

Introduction to **The Design &
Analysis of Algorithms**

3RD EDITION

Vice President and Editorial Director, ECS	<i>Marcia Horton</i>
Editor-in-Chief	<i>Michael Hirsch</i>
Acquisitions Editor	<i>Matt Goldstein</i>
Editorial Assistant	<i>Chelsea Bell</i>
Vice President, Marketing	<i>Patrice Jones</i>
Marketing Manager	<i>Yezan Alayan</i>
Senior Marketing Coordinator	<i>Kathryn Ferranti</i>
Marketing Assistant	<i>Emma Snider</i>
Vice President, Production	<i>Vince O'Brien</i>
Managing Editor	<i>Jeff Holcomb</i>
Production Project Manager	<i>Kayla Smith-Tarbox</i>
Senior Operations Supervisor	<i>Alan Fischer</i>
Manufacturing Buyer	<i>Lisa McDowell</i>
Art Director	<i>Anthony Gemmellaro</i>
Text Designer	<i>Sandra Rigney</i>
Cover Designer	<i>Anthony Gemmellaro</i>
Cover Illustration	<i>Jennifer Kohnke</i>
Media Editor	<i>Daniel Sandin</i>
Full-Service Project Management	<i>Windfall Software</i>
Composition	<i>Windfall Software, using ZzT_EX</i>
Printer/Binder	<i>Courier Westford</i>
Cover Printer	<i>Courier Westford</i>
Text Font	<i>Times Ten</i>

Copyright © 2012, 2007, 2003 Pearson Education, Inc., publishing as Addison-Wesley. All rights reserved. Printed in the United States of America. This publication is protected by Copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to 201-236-3290.

This is the eBook of the printed book and may not include any media, Website access codes or print supplements that may come packaged with the bound book.

Many of the designations 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.

Library of Congress Cataloging-in-Publication Data

Levitin, Anany.

Introduction to the design & analysis of algorithms / Anany Levitin. — 3rd ed.

p. cm.

Includes bibliographical references and index.

ISBN-13: 978-0-13-231681-1

ISBN-10: 0-13-231681-1

1. Computer algorithms. I. Title. II. Title: Introduction to the design and analysis of algorithms.

QA76.9.A43L48 2012

005.1—dc23

2011027089

15 14 13 12 11—CRW—10 9 8 7 6 5 4 3 2 1

PEARSON

ISBN 10: 0-13-231681-1

ISBN 13: 978-0-13-231681-1

Introduction to **The Design &
Analysis of Algorithms**

3RD EDITION

Anany Levitin

Villanova University

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

This page intentionally left blank

Brief Contents

New to the Third Edition	xvii
Preface	xix
1 Introduction	1
2 Fundamentals of the Analysis of Algorithm Efficiency	41
3 Brute Force and Exhaustive Search	97
4 Decrease-and-Conquer	131
5 Divide-and-Conquer	169
6 Transform-and-Conquer	201
7 Space and Time Trade-Offs	253
8 Dynamic Programming	283
9 Greedy Technique	315
10 Iterative Improvement	345
11 Limitations of Algorithm Power	387
12 Coping with the Limitations of Algorithm Power	423
Epilogue	471
APPENDIX A	
 Useful Formulas for the Analysis of Algorithms	475
APPENDIX B	
 Short Tutorial on Recurrence Relations	479
References	493
Hints to Exercises	503
Index	547

This page intentionally left blank

Contents

New to the Third Edition	xvii
Preface	xix
1 Introduction	1
1.1 What Is an Algorithm?	3
Exercises 1.1	7
1.2 Fundamentals of Algorithmic Problem Solving	9
Understanding the Problem	9
Ascertaining the Capabilities of the Computational Device	9
Choosing between Exact and Approximate Problem Solving	11
Algorithm Design Techniques	11
Designing an Algorithm and Data Structures	12
Methods of Specifying an Algorithm	12
Proving an Algorithm's Correctness	13
Analyzing an Algorithm	14
Coding an Algorithm	15
Exercises 1.2	17
1.3 Important Problem Types	18
Sorting	19
Searching	20
String Processing	20
Graph Problems	21
Combinatorial Problems	21
Geometric Problems	22
Numerical Problems	22
Exercises 1.3	23

1.4 Fundamental Data Structures	25
Linear Data Structures	25
Graphs	28
Trees	31
Sets and Dictionaries	35
Exercises 1.4	37
Summary	38
2 Fundamentals of the Analysis of Algorithm Efficiency	41
2.1 The Analysis Framework	42
Measuring an Input's Size	43
Units for Measuring Running Time	44
Orders of Growth	45
Worst-Case, Best-Case, and Average-Case Efficiencies	47
Recapitulation of the Analysis Framework	50
Exercises 2.1	50
2.2 Asymptotic Notations and Basic Efficiency Classes	52
Informal Introduction	52
O -notation	53
Ω -notation	54
Θ -notation	55
Useful Property Involving the Asymptotic Notations	55
Using Limits for Comparing Orders of Growth	56
Basic Efficiency Classes	58
Exercises 2.2	58
2.3 Mathematical Analysis of Nonrecursive Algorithms	61
Exercises 2.3	67
2.4 Mathematical Analysis of Recursive Algorithms	70
Exercises 2.4	76
2.5 Example: Computing the nth Fibonacci Number	80
Exercises 2.5	83
2.6 Empirical Analysis of Algorithms	84
Exercises 2.6	89
2.7 Algorithm Visualization	91
Summary	94

3	Brute Force and Exhaustive Search	97
3.1	Selection Sort and Bubble Sort	98
	Selection Sort	98
	Bubble Sort	100
	Exercises 3.1	102
3.2	Sequential Search and Brute-Force String Matching	104
	Sequential Search	104
	Brute-Force String Matching	105
	Exercises 3.2	106
3.3	Closest-Pair and Convex-Hull Problems by Brute Force	108
	Closest-Pair Problem	108
	Convex-Hull Problem	109
	Exercises 3.3	113
3.4	Exhaustive Search	115
	Traveling Salesman Problem	116
	Knapsack Problem	116
	Assignment Problem	119
	Exercises 3.4	120
3.5	Depth-First Search and Breadth-First Search	122
	Depth-First Search	122
	Breadth-First Search	125
	Exercises 3.5	128
	Summary	130
4	Decrease-and-Conquer	131
4.1	Insertion Sort	134
	Exercises 4.1	136
4.2	Topological Sorting	138
	Exercises 4.2	142
4.3	Algorithms for Generating Combinatorial Objects	144
	Generating Permutations	144
	Generating Subsets	146
	Exercises 4.3	148

