Introduction to

The Design & Analysis of Algorithms

3RD EDITION

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

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

This page intentionally left blank

Contents

This page intentionally left blank

2

Fundamentals of the Analysis of Algorithm Efficiency

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

—Lord Kelvin (1824–1907)

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

—Albert Einstein (1879–1955)

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

We start with a general framework for analyzing algorithm efficiency in Section 2.1. This section is arguably the most important in the chapter; the fundamental nature of the topic makes it also one of the most important sections in the entire book.

In Section 2.2, we introduce three notations: O ("big oh"), Ω ("big omega"), and Θ ("big theta"). Borrowed from mathematics, these notations have become *the* language for discussing the efficiency of algorithms.

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

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

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

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

2.1 The Analysis Framework

In this section, we outline a general framework for analyzing the efficiency of algorithms. We already mentioned in Section 1.2 that there are two kinds of efficiency: time efficiency and space efficiency. ***Time efficiency***, also called ***time complexity***, indicates how fast an algorithm in question runs. ***Space efficiency***, also called ***space complexity***, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output. In the early days of electronic computing, both resources—time and space—were at a premium. Half a century

of relentless technological innovations have improved the computer's speed and memory size by many orders of magnitude. Now the amount of extra space required by an algorithm is typically not of as much concern, with the caveat that there is still, of course, a difference between the fast main memory, the slower secondary memory, and the cache. The time issue has not diminished quite to the same extent, however. In addition, the research experience has shown that for most problems, we can achieve much more spectacular progress in speed than in space. Therefore, following a well-established tradition of algorithm textbooks, we primarily concentrate on time efficiency, but the analytical framework introduced here is applicable to analyzing space efficiency as well.

Measuring an Input's Size

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

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

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

We should make a special note about measuring input size for algorithms solving problems such as checking primality of a positive integer n. Here, the input is just one number, and it is this number's magnitude that determines the input

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

size. In such situations, it is preferable to measure size by the number b of bits in the n's binary representation:

$$b = \lfloor \log_2 n \rfloor + 1. \quad \textbf{(2.1)}$$

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

Units for Measuring Running Time

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

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

As a rule, it is not difficult to identify the basic operation of an algorithm: it is usually the most time-consuming operation in the algorithm's innermost loop. For example, most sorting algorithms work by comparing elements (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key comparison. As another example, algorithms for mathematical problems typically involve some or all of the four arithmetical operations: addition, subtraction, multiplication, and division. Of the four, the most time-consuming operation is division, followed by multiplication and then addition and subtraction, with the last two usually considered together.[2]

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

Here is an important application. Let c_{op} be the execution time of an algorithm's basic operation on a particular computer, and let $C(n)$ be the number of times this operation needs to be executed for this algorithm. Then we can estimate

2. On some computers, multiplication does not take longer than addition/subtraction (see, for example, the timing data provided by Kernighan and Pike in [Ker99, pp. 185–186]).

the running time $T(n)$ of a program implementing this algorithm on that computer by the formula

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

Of course, this formula should be used with caution. The count $C(n)$ does not contain any information about operations that are not basic, and, in fact, the count itself is often computed only approximately. Further, the constant c_{op} is also an approximation whose reliability is not always easy to assess. Still, unless n is extremely large or very small, the formula can give a reasonable estimate of the algorithm's running time. It also makes it possible to answer such questions as "How much faster would this algorithm run on a machine that is 10 times faster than the one we have?" The answer is, obviously, 10 times. Or, assuming that $C(n) = \frac{1}{2}n(n-1)$, how much longer will the algorithm run if we double its input size? The answer is about four times longer. Indeed, for all but very small values of n,

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

and therefore

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

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

Orders of Growth

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

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

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

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

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

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

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

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

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

Another way to appreciate the qualitative difference among the orders of growth of the functions in Table 2.1 is to consider how they react to, say, a twofold increase in the value of their argument n. The function $\log_2 n$ increases in value by just 1 (because $\log_2 2n = \log_2 2 + \log_2 n = 1 + \log_2 n$); the linear function increases twofold, the linearithmic function $n \log_2 n$ increases slightly more than twofold; the quadratic function n^2 and cubic function n^3 increase fourfold and

eightfold, respectively (because $(2n)^2 = 4n^2$ and $(2n)^3 = 8n^3$); the value of 2^n gets squared (because $2^{2n} = (2^n)^2$); and $n!$ increases much more than that (yes, even mathematics refuses to cooperate to give a neat answer for $n!$).

