraints include two contradictory requirements, such as $x + y \leq 1$ and $x + y \geq 2$, there can be no points in the problem's feasible region. Linear programming problems with the empty feasible region are called *infeasible*. Obviously, infeasible problems do not have optimal solutions.

Another complication may arise if the problem's feasible region is unbounded, as the following example demonstrates.

EXAMPLE 2 If we reverse the inequalities in problem (10.2) to $x + y \geq 4$ and $x + 3y \geq 6$, the feasible region of the new problem will become unbounded (see Figure 10.3). If the feasible region of a linear programming problem is unbounded, its objective function may or may not attain a finite optimal value on it. For example, the problem of maximizing $z = 3x + 5y$ subject to the constraints $x + y \geq 4$, $x + 3y \geq 6$, $x \geq 0$, $y \geq 0$ has no optimal solution, because there are points in the feasible region making $3x + 5y$ as large as we wish. Such problems are called *unbounded*. On the other hand, the problem of minimizing $z = 3x + 5y$ subject to the same constraints has an optimal solution (which?). ■

Fortunately, the most important features of the examples we considered above hold for problems with more than two variables. In particular, a feasible region of a typical linear programming problem is in many ways similar to convex polygons in the two-dimensional Cartesian plane. Specifically, it always has a finite number of vertices, which mathematicians prefer to call *extreme points* (see Section 3.3). Furthermore, an optimal solution to a linear programming problem can be found at one of the extreme points of its feasible region. We reiterate these properties in the following theorem.