4.4 Decrease-by-a-Constant-Factor Algorithms	150
Binary Search	150
Fake-Coin Problem	152
Russian Peasant Multiplication	153
Josephus Problem	154
Exercises 4.4	156
4.5 Variable-Size-Decrease Algorithms	157
Computing a Median and the Selection Problem	158
Interpolation Search	161
Searching and Insertion in a Binary Search Tree	163
The Game of Nim	164
Exercises 4.5	166
Summary	167
5 Divide-and-Conquer	169
5.1 Mergesort	172
Exercises 5.1	174
5.2 Quicksort	176
Exercises 5.2	181
5.3 Binary Tree Traversals and Related Properties	182
Exercises 5.3	185
5.4 Multiplication of Large Integers and Strassen's Matrix Multiplication	186
Multiplication of Large Integers	187
Strassen's Matrix Multiplication	189
Exercises 5.4	191
5.5 The Closest-Pair and Convex-Hull Problems by Divide-and-Conquer	192
The Closest-Pair Problem	192
Convex-Hull Problem	195
Exercises 5.5	197
Summary	198

6	Transform-and-Conquer	201
6.1	Presorting	202
	Exercises 6.1	205
6.2	Gaussian Elimination	208
	<i>LU</i> Decomposition	212
	Computing a Matrix Inverse	214
	Computing a Determinant	215
	Exercises 6.2	216
6.3	Balanced Search Trees	218
	AVL Trees	218
	2-3 Trees	223
	Exercises 6.3	225
6.4	Heaps and Heapsort	226
	Notion of the Heap	227
	Heapsort	231
	Exercises 6.4	233
6.5	Horner's Rule and Binary Exponentiation	234
	Horner's Rule	234
	Binary Exponentiation	236
	Exercises 6.5	239
6.6	Problem Reduction	240
	Computing the Least Common Multiple	241
	Counting Paths in a Graph	242
	Reduction of Optimization Problems	243
	Linear Programming	244
	Reduction to Graph Problems	246
	Exercises 6.6	248
	Summary	250
7	Space and Time Trade-Offs	253
7.1	Sorting by Counting	254
	Exercises 7.1	257
7.2	Input Enhancement in String Matching	258
	Horspool's Algorithm	259

Boyer-Moore Algorithm	263
Exercises 7.2	267
7.3 Hashing	269
Open Hashing (Separate Chaining)	270
Closed Hashing (Open Addressing)	272
Exercises 7.3	274
7.4 B-Trees	276
Exercises 7.4	279
Summary	280
8 Dynamic Programming	283
8.1 Three Basic Examples	285
Exercises 8.1	290
8.2 The Knapsack Problem and Memory Functions	292
Memory Functions	294
Exercises 8.2	296
8.3 Optimal Binary Search Trees	297
Exercises 8.3	303
8.4 Warshall's and Floyd's Algorithms	304
Warshall's Algorithm	304
Floyd's Algorithm for the All-Pairs Shortest-Paths Problem	308
Exercises 8.4	311
Summary	312
9 Greedy Technique	315
9.1 Prim's Algorithm	318
Exercises 9.1	322
9.2 Kruskal's Algorithm	325
Disjoint Subsets and Union-Find Algorithms	327
Exercises 9.2	331
9.3 Dijkstra's Algorithm	333
Exercises 9.3	337

9.4 Huffman Trees and Codes	338
Exercises 9.4	342
Summary	344

10 Iterative Improvement **345**

10.1 The Simplex Method	346
Geometric Interpretation of Linear Programming	347
An Outline of the Simplex Method	351
Further Notes on the Simplex Method	357
Exercises 10.1	359
10.2 The Maximum-Flow Problem	361
Exercises 10.2	371
10.3 Maximum Matching in Bipartite Graphs	372
Exercises 10.3	378
10.4 The Stable Marriage Problem	380
Exercises 10.4	383
Summary	384

11 Limitations of Algorithm Power **387**

11.1 Lower-Bound Arguments	388
Trivial Lower Bounds	389
Information-Theoretic Arguments	390
Adversary Arguments	390
Problem Reduction	391
Exercises 11.1	393
11.2 Decision Trees	394
Decision Trees for Sorting	395
Decision Trees for Searching a Sorted Array	397
Exercises 11.2	399
11.3 P, NP, and NP-Complete Problems	401
P and NP Problems	402
NP -Complete Problems	406
Exercises 11.3	409

11.4 Challenges of Numerical Algorithms	412
Exercises 11.4	419
Summary	420

12 Coping with the Limitations of Algorithm Power **423**

12.1 Backtracking	424
n -Queens Problem	425
Hamiltonian Circuit Problem	426
Subset-Sum Problem	427
General Remarks	428
Exercises 12.1	430
12.2 Branch-and-Bound	432
Assignment Problem	433
Knapsack Problem	436
Traveling Salesman Problem	438
Exercises 12.2	440
12.3 Approximation Algorithms for <i>NP</i>-Hard Problems	441
Approximation Algorithms for the Traveling Salesman Problem	443
Approximation Algorithms for the Knapsack Problem	453
Exercises 12.3	457
12.4 Algorithms for Solving Nonlinear Equations	459
Bisection Method	460
Method of False Position	464
Newton's Method	464
Exercises 12.4	467
Summary	468

Epilogue **471**

APPENDIX A

Useful Formulas for the Analysis of Algorithms	475
Properties of Logarithms	475
Combinatorics	475
Important Summation Formulas	476
Sum Manipulation Rules	476

Approximation of a Sum by a Definite Integral	477
Floor and Ceiling Formulas	477
Miscellaneous	477

APPENDIX B

Short Tutorial on Recurrence Relations	479
Sequences and Recurrence Relations	479
Methods for Solving Recurrence Relations	480
Common Recurrence Types in Algorithm Analysis	485
 References	 493
 Hints to Exercises	 503
 Index	 547

This page intentionally left blank

4

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 its smaller instance. Once such a relationship is established, it can be exploited either top down or bottom up. The former leads naturally to a recursive implementation, although, as one can see from several examples in this chapter, an ultimate implementation may well be nonrecursive. The bottom-up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the ***incremental approach***.

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 4.1), although other constant size reductions do happen occasionally.

Consider, as an example, the exponentiation problem of computing a^n where $a \neq 0$ and n is a nonnegative integer. 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

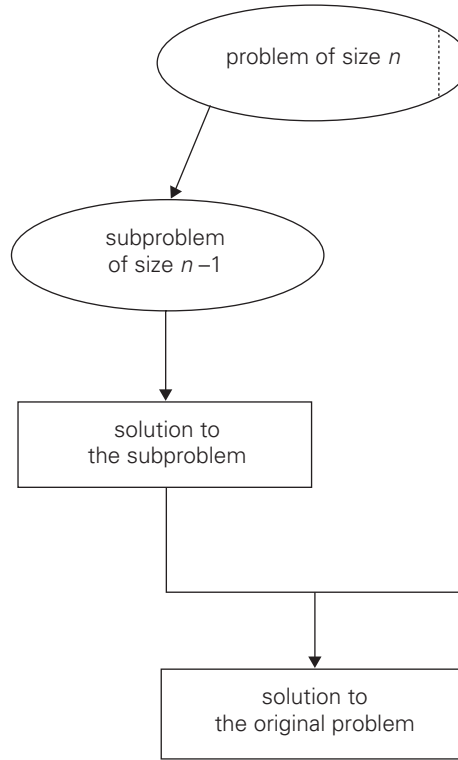


FIGURE 4.1 Decrease-(by one)-and-conquer technique.