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

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

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

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

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

The ***worst-case efficiency*** of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size. The way to determine the worst-case efficiency of an algorithm is, in principle, quite straightforward: analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count $C(n)$ among all possible inputs of size n and then compute this worst-case value $C_{worst}(n)$. (For sequential search, the answer was obvious. The methods for handling less trivial situations are explained in subsequent sections of this chapter.) Clearly, the worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other

words, it guarantees that for any instance of size n, the running time will not exceed $C_{worst}(n)$, its running time on the worst-case inputs.

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

The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency. But it is not completely useless, either. Though we should not expect to get best-case inputs, we might be able to take advantage of the fact that for some algorithms a good best-case performance extends to some useful types of inputs close to being the best-case ones. For example, there is a sorting algorithm (insertion sort) for which the best-case inputs are already sorted arrays on which the algorithm works very fast. Moreover, the best-case efficiency deteriorates only slightly for almost-sorted arrays. Therefore, such an algorithm might well be the method of choice for applications dealing with almost-sorted arrays. And, of course, if the best-case efficiency of an algorithm is unsatisfactory, we can immediately discard it without further analysis.

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

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

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

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

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

The technical implementation of this plan is rarely easy, however, and probabilistic assumptions underlying it in each particular case are usually difficult to verify. Given our quest for simplicity, we will mostly quote known results about the average-case efficiency of algorithms under discussion. If you are interested in derivations of these results, consult such books as [Baa00], [Sed96], [KnuI], [KnuII], and [KnuIII].

It should be clear from the preceding discussion that the average-case efficiency cannot be obtained by taking the average of the worst-case and the best-case efficiencies. Even though this average does occasionally coincide with the average-case cost, it is not a legitimate way of performing the average-case analysis.

Does one really need the average-case efficiency information? The answer is unequivocally yes: there are many important algorithms for which the average-case efficiency is much better than the overly pessimistic worst-case efficiency would lead us to believe. So, without the average-case analysis, computer scientists could have missed many important algorithms.

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

Recapitulation of the Analysis Framework

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

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

In the next section, we look at formal means to investigate orders of growth. In Sections 2.3 and 2.4, we discuss particular methods for investigating nonrecursive and recursive algorithms, respectively. It is there that you will see how the analysis framework outlined here can be applied to investigating the efficiency of specific algorithms. You will encounter many more examples throughout the rest of the book.

Exercises 2.1

1. For each of the following algorithms, indicate (i) a natural size metric for its inputs, (ii) its basic operation, and (iii) whether the basic operation count can be different for inputs of the same size:
   a. computing the sum of n numbers
   b. computing $n!$
   c. finding the largest element in a list of n numbers
   d. Euclid's algorithm
   e. sieve of Eratosthenes
   f. pen-and-pencil algorithm for multiplying two n-digit decimal integers
2. **a.** Consider the definition-based algorithm for adding two $n \times n$ matrices. What is its basic operation? How many times is it performed as a function of the matrix order n? As a function of the total number of elements in the input matrices?
   b. Answer the same questions for the definition-based algorithm for matrix multiplication.

3. Consider a variation of sequential search that scans a list to return the number of occurrences of a given search key in the list. Does its efficiency differ from the efficiency of classic sequential search?

4. a. *Glove selection* There are 22 gloves in a drawer: 5 pairs of red gloves, 4 pairs of yellow, and 2 pairs of green. You select the gloves in the dark and can check them only after a selection has been made. What is the smallest number of gloves you need to select to have at least one matching pair in the best case? In the worst case?

b. *Missing socks* Imagine that after washing 5 distinct pairs of socks, you discover that two socks are missing. Of course, you would like to have the largest number of complete pairs remaining. Thus, you are left with 4 complete pairs in the best-case scenario and with 3 complete pairs in the worst case. Assuming that the probability of disappearance for each of the 10 socks is the same, find the probability of the best-case scenario; the probability of the worst-case scenario; the number of pairs you should expect in the average case.

5. a. Prove formula (2.1) for the number of bits in the binary representation of a positive decimal integer.

b. Prove the alternative formula for the number of bits in the binary representation of a positive integer n:

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

c. What would be the analogous formulas for the number of decimal digits?

d. Explain why, within the accepted analysis framework, it does not matter whether we use binary or decimal digits in measuring n's size.

6. Suggest how any sorting algorithm can be augmented in a way to make the best-case count of its key comparisons equal to just $n - 1$ (n is a list's size, of course). Do you think it would be a worthwhile addition to any sorting algorithm?