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases} \quad (4.1)$$

or “bottom up” by multiplying 1 by a n 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 4.1–4.3.

The **decrease-by-a-constant-factor** technique suggests reducing a problem 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 4.2.

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 is to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = (a^{n/2})^2$. But since we consider here instances 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:

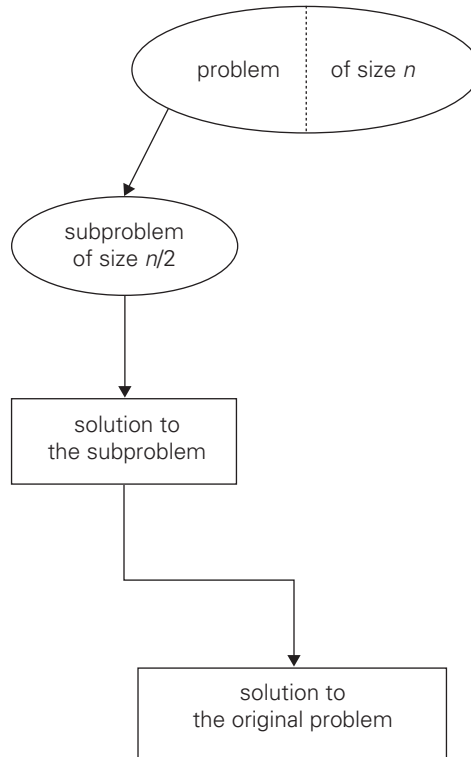


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases} \quad (4.2)$$

If we compute a^n recursively according to formula (4.2) and measure the algorithm's efficiency by the number of multiplications, we should expect the algorithm to be in $\Theta(\log n)$ because, on each iteration, the size is reduced by about a half at the expense of one or two multiplications.

A few other examples of decrease-by-a-constant-factor algorithms are given in Section 4.4 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, the 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 value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor. A few other examples of such algorithms appear in Section 4.5.

4.1 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. This is usually done by scanning the sorted subarray from right to left until the first element smaller than or equal to $A[n-1]$ is encountered to insert $A[n-1]$ right after that element. The resulting algorithm is called **straight insertion sort** or simply **insertion sort**.

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 4.3, 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).

Here is 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 $i \leftarrow 1$ **to** $n-1$ **do**
 $v \leftarrow A[i]$
 $j \leftarrow i-1$
 while $j \geq 0$ **and** $A[j] > v$ **do**
 $A[j+1] \leftarrow A[j]$
 $j \leftarrow j-1$
 $A[j+1] \leftarrow v$

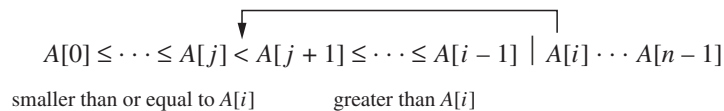


FIGURE 4.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

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 4.4 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.

The operation of the algorithm is illustrated in Figure 4.4.

The basic operation of the algorithm is the key comparison $A[j] > v$. (Why not $j \geq 0$? Because it is almost certainly 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 8 in this section’s 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 $i = 2$), \dots , $A[n - 2] > A[n - 1]$ (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

$$C_{\text{worst}}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

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 nondecreasing 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, as you are going to see in our discussion of quicksort in Chapter 5.) 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 do arise in a variety of applications, and insertion sort preserves its excellent performance on such inputs.

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 11 in this section's exercises). It shows that on randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing arrays, i.e.,

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

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 12 in this section's exercises).

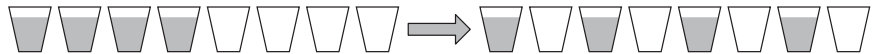
Exercises 4.1



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?



2. *Alternating glasses*
 - a. There are $2n$ glasses standing next to each other in a row, the first n of them filled with a soda drink and the remaining n glasses empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of glass moves. [Gar78]



- b. Solve the same problem if $2n$ glasses— n with a drink and n empty—are initially in a random order.



3. *Marking cells* Design an algorithm for the following task. For any even n , mark n cells on an infinite sheet of graph paper so that each marked cell has an odd number of marked neighbors. Two cells are considered neighbors if they are next to each other either horizontally or vertically but not diagonally. The marked cells must form a contiguous region, i.e., a region in which there is a path between any pair of marked cells that goes through a sequence of marked neighbors. [Kor05]

4. 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.)
5. Consider the following algorithm to check connectivity of a graph defined by its adjacency matrix.

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

//Input: Adjacency matrix $A[0..n-1, 0..n-1]$ of an undirected graph G

//Output: 1 (true) if G is connected and 0 (false) if it is not

if $n = 1$ **return** 1 //one-vertex graph is connected by definition

else

if not *Connected*($A[0..n-2, 0..n-2]$) **return** 0

else for $j \leftarrow 0$ **to** $n-2$ **do**

if $A[n-1, j]$ **return** 1

return 0

Does this algorithm work correctly for every undirected graph with $n > 0$ vertices? If you answer yes, indicate the algorithm's efficiency class in the worst case; if you answer no, explain why.



6. *Team ordering* You have the results of a completed round-robin tournament in which n teams played each other once. Each game ended either with a victory for one of the teams or with a tie. Design an algorithm that lists the teams in a sequence so that every team did not lose the game with the team listed immediately after it. What is the time efficiency class of your algorithm?
7. Apply insertion sort to sort the list E, X, A, M, P, L, E in alphabetical order.
8. **a.** What sentinel should be put before the first element of an array being sorted in order to avoid checking the in-bound condition $j \geq 0$ on each iteration of the inner loop of insertion sort?
b. Is the sentinel version in the same efficiency class as the original version?
9. Is it possible to implement insertion sort for sorting linked lists? Will it have the same $O(n^2)$ time efficiency as the array version?
10. Compare the text's implementation of insertion sort with the following version.

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 the time efficiency of this algorithm? How is it compared to that of the version given in Section 4.1?

11. Let $A[0..n-1]$ be an array of n sortable elements. (For simplicity, you may assume that all the elements are distinct.) A pair $(A[i], A[j])$ is called an ***inversion*** if $i < j$ and $A[i] > A[j]$.
 - a. What arrays of size n have the largest number of inversions and what is this number? Answer the same questions for the smallest number of inversions.
 - b. Show that the average-case number of key comparisons in insertion sort is given by the formula

$$C_{avg}(n) \approx \frac{n^2}{4}.$$

12. 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, selection sort, and bubble sort in the language of your choice and compare their performance on random arrays of sizes 10^n for $n = 2, 3, 4, 5$, and 6 as well as on increasing and decreasing arrays of these sizes.

4.2 Topological Sorting

In this section, we discuss an important problem for directed graphs, with a variety of applications involving prerequisite-restricted tasks. 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 4.5a 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.