7. Gaussian elimination, the classic algorithm for solving systems of n linear equations in n unknowns, requires about $\frac{1}{3}n^3$ multiplications, which is the algorithm's basic operation.

a. How much longer should you expect Gaussian elimination to work on a system of 1000 equations versus a system of 500 equations?

b. You are considering buying a computer that is 1000 times faster than the one you currently have. By what factor will the faster computer increase the sizes of systems solvable in the same amount of time as on the old computer?

8. For each of the following functions, indicate how much the function's value will change if its argument is increased fourfold.

a. $\log_2 n$ **b.** $\sqrt{n}$ **c.** n **d.** n^2 **e.** n^3 **f.** 2^n

9. For each of the following pairs of functions, indicate whether the first function of each of the following pairs has a lower, same, or higher order of growth (to within a constant multiple) than the second function.

a. $n(n+1)$ and $2000n^2$ **b.** $100n^2$ and $0.01n^3$

c. $\log_2 n$ and $\ln n$ **d.** $\log_2^2 n$ and $\log_2 n^2$

e. 2^{n-1} and 2^n **f.** $(n-1)!$ and $n!$

10. *Invention of chess*

a. According to a well-known legend, the game of chess was invented many centuries ago in northwestern India by a certain sage. When he took his invention to his king, the king liked the game so much that he offered the inventor any reward he wanted. The inventor asked for some grain to be obtained as follows: just a single grain of wheat was to be placed on the first square of the chessboard, two on the second, four on the third, eight on the fourth, and so on, until all 64 squares had been filled. If it took just 1 second to count each grain, how long would it take to count all the grain due to him?

b. How long would it take if instead of doubling the number of grains for each square of the chessboard, the inventor asked for adding two grains?

2.2 Asymptotic Notations and Basic Efficiency Classes

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

Informal Introduction

Informally, $O(g(n))$ is the set of all functions with a lower or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). Thus, to give a few examples, the following assertions are all true:

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

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

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

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

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

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

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

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

O-notation

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

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

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

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

$$100n + 5 \le 100n + n \text{ (for all } n \ge 5) = 101n \le 101n^2.$$

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

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

$$100n + 5 \le 100n + 5n \text{ (for all } n \ge 1) = 105n$$

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

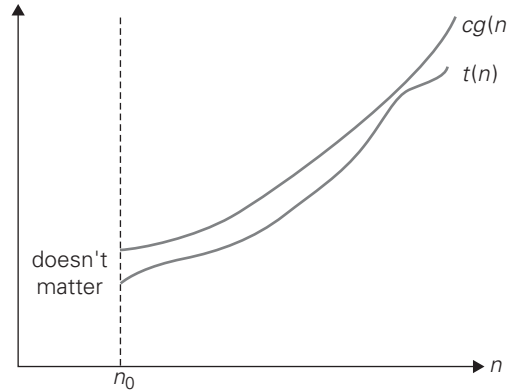


FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$.

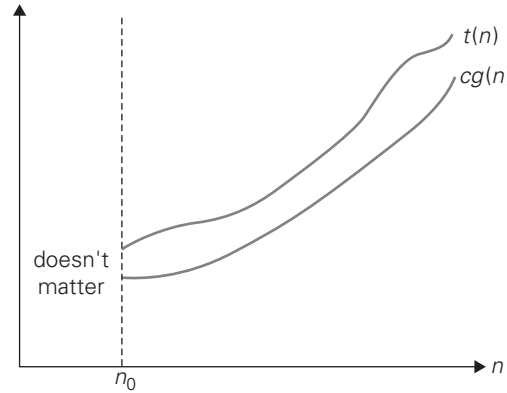


FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$.

Ω-notation

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

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

The definition is illustrated in Figure 2.2.

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

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

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

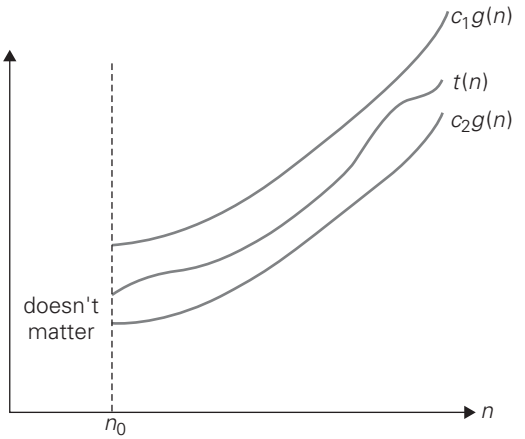


FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$.

Θ-notation

DEFINITION A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

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

The definition is illustrated in Figure 2.3.

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

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \le \frac{1}{2}n^2 \quad \text{for all } n \ge 0.$$

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

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

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

Useful Property Involving the Asymptotic Notations

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

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

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

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

PROOF The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2 \max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

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

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

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

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

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

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

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

$$\left.\begin{array}{|c|}\hline t_1(n) \in O(g_1(n)) \\ \hline t_2(n) \in O(g_2(n)) \\ \hline \end{array}\right\} \quad t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

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

Using Limits for Comparing Orders of Growth

Though the formal definitions of O, Ω, and Θ are indispensable for proving their abstract properties, they are rarely used for comparing the orders of growth of two specific functions. A much more convenient method for doing so is based on

computing the limit of the ratio of two functions in question. Three principal cases may arise:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3 \end{cases}$$

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

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

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

and Stirling's formula

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

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

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

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

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

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

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

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

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

EXAMPLE 3 Compare the orders of growth of $n!$ and 2^n. (We discussed this informally in Section 2.1.) Taking advantage of Stirling's formula, we get

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

Thus, though 2^n grows very fast, $n!$ grows still faster. We can write symbolically that $n! \in \Omega(2^n)$; note, however, that while the big-Omega notation does not preclude the possibility that $n!$ and 2^n have the same order of growth, the limit computed here certainly does. ■

Basic Efficiency Classes

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

You could raise a concern that classifying algorithms by their asymptotic efficiency would be of little practical use since the values of multiplicative constants are usually left unspecified. This leaves open the possibility of an algorithm in a worse efficiency class running faster than an algorithm in a better efficiency class for inputs of realistic sizes. For example, if the running time of one algorithm is n^3 while the running time of the other is $10^6 n^2$, the cubic algorithm will outperform the quadratic algorithm unless n exceeds 10^6. A few such anomalies are indeed known. Fortunately, multiplicative constants usually do not differ that drastically. As a rule, you should expect an algorithm from a better asymptotic efficiency class to outperform an algorithm from a worse class even for moderately sized inputs. This observation is especially true for an algorithm with a better than exponential running time versus an exponential (or worse) algorithm.

Exercises 2.2

1. Use the most appropriate notation among O, Θ, and Ω to indicate the time efficiency class of sequential search (see Section 2.1)
   a. in the worst case.
   b. in the best case.
   c. in the average case.
2. Use the informal definitions of O, Θ, and Ω to determine whether the following assertions are true or false.

TABLE 2.2 Basic asymptotic efficiency classes

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

a. $n(n+1)/2 \in O(n^3)$ **b.** $n(n+1)/2 \in O(n^2)$
c. $n(n+1)/2 \in \Theta(n^3)$ **d.** $n(n+1)/2 \in \Omega(n)$

3. For each of the following functions, indicate the class $\Theta(g(n))$ the function belongs to. (Use the simplest $g(n)$ possible in your answers.) Prove your assertions.

a. $(n^2+1)^{10}$ **b.** $\sqrt{10n^2+7n+3}$
c. $2n \lg(n+2)^2 + (n+2)^2 \lg \frac{n}{2}$ **d.** $2^{n+1} + 3^{n-1}$
e. $\lfloor \log_2 n \rfloor$

4. a. Table 2.1 contains values of several functions that often arise in the analysis of algorithms. These values certainly suggest that the functions

$$\log n, \quad n, \quad n \log_2 n, \quad n^2, \quad n^3, \quad 2^n, \quad n!$$

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

b. Prove that the functions are indeed listed in increasing order of their order of growth.

5. List the following functions according to their order of growth from the lowest to the highest:

$$(n-2)!, \quad 5 \lg(n+100)^{10}, \quad 2^{2n}, \quad 0.001n^4 + 3n^3 + 1, \quad \ln^2 n, \quad \sqrt[3]{n}, \quad 3^n.$$

6. a. Prove that every polynomial of degree k, $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$ with $a_k > 0$, belongs to $\Theta(n^k)$.

b. Prove that exponential functions a^n have different orders of growth for different values of base $a > 0$.

7. Prove the following assertions by using the definitions of the notations involved, or disprove them by giving a specific counterexample.

a. If $t(n) \in O(g(n))$, then $g(n) \in \Omega(t(n))$.

b. $\Theta(\alpha g(n)) = \Theta(g(n))$, where $\alpha > 0$.

c. $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

d. For any two nonnegative functions $t(n)$ and $g(n)$ defined on the set of nonnegative integers, either $t(n) \in O(g(n))$, or $t(n) \in \Omega(g(n))$, or both.