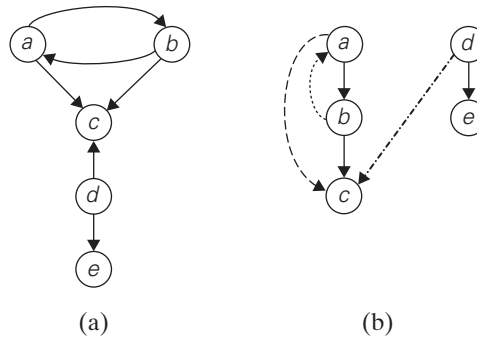


FIGURE 4.5 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a .

Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs as well, but the structure of corresponding forests can be more complex than for undirected graphs. Thus, even for the simple example of Figure 4.5a, the depth-first search forest (Figure 4.5b) 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. For example, a, b, a is a directed cycle in the digraph in Figure 4.5a. Conversely, if a DFS forest of a digraph has no back edges, the digraph is a **dag**, an acronym for **directed acyclic graph**.

Edge directions lead to new questions about digraphs that are either meaningless or trivial for undirected graphs. In this section, we discuss one such question. 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 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 4.6). 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

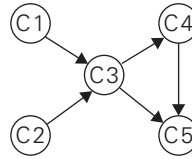


FIGURE 4.6 Digraph representing the prerequisite structure of five courses.

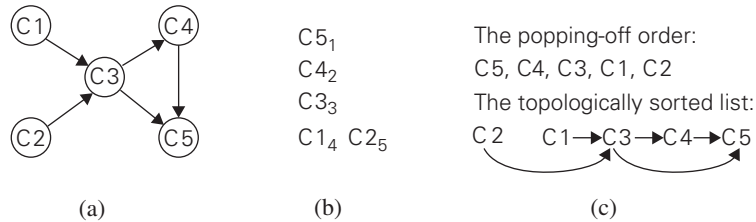


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

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 in question 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 directed 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., 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 4.7 illustrates an application of this algorithm to the digraph in Figure 4.6. Note that in Figure 4.7c, 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.

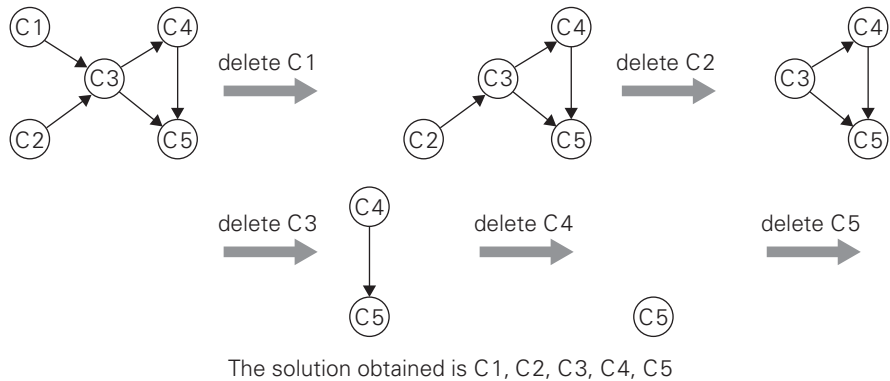


FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

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 are none, stop because the problem cannot be solved—see Problem 6a in this section’s exercises.) The order in which the vertices are deleted yields a solution to the topological sorting problem. The application of this algorithm to the same digraph representing the five courses is given in Figure 4.8.

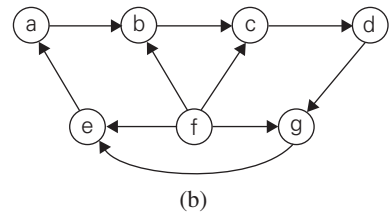
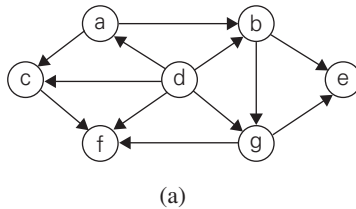
Note that the solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, of course; the topological sorting problem may have several alternative solutions.

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, research, or software development—that involves a multitude of interrelated tasks with known prerequisites. The first thing to 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 one start thinking about scheduling 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 CPM (Critical Path Method) and PERT (Program Evaluation and Review Technique) methodologies.

As to applications of topological sorting in computer science, they include instruction scheduling in program compilation, cell evaluation ordering in spreadsheet formulas, and resolving symbol dependencies in linkers.

Exercises 4.2

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 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 above.
6.
 - a. Prove that a nonempty 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. Implement the two topological sorting algorithms in the language of your choice. Run an experiment to compare their running times.
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; these subsets are called **strongly connected components** of the digraph. There are two DFS-

based algorithms for identifying strongly connected components. Here is the simpler (but somewhat less efficient) one of the two:

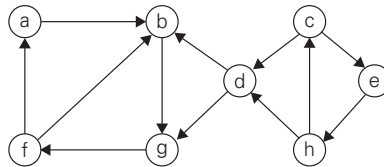
Step 1 Perform a DFS traversal of the digraph given and number its vertices in the order they become dead ends.

Step 2 Reverse the directions of all the edges of the digraph.

Step 3 Perform 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 vertices of the DFS trees obtained during the last traversal.

a. Apply this algorithm to the following digraph to determine its strongly connected components:

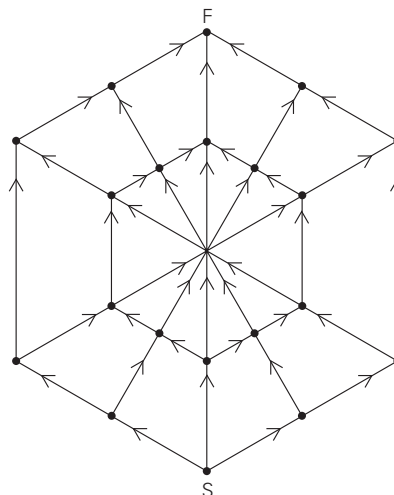


b. What is the time efficiency class of this algorithm? Give separate answers for the adjacency matrix representation and adjacency list representation of an input digraph.

c. How many strongly connected components does a dag have?



10. *Spider's web* A spider sits at the bottom (point S) of its web, and a fly sits at the top (F). How many different ways can the spider reach the fly by moving along the web's lines in the directions indicated by the arrows? [Kor05]



4.3 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 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 4.9.

The advantage of this order of generating permutations 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. (For the method being discussed, these two elements are always adjacent to each other.

start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 12 right to left	123	132	312
insert 3 into 21 left to right	321	231	213

FIGURE 4.9 Generating permutations bottom up.

Check this for the permutations generated in Figure 4.9.) 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.