8. Prove the section's theorem for

a. Ω notation. **b.** Θ notation.

9. We mentioned in this section that one can check whether all elements of an array are distinct by a two-part algorithm based on the array's presorting.

a. If the presorting is done by an algorithm with a time efficiency in $\Theta(n \log n)$, what will be a time-efficiency class of the entire algorithm?

b. If the sorting algorithm used for presorting needs an extra array of size n, what will be the space-efficiency class of the entire algorithm?

10. The ***range*** of a finite nonempty set of n real numbers S is defined as the difference between the largest and smallest elements of S. For each representation of S given below, describe in English an algorithm to compute the range. Indicate the time efficiency classes of these algorithms using the most appropriate notation (O, Θ, or Ω).

a. An unsorted array

b. A sorted array

c. A sorted singly linked list

d. A binary search tree

11. *Lighter or heavier?* You have $n > 2$ identical-looking coins and a two-pan balance scale with no weights. One of the coins is a fake, but you do not know whether it is lighter or heavier than the genuine coins, which all weigh the same. Design a $\Theta(1)$ algorithm to determine whether the fake coin is lighter or heavier than the others.

12. *Door in a wall* You are facing a wall that stretches infinitely in both directions. There is a door in the wall, but you know neither how far away nor in which direction. You can see the door only when you are right next to it. Design an algorithm that enables you to reach the door by walking at most $O(n)$ steps where n is the (unknown to you) number of steps between your initial position and the door. [Par95]

2.3 Mathematical Analysis of Nonrecursive Algorithms

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

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

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

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

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

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

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

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

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

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

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.[4]
5. Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.

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

$$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

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

and two summation formulas

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

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

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

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

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

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

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

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

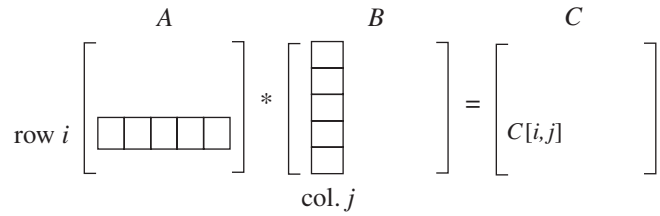
$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$
$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2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

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

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2},$$

where the last equality is obtained by applying summation formula (S2). Note that this result was perfectly predictable: in the worst case, the algorithm needs to compare all $n(n-1)/2$ distinct pairs of its n elements. ■