ALGORITHM *JohnsonTrotter*(n)

```
//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer  $n$ 
//Output: A list of all permutations of  $\{1, \dots, n\}$ 
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  $k$  with the adjacent element  $k$ '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 element shown in bold:

$$\overleftarrow{1} \overleftarrow{2} \overleftarrow{\mathbf{3}} \quad \overleftarrow{1} \overleftarrow{\mathbf{3}} \overleftarrow{2} \quad \overleftarrow{3} \overleftarrow{1} \overleftarrow{\mathbf{2}} \quad \overrightarrow{\mathbf{3}} \overrightarrow{2} \overrightarrow{1} \quad \overrightarrow{2} \overrightarrow{\mathbf{3}} \overrightarrow{1} \quad \overrightarrow{2} \overrightarrow{1} \overrightarrow{\mathbf{3}}.$$

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; for example, 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 or-**

der—which is the order in which they would be listed in a dictionary if the numbers were interpreted as letters of an alphabet. For example, for $n = 3$,

123 132 213 231 312 321.

So how can we generate the permutation following $a_1a_2 \dots a_{n-1}a_n$ in lexicographic order? If $a_{n-1} < a_n$, which is the case for exactly one half of all the permutations, we can simply transpose these last two elements. For example, 123 is followed by 132. If $a_{n-1} > a_n$, we find the permutation's longest decreasing suffix $a_{i+1} > a_{i+2} > \dots > a_n$ (but $a_i < a_{i+1}$); increase a_i by exchanging it with the smallest element of the suffix that is greater than a_i ; and reverse the new suffix to put it in increasing order. For example, 362541 is followed by 364125. Here is pseudocode of this simple algorithm whose origins go as far back as 14th-century India.

ALGORITHM *LexicographicPermute(n)*

```
//Generates permutations in lexicographic order
//Input: A positive integer  $n$ 
//Output: A list of all permutations of  $\{1, \dots, n\}$  in lexicographic order
initialize the first permutation with  $12 \dots n$ 
while last permutation has two consecutive elements in increasing order do
    let  $i$  be its largest index such that  $a_i < a_{i+1}$  //  $a_{i+1} > a_{i+2} > \dots > a_n$ 
    find the largest index  $j$  such that  $a_i < a_j$  //  $j \geq i + 1$  since  $a_i < a_{i+1}$ 
    swap  $a_i$  with  $a_j$  //  $a_{i+1}a_{i+2} \dots a_n$  will remain in decreasing order
    reverse the order of the elements from  $a_{i+1}$  to  $a_n$  inclusive
    add the new permutation to the list
```

Generating Subsets

Recall that in Section 3.4 we examined the knapsack problem, which 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 4.10.

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 $A = \{a_1, \dots, a_n\}$

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 4.10 Generating subsets bottom up.

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., $\{a_1, a_2, a_3\}$, and 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 although the bit strings are generated by this algorithm in 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 to have the so-called **squashed order**, in which any subset involving a_j can be listed only after all the subsets involving a_1, \dots, a_{j-1} , as was the case for the list of the three-element set in Figure 4.10. It is easy to adjust the bit string–based algorithm above to yield a squashed ordering of the subsets involved (see Problem 6 in this section's exercises).

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. 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**. Frank Gray, a researcher at AT&T Bell Laboratories, reinvented it in the 1940s to minimize the effect of errors in transmitting digital signals (see, e.g., [Ros07], pp. 642–643). Seventy years earlier, the French engineer Émile Baudot used such codes

in telegraphy. Here is pseudocode that generates the binary reflected Gray code recursively.

ALGORITHM *BRGC*(n)