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



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

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

```
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

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

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

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

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

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

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

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

This example is simple enough so that we could get this result without all the summation machinations. How? The algorithm computes n^2 elements of the product matrix. Each of the product's elements is computed as the scalar (dot) product of an n-element row of the first matrix and an n-element column of the second matrix, which takes n multiplications. So the total number of multiplications is $n \cdot n^2 = n^3$. (It is this kind of reasoning that we expected you to employ when answering this question in Problem 2 of Exercises 2.1.)

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

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

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

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

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

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

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

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

ALGORITHM *Binary*(n)

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

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

A more significant feature of this example is the fact that the loop variable takes on only a few values between its lower and upper limits; therefore, we have to use an alternative way of computing the number of times the loop is executed. Since the value of n is about halved on each repetition of the loop, the answer should be about $\log_2 n$. The exact formula for the number of times the comparison $n > 1$ will be executed is actually $\lfloor \log_2 n \rfloor + 1$—the number of bits in the binary representation of n according to formula (2.1). We could also get this answer by applying the analysis technique based on recurrence relations; we discuss this technique in the next section because it is more pertinent to the analysis of recursive algorithms. ■

Exercises 2.3

1. Compute the following sums.

a. $1 + 3 + 5 + 7 + \cdots + 999$

b. $2 + 4 + 8 + 16 + \cdots + 1024$

c. $\sum_{i=3}^{n+1} 1$ **d.** $\sum_{i=3}^{n+1} i$ **e.** $\sum_{i=0}^{n-1} i(i+1)$

f. $\sum_{j=1}^{n} 3^{j+1}$ **g.** $\sum_{i=1}^{n} \sum_{j=1}^{n} ij$ **h.** $\sum_{i=1}^{n} 1/i(i+1)$

2. Find the order of growth of the following sums. Use the $\Theta(g(n))$ notation with the simplest function $g(n)$ possible.

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

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

3. The sample variance of n measurements $x_1, \ldots, x_n$ can be computed as either

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

or

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

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

4. Consider the following algorithm.

ALGORITHM *Mystery*(n)

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

a. What does this algorithm compute?

b. What is its basic operation?

c. How many times is the basic operation executed?

d. What is the efficiency class of this algorithm?

e. Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

5. Consider the following algorithm.

ALGORITHM *Secret*$(A[0..n-1])$
//Input: An array $A[0..n-1]$ of n real numbers
$minval \leftarrow A[0]$; $maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $A[i] < minval$
    $minval \leftarrow A[i]$
  if $A[i] > maxval$
    $maxval \leftarrow A[i]$
return $maxval - minval$

Answer questions (a)–(e) of Problem 4 about this algorithm.

6. Consider the following algorithm.

ALGORITHM *Enigma*$(A[0..n-1, 0..n-1])$
//Input: A matrix $A[0..n-1, 0..n-1]$ of real numbers
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i, j] \neq A[j, i]$
      return false
return true

Answer questions (a)–(e) of Problem 4 about this algorithm.

7. Improve the implementation of the matrix multiplication algorithm (see Example 3) by reducing the number of additions made by the algorithm. What effect will this change have on the algorithm's efficiency?

8. Determine the asymptotic order of growth for the total number of times all the doors are toggled in the locker doors puzzle (Problem 12 in Exercises 1.1).

9. Prove the formula

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

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

10. *Mental arithmetic* A 10×10 table is filled with repeating numbers on its diagonals as shown below. Calculate the total sum of the table's numbers in your head (after [Cra07, Question 1.33]).

1	2	3			⋯			9	10
2	3						9	10	11
3						9	10	11	
					9	10	11		
				9	10	11			
⋮			9	10	11				⋮
		9	10	11					
	9	10	11						17
9	10	11						17	18
10	11				⋯		17	18	19

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

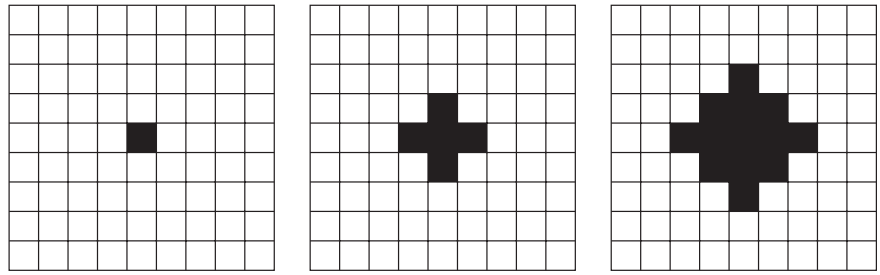
ALGORITHM $GE(A[0..n-1, 0..n])$