```
//Generates recursively the binary reflected Gray code of order  $n$ 
//Input: A positive integer  $n$ 
//Output: A list of all bit strings of length  $n$  composing the Gray code
if  $n = 1$  make list  $L$  containing bit strings 0 and 1 in this order
else generate list  $L1$  of bit strings of size  $n - 1$  by calling BRGC( $n - 1$ )
    copy list  $L1$  to list  $L2$  in reversed order
    add 0 in front of each bit string in list  $L1$ 
    add 1 in front of each bit string in list  $L2$ 
    append  $L2$  to  $L1$  to get list  $L$ 
return  $L$ 
```

The correctness of the algorithm stems from the fact that it generates 2^n bit strings and all of them are distinct. Both these assertions are easy to check by mathematical induction. Note that the binary reflected Gray code is cyclic: its last bit string differs from the first one by a single bit. For a nonrecursive algorithm for generating the binary reflected Gray code see Problem 9 in this section's exercises.

Exercises 4.3

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. Apply *LexicographicPermute* to multiset $\{1, 2, 2, 3\}$. Does it generate correctly all the permutations 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  $i \leftarrow 1$  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 the 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 pseudocode for a recursive 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. Generate the binary reflexive Gray code of order 4.
 - b. Trace the following nonrecursive algorithm to generate the binary reflexive Gray code of order 4. Start with the n -bit string of all 0's. For $i = 1, 2, \dots, 2^n - 1$, generate the i th bit string by flipping bit b in the previous bit string, where b is the position of the least significant 1 in the binary representation of i .
 10. Design a decrease-and-conquer algorithm for generating all combinations 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.
- 

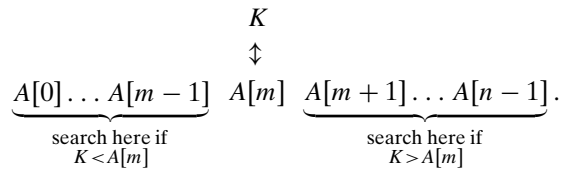
12. *Fair attraction* In olden days, one could encounter the following attraction at a fair. A light bulb was connected to several switches in such a way that it lighted up only when all the switches were closed. Each switch was controlled by a push button; pressing the button toggled the switch, but there was no way to know the state of the switch. The object was to turn the light bulb on. Design an algorithm to turn on the light bulb with the minimum number of button pushes needed in the worst case for n switches.

4.4 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. As an example of an algorithm based on this technique, we mentioned there exponentiation by squaring defined by formula (4.2). In this section, you will find a few other examples of such algorithms. The most important and well-known of them is binary search. Decrease-by-a-constant-factor algorithms usually run in logarithmic time, and, being very efficient, do not happen often; a reduction by a factor other than two is especially rare.

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	l						m						r
iteration 2								l		m			r
iteration 3								l, m	r				

Though binary search is clearly based on a recursive idea, it can be easily implemented as a nonrecursive algorithm, too. Here is pseudocode of 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, as well as some successful searches. Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for $C_{\text{worst}}(n)$:

$$C_{\text{worst}}(n) = C_{\text{worst}}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{\text{worst}}(1) = 1. \quad (4.3)$$

(Stop and convince yourself that $n/2$ must be, indeed, rounded down and that the initial condition must be written as specified.)

We already encountered recurrence (4.3), with a different initial condition, in Section 2.4 (see recurrence (2.4) and its solution there for $n = 2^k$). For the initial condition $C_{\text{worst}}(1) = 1$, we obtain

$$C_{\text{worst}}(2^k) = k + 1 = \log_2 n + 1. \quad (4.4)$$

Further, similarly to the case of recurrence (2.4) (Problem 7 in Exercises 2.4), the solution given by formula (4.4) for $n = 2^k$ can be tweaked to get a solution valid for an arbitrary positive integer n :

$$C_{\text{worst}}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil. \quad (4.5)$$

Formula (4.5) deserves attention. First, it implies that the worst-case time efficiency of binary search is in $\Theta(\log n)$. 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 (4.5), it will take no more than $\lceil \log_2(10^3 + 1) \rceil = 10$ three-way comparisons to find an element of a given value (or establish that there is no such element) in any sorted array of one thousand elements, and it will take no more than $\lceil \log_2(10^6 + 1) \rceil = 20$ comparisons to do it 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:

$$C_{avg}(n) \approx \log_2 n.$$

(More accurate formulas for the average number of comparisons in a successful and an unsuccessful search are $C_{avg}^{yes}(n) \approx \log_2 n - 1$ and $C_{avg}^{no}(n) \approx \log_2(n + 1)$, respectively.)

Though binary search is an optimal searching algorithm if we restrict our operations only to comparisons between keys (see Section 11.2), there are searching algorithms (see interpolation search in Section 4.5 and hashing in Section 7.3) with a better average-case time 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.

Fake-Coin Problem

Of several versions of the fake-coin identification problem, we consider here the one that best illustrates the decrease-by-a-constant-factor strategy. Among n identical-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 scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much. The problem is to design an efficient algorithm for detecting the fake coin. An easier version of the problem—the one we discuss here—assumes that the fake coin is known to be, say, lighter than the genuine one.¹

The most natural idea for solving this problem is to divide n coins into two piles of $\lfloor n/2 \rfloor$ coins each, leaving one extra coin aside if n is odd, and put the two

-
1. A much more challenging version assumes no additional information about the relative weights of the fake and genuine coins or even the presence of the fake coin among n given coins. We pursue this more difficult version in the exercises for Section 11.2.

piles on the scale. If the piles weigh the same, the coin put aside must be fake; otherwise, we can proceed in the same manner with the lighter pile, which must be the one with the fake coin.

We can easily set up a recurrence relation for the number of weighings $W(n)$ needed by this algorithm in the worst case:

$$W(n) = W(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad W(1) = 0.$$

This recurrence should look familiar to you. Indeed, it is almost identical to the one for the worst-case number of comparisons in binary search. (The difference is in the initial condition.) This similarity is not really surprising, since both algorithms are based on the same technique of halving an instance size. The solution to the recurrence for the number of weighings is also very similar to the one we had for binary search: $W(n) = \lfloor \log_2 n \rfloor$.

This stuff should look elementary by now, if not outright boring. But wait: the interesting point here is the fact that the above algorithm is not the most efficient solution. It would be more efficient to divide the coins not into two but into *three* piles of about $n/3$ coins each. (Details of a precise formulation are developed in this section's exercises. Do not miss it! If your instructor forgets, demand the instructor to assign Problem 10.) After weighing two of the piles, we can reduce the instance size by a factor of three. Accordingly, we should expect the number of weighings to be about $\log_3 n$, which is smaller than $\log_2 n$.

Russian Peasant Multiplication

Now we consider a nonorthodox algorithm for multiplying two positive integers called *multiplication à la russe* or the **Russian peasant method**. Let n and m be positive integers whose product we want to compute, and let us measure the instance size by the value of n . Now, if n is even, an instance of half the size has to deal with $n/2$, and we have an obvious formula relating the solution to the problem's larger instance to the solution to the smaller one:

$$n \cdot m = \frac{n}{2} \cdot 2m.$$

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 4.11. Note that all the extra addends shown in parentheses in Figure 4.11a 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 4.11b).

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

n	m		n	m	
50	65		50	65	
25	130		25	130	130
12	260	(+130)	12	260	
6	520		6	520	
3	1040		3	1040	1040
1	2080	(+1040)	1	2080	2080
	2080	+(130 + 1040) = 3250			3250

(a)
(b)

FIGURE 4.11 Computing $50 \cdot 65$ by the Russian peasant method.

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 method was known to 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.

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 4.12), 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, $J(7) = 7$.

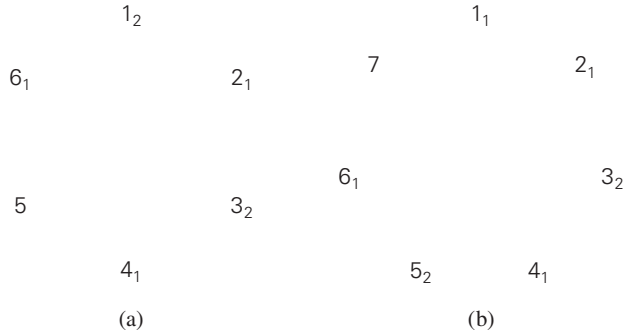


FIGURE 4.12 Instances of the Josephus problem for (a) $n = 6$ and (b) $n = 7$. Subscript numbers indicate the pass on which the person in that position is eliminated. The solutions are $J(6) = 5$ and $J(7) = 7$, respectively.

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 4.12a). It is easy to see that to get the initial position of a person, we simply need to multiply his new position by 2 and subtract 1. This relationship will hold, in particular, for the survivor, i.e.,

$$J(2k) = 2J(k) - 1.$$

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 2 and add 1 (check Figure 4.12b). Thus, for odd values of n , we get

$$J(2k + 1) = 2J(k) + 1.$$

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 [GKP94], 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 1-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 4.4



1. *Cutting a stick* A stick n inches long needs to be cut into n 1-inch pieces. Outline an algorithm that performs this task with the minimum number of cuts if several pieces of the stick can be cut at the same time. Also give a formula for the minimum number of cuts.
2. Design a decrease-by-half algorithm for computing $\lfloor \log_2 n \rfloor$ and determine its time efficiency.
3. a. What is the largest number of key comparisons made by binary search in searching for a key in the following array?

3	14	27	31	39	42	55	70	74	81	85	93	98
---	----	----	----	----	----	----	----	----	----	----	----	----

- b. List all the keys of this array that will require the largest number of key comparisons when searched for by binary search.
- c. Find the average number of key comparisons made by binary search in a successful search in this array. Assume that each key is searched for with the same probability.
- d. Find the average number of key comparisons made by binary search in an unsuccessful search in this array. Assume that searches for keys in each of the 14 intervals formed by the array's elements are equally likely.
4. Estimate how many times faster an average successful search will be in a sorted array of one million elements if it is done by binary search versus sequential search.
5. The time efficiency of sequential search does not depend on whether a list is implemented as an array or as a linked list. Is it also true for searching a sorted list by binary search?
6. a. 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.
b. Analyze the time efficiency of the two-way comparison version designed in part a.



7. *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.
8. 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.
9. An array $A[0..n-2]$ contains $n-1$ integers from 1 to n in increasing order. (Thus one integer in this range is missing.) Design the most efficient algorithm you can to find the missing integer and indicate its time efficiency.
 10. a. Write 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 .
 11. a. Apply the Russian peasant algorithm to compute $26 \cdot 47$.
 b. From the standpoint of time efficiency, does it matter whether we multiply n by m or m by n by the Russian peasant algorithm?
 12. a. Write pseudocode for the Russian peasant multiplication algorithm.
 b. What is the time efficiency class of Russian peasant multiplication?
 13. Find $J(40)$ —the solution to the Josephus problem for $n = 40$.
 14. Prove that the solution to the Josephus problem is 1 for every n that is a power of 2.
 15. 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 1-bit cyclic shift left of the binary representation of n .

4.5 Variable-Size-Decrease Algorithms

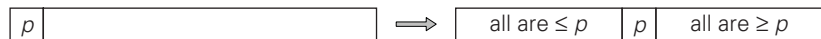
In the third principal variety of decrease-and-conquer, the size reduction pattern varies from one iteration of the algorithm to another. Euclid's algorithm for computing the greatest common divisor (Section 1.1) provides a good example

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 not larger than one half of the list's elements and not smaller than the other half. This middle value is called the **median**, and it is one of the most important notions 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 fast sorting algorithm such as mergesort (discussed in the next chapter), 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. Indeed, we can take advantage of the idea of **partitioning** a given list around some value p of, say, its first element. In general, this is a rearrangement of the list's elements so that the left part contains all the elements smaller than or equal to p , followed by the **pivot** p itself, followed by all the elements greater than or equal to p .



Of the two principal algorithmic alternatives to partition an array, here we discuss the **Lomuto partitioning** [Ben00, p. 117]; we introduce the better known Hoare's algorithm in the next chapter. To get the idea behind the Lomuto partitioning, it is helpful to think of an array—or, more generally, a subarray $A[l..r]$ ($0 \leq l \leq r \leq n - 1$)—under consideration as composed of three contiguous segments. Listed in the order they follow pivot p , they are as follows: a segment with elements known to be smaller than p , the segment of elements known to be greater than or equal to p , and the segment of elements yet to be compared to p (see Figure 4.13a). Note that the segments can be empty; for example, it is always the case for the first two segments before the algorithm starts.

Starting with $i = l + 1$, the algorithm scans the subarray $A[l..r]$ left to right, maintaining this structure until a partition is achieved. On each iteration, it compares the first element in the unknown segment (pointed to by the scanning index i in Figure 4.13a) with the pivot p . If $A[i] \geq p$, i is simply incremented to expand the segment of the elements greater than or equal to p while shrinking the unprocessed segment. If $A[i] < p$, it is the segment of the elements smaller than p that needs to be expanded. This is done by incrementing s , the index of the last

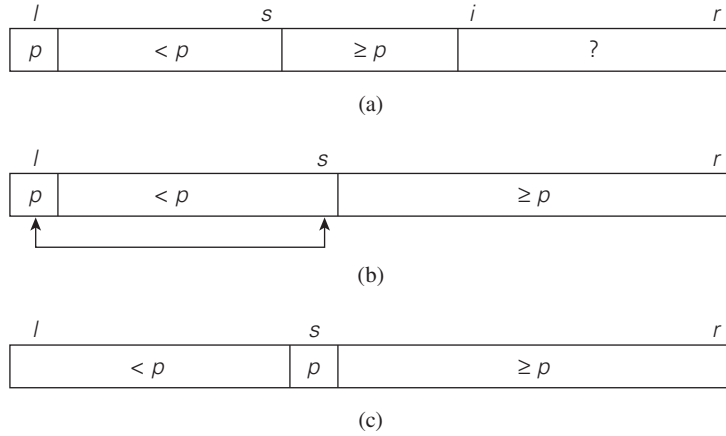


FIGURE 4.13 Illustration of the Lomuto partitioning.

element in the first segment, swapping $A[i]$ and $A[s]$, and then incrementing i to point to the new first element of the shrunk unprocessed segment. After no unprocessed elements remain (Figure 4.13b), the algorithm swaps the pivot with $A[s]$ to achieve a partition being sought (Figure 4.13c).

Here is pseudocode implementing this partitioning procedure.

ALGORITHM *LomutoPartition*($A[l..r]$)

```
//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray  $A[l..r]$  of array  $A[0..n-1]$ , defined by its left and right
//       indices  $l$  and  $r$  ( $l \leq r$ )
//Output: Partition of  $A[l..r]$  and the new position of the pivot
 $p \leftarrow A[l]$ 
 $s \leftarrow l$ 
for  $i \leftarrow l + 1$  to  $r$  do
    if  $A[i] < p$ 
         $s \leftarrow s + 1$ ;  $\text{swap}(A[s], A[i])$ 
 $\text{swap}(A[l], A[s])$ 
return  $s$ 
```

How can we take advantage of a list partition to find the k th smallest element in it? Let us assume that the list is implemented as an array whose elements are indexed starting with a 0, and let s be the partition's split position, i.e., the index of the array's element occupied by the pivot after partitioning. If $s = k - 1$, pivot p itself is obviously the k th smallest element, which solves the problem. If $s > k - 1$, the k th smallest element in the entire array can be found as the k th smallest element in the left part of the partitioned array. And if $s < k - 1$, it can

be found as 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. This algorithm is called **quickselect**.

To find the k th smallest element in array $A[0..n - 1]$ by this algorithm, call $Quickselect(A[0..n - 1], k)$ where

ALGORITHM $Quickselect(A[l..r], k)$