```
//Input: An n × (n + 1) matrix A[0..n − 1, 0..n] of real numbers
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        for k ← i to n do
            A[j, k] ← A[j, k] − A[i, k] ∗ A[j, i] / A[i, i]
```

a. Find the time efficiency class of this algorithm.

b. What glaring inefficiency does this pseudocode contain and how can it be eliminated to speed the algorithm up?

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



$n = 0$ $n = 1$ $n = 2$

13. *Page numbering* Find the total number of decimal digits needed for numbering pages in a book of 1000 pages. Assume that the pages are numbered consecutively starting with 1.

2.4 Mathematical Analysis of Recursive Algorithms

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

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

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

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

ALGORITHM $F(n)$

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

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

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

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

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

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute}\\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply}\\ F(n-1) \text{ by } n}} \qquad \text{for } n > 0.$$

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

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

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

if $n = 0$ **return** 1.

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

$$M(0) = 0.$$

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

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

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

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

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

The second is the number of multiplications $M(n)$ needed to compute $F(n)$ by the recursive algorithm whose pseudocode was given at the beginning of the section.

As we just showed, $M(n)$ is defined by recurrence (2.2). And it is recurrence (2.2) that we need to solve now.

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

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

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

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

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

You should not be disappointed after exerting so much effort to get this "obvious" answer. The benefits of the method illustrated in this simple example will become clear very soon, when we have to solve more difficult recurrences. Also, note that the simple iterative algorithm that accumulates the product of n consecutive integers requires the same number of multiplications, and it does so without the overhead of time and space used for maintaining the recursion's stack.

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

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

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.

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

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

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

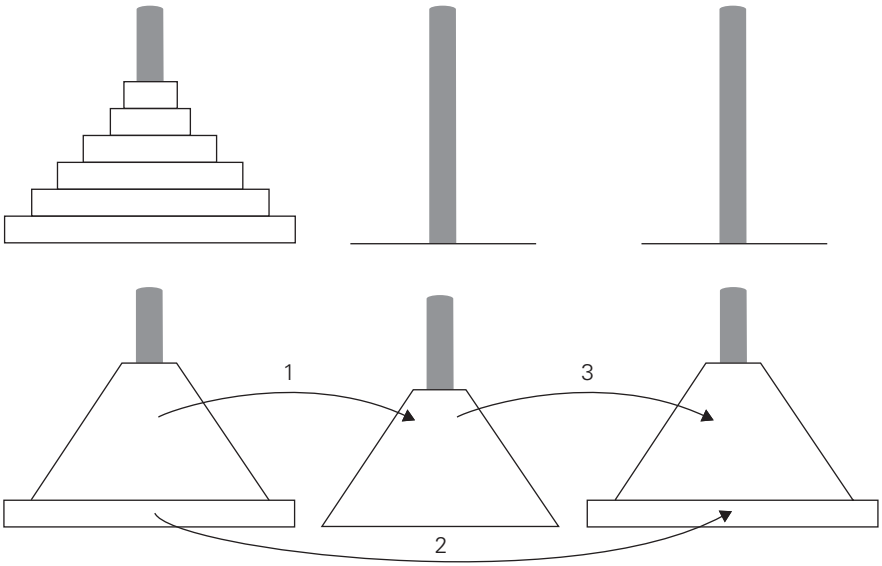


FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle.

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

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

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

$$\begin{aligned} M(n) &= 2M(n-1) + 1 \quad \text{for } n > 1, \\ M(1) &= 1. \end{aligned} \tag{2.3}$$

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

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

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

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

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

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

Thus, we have an exponential algorithm, which will run for an unimaginably long time even for moderate values of n (see Problem 5 in this section's exercises). This is not due to the fact that this particular algorithm is poor; in fact, it is not difficult to prove that this is the most efficient algorithm possible for this problem. It is the problem's intrinsic difficulty that makes it so computationally hard. Still, this example makes an important general point:

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

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

$$C(n) = \sum_{l=0}^{n-1} 2^l \text{ (where } l \text{ is the level in the tree in Figure 2.5)} = 2^n - 1.$$

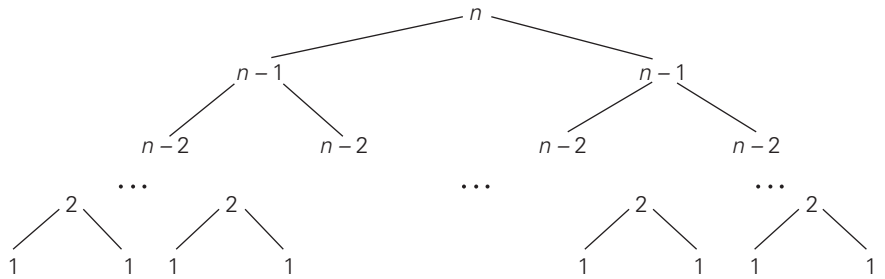


FIGURE 2.5 Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.

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

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

ALGORITHM *BinRec*(n)

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

Let us set up a recurrence and an initial condition for the number of additions $A(n)$ made by the algorithm. The number of additions made in computing $BinRec(\lfloor n/2 \rfloor)$ is $A(\lfloor n/2 \rfloor)$, plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence

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

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

$$A(1) = 0.$$

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

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

Now backward substitutions encounter no problems:

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

Thus, we end up with

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

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

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

In fact, one can prove (Problem 7 in this section's exercises) that the exact solution for an arbitrary value of n is given by just a slightly more refined formula $A(n) = \lfloor \log_2 n \rfloor$. ■

This section provides an introduction to the analysis of recursive algorithms. These techniques will be used throughout the book and expanded further as necessary. In the next section, we discuss the Fibonacci numbers; their analysis involves more difficult recurrence relations to be solved by a method different from backward substitutions.

Exercises 2.4

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

ALGORITHM $S(n)$

//Input: A positive integer n
//Output: The sum of the first n cubes
if $n = 1$ **return** 1
else return $S(n-1) + n * n * n$

a. Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.

b. How does this algorithm compare with the straightforward nonrecursive algorithm for computing this sum?

4. Consider the following recursive algorithm.

ALGORITHM $Q(n)$

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

a. Set up a recurrence relation for this function's values and solve it to determine what this algorithm computes.

b. Set up a recurrence relation for the number of multiplications made by this algorithm and solve it.

c. Set up a recurrence relation for the number of additions/subtractions made by this algorithm and solve it.

5. *Tower of Hanoi*

a. In the original version of the Tower of Hanoi puzzle, as it was published in the 1890s by Édouard Lucas, a French mathematician, the world will end after 64 disks have been moved from a mystical Tower of Brahma. Estimate the number of years it will take if monks could move one disk per minute. (Assume that monks do not eat, sleep, or die.)

b. How many moves are made by the ith largest disk $(1 \le i \le n)$ in this algorithm?

c. Find a nonrecursive algorithm for the Tower of Hanoi puzzle and implement it in the language of your choice.

6. *Restricted Tower of Hanoi* Consider the version of the Tower of Hanoi puzzle in which n disks have to be moved from peg A to peg C using peg B so that any move should either place a disk on peg B or move a disk from that peg. (Of course, the prohibition of placing a larger disk on top of a smaller one remains in place, too.) Design a recursive algorithm for this problem and find the number of moves made by it.

7. a. Prove that the exact number of additions made by the recursive algorithm $BinRec(n)$ for an arbitrary positive decimal integer n is $\lfloor \log_2 n \rfloor$.

b. Set up a recurrence relation for the number of additions made by the nonrecursive version of this algorithm (see Section 2.3, Example 4) and solve it.

8. a. Design a recursive algorithm for computing 2^n for any nonnegative integer n that is based on the formula $2^n = 2^{n-1} + 2^{n-1}$.

b. Set up a recurrence relation for the number of additions made by the algorithm and solve it.

c. Draw a tree of recursive calls for this algorithm and count the number of calls made by the algorithm.

d. Is it a good algorithm for solving this problem?

9. Consider the following recursive algorithm.

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