```
//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray  $A[l..r]$  of array  $A[0..n - 1]$  of orderable elements and
//      integer  $k$  ( $1 \leq k \leq r - l + 1$ )
//Output: The value of the  $k$ th smallest element in  $A[l..r]$ 
 $s \leftarrow LomutoPartition(A[l..r])$  //or another partition algorithm
if  $s = k - 1$  return  $A[s]$ 
else if  $s > l + k - 1$   $Quickselect(A[l..s - 1], k)$ 
else  $Quickselect(A[s + 1..r], k - 1 - s)$ 
```

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 $s = k - 1$.

EXAMPLE Apply the partition-based algorithm to find the median of the following list of nine numbers: 4, 1, 10, 8, 7, 12, 9, 2, 15. Here, $k = \lceil 9/2 \rceil = 5$ and our task is to find the 5th smallest element in the array.

We use the above version of array partitioning, showing the pivots in bold.

	0	1	2	3	4	5	6	7	8
s		i							
4		1	10	8	7	12	9	2	15
		s	i						
4		1	10	8	7	12	9	2	15
		s						i	
4		1	10	8	7	12	9	2	15
			s					i	
4		1	2	8	7	12	9	10	15
			s						i
4		1	2	8	7	12	9	10	15
2		1	4	8	7	12	9	10	15

Since $s = 2$ is smaller than $k - 1 = 4$, we proceed with the right part of the array:

0	1	2	3	4	5	6	7	8
<hr/>								
			<i>s</i>	<i>i</i>				
			8	7	12	9	10	15
			<i>s</i>	<i>i</i>				
			8	7	12	9	10	15
			<i>s</i>					<i>i</i>
			8	7	12	9	10	15
			7	8	12	9	10	15

Now $s = k - 1 = 4$, and hence we can stop: the found median is 8, which is greater than 2, 1, 4, and 7 but smaller than 12, 9, 10, and 15. ■

How efficient is quickselect? Partitioning an n -element array always requires $n - 1$ key comparisons. If it produces the split that solves the selection problem without requiring more iterations, then for this best case we obtain $C_{best}(n) = n - 1 \in \Theta(n)$. Unfortunately, the algorithm can produce an extremely unbalanced partition of a given array, with one part being empty and the other containing $n - 1$ elements. In the worst case, this can happen on each of the $n - 1$ iterations. (For a specific example of the worst-case input, consider, say, the case of $k = n$ and a strictly increasing array.) This implies that

$$C_{worst}(n) = (n - 1) + (n - 2) + \cdots + 1 = (n - 1)n/2 \in \Theta(n^2),$$

which compares poorly with the straightforward sorting-based approach mentioned in the beginning of our selection problem discussion. Thus, the usefulness of the partition-based algorithm depends on the algorithm's efficiency in the average case. Fortunately, a careful mathematical analysis has shown that the average-case efficiency is linear. In fact, computer scientists have discovered a more sophisticated way of choosing a pivot in quickselect that guarantees linear time even in the worst case [Blo73], but it is too complicated to be recommended for practical applications.

It is also worth noting that the partition-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

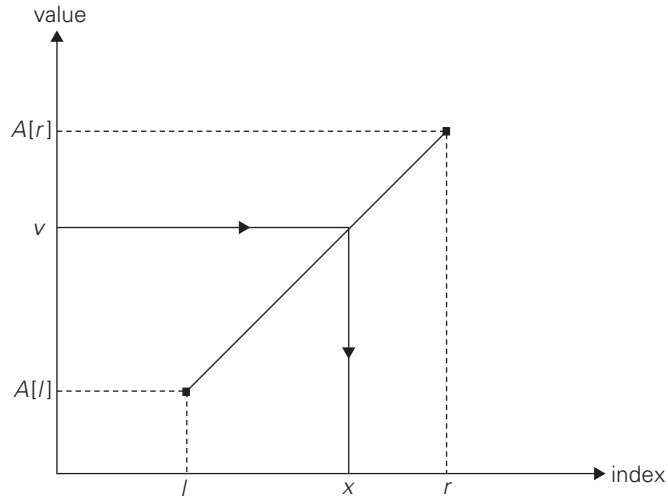


FIGURE 4.14 Index computation in interpolation search.

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 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 round-off 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 4.14).

Writing down 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:

$$x = l + \left\lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \right\rfloor. \quad (4.6)$$

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 these values increased linearly, which is the simplest manner possible, the index computed by formula (4.4) 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 (4.4) need not be applied (why?).

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 is a very small constant for all practically feasible inputs (see Problem 6 in this section's exercises). But in the worst case, interpolation search is only linear, which must be considered a bad performance (why?).

Assessing the worthiness of interpolation search versus that of binary search, Robert Sedgewick wrote in the second edition of his *Algorithms* 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

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 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 the 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 4.15).

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 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.

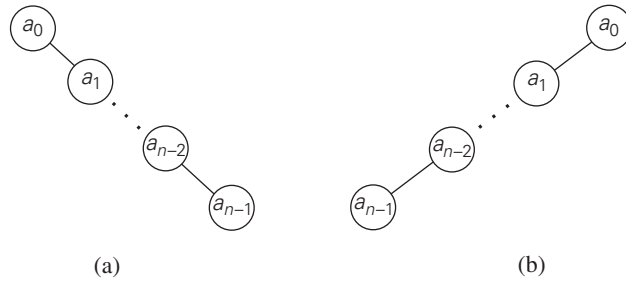


FIGURE 4.15 Binary search trees for (a) an increasing sequence of keys and (b) a decreasing sequence of keys.

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, i.e., 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 4.16.) 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

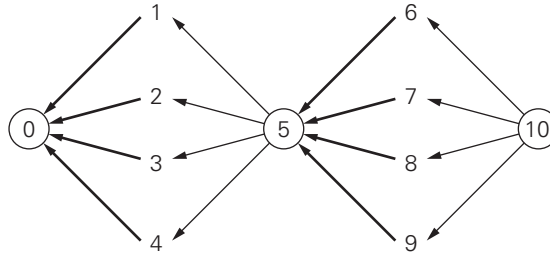


FIGURE 4.16 Illustration of one-pile Nim with the maximum number of chips that may be taken on each move $m = 4$. The numbers indicate n , the number of chips in the pile. The losing positions for the player to move are circled. Only winning moves from the winning positions are shown (in bold).

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 $n \bmod (m + 1)$ 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, 10, and then his opponent and he take turns adding any integer less than 10; 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**, also known as the **nim 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 nim sum contains at least one 1; consequently, Nim's instance is a losing instance if and only if its nim sum contains only zeros. For example, for the commonly played instance with $n_1 = 3$, $n_2 = 4$, $n_3 = 5$, the nim sum is

$$\begin{array}{r}
 011 \\
 100 \\
 \underline{101} \\
 010
 \end{array}$$

Since this sum contains a 1, the instance is a winning one for the player 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 nim 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 4.5

1. **a.** If we measure an instance size of computing the greatest common divisor of m and n by the size of the second number n , by how much can the size decrease after one iteration of Euclid's algorithm?
b. Prove that an instance size will always decrease at least by a factor of two after two successive iterations of Euclid's algorithm.
2. Apply quickselect to find the median of the list of numbers 9, 12, 5, 17, 20, 30, 8.
3. Write pseudocode for a nonrecursive implementation of quickselect.
4. Derive the formula underlying interpolation search.
5. Give an example of the worst-case input for interpolation search and show that the algorithm is linear in the worst case.
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 largest 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. Outline a variable-size-decrease algorithm for constructing an Eulerian circuit in a connected graph with all vertices of even degrees.



10. *Misère one-pile Nim* Consider the so-called ***misère 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 losing positions (for the player to move next) in this game.



11. a. *Moldy chocolate* Two players take turns by breaking an $m \times n$ chocolate bar, which has one spoiled 1×1 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.



12. *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 stack 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.
13. You need to search for a given number in an $n \times n$ matrix in which every row and every column is sorted in increasing order. Can you design a $O(n)$ algorithm for this problem? [Laa10]

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 (usually recursively) or bottom up.
- 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.

- 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 a direct application of the decrease-by-one technique.
- The decrease-by-one technique is a natural approach to developing algorithms for generating elementary combinatorial objects. The most efficient class 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.
- *Binary search* is a very efficient algorithm for searching in a sorted array. It is a principal example of a decrease-by-a-constant-factor algorithm. Other examples include *exponentiation by squaring*, identifying a fake coin with a balance scale, *Russian peasant multiplication*, and the *Josephus problem*.
- For some decrease-and-conquer algorithms, 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.