```
//Input: An array A[0..n − 1] of real numbers
if n = 1 return A[0]
else temp ← Riddle(A[0..n − 2])
    if temp ≤ A[n − 1] return temp
    else return A[n − 1]
```

a. What does this algorithm compute?

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

10. Consider the following algorithm to check whether a graph defined by its adjacency matrix is complete.

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

```
//Input: Adjacency matrix A[0..n − 1, 0..n − 1]) of an undirected graph G
//Output: 1 (true) if G is complete and 0 (false) otherwise
if n = 1 return 1   //one-vertex graph is complete by definition
else
    if not GraphComplete(A[0..n − 2, 0..n − 2]) return 0
    else for j ← 0 to n − 2 do
            if A[n − 1, j] = 0 return 0
        return 1
```

What is the algorithm's efficiency class in the worst case?

11. The determinant of an $n \times n$ matrix

$$A = \begin{bmatrix} a_{0\,0} & \cdots & a_{0\,n-1} \\ a_{1\,0} & \cdots & a_{1\,n-1} \\ \vdots & & \vdots \\ a_{n-1\,0} & \cdots & a_{n-1\,n-1} \end{bmatrix},$$

denoted det A, can be defined as a_{00} for $n = 1$ and, for $n > 1$, by the recursive formula

$$\det A = \sum_{j=0}^{n-1} s_j a_{0\,j} \det A_j,$$

where s_j is $+1$ if j is even and -1 if j is odd, $a_{0\,j}$ is the element in row 0 and column j, and A_j is the $(n-1) \times (n-1)$ matrix obtained from matrix A by deleting its row 0 and column j.

a. Set up a recurrence relation for the number of multiplications made by the algorithm implementing this recursive definition.

b. Without solving the recurrence, what can you say about the solution's order of growth as compared to $n!$?

12. *von Neumann's neighborhood revisited* Find the number of cells in the von Neumann neighborhood of range n (Problem 12 in Exercises 2.3) by setting up and solving a recurrence relation.

13. *Frying hamburgers* There are n hamburgers to be fried on a small grill that can hold only two hamburgers at a time. Each hamburger has to be fried on both sides; frying one side of a hamburger takes 1 minute, regardless of whether one or two hamburgers are fried at the same time. Consider the following recursive algorithm for executing this task in the minimum amount of time. If $n \leq 2$, fry the hamburger or the two hamburgers together on each side. If $n > 2$, fry any two hamburgers together on each side and then apply the same procedure recursively to the remaining $n - 2$ hamburgers.

a. Set up and solve the recurrence for the amount of time this algorithm needs to fry n hamburgers.

b. Explain why this algorithm does *not* fry the hamburgers in the minimum amount of time for all $n > 0$.

c. Give a correct recursive algorithm that executes the task in the minimum amount of time.

14. *Celebrity problem* A celebrity among a group of n people is a person who knows nobody but is known by everybody else. The task is to identify a celebrity by only asking questions to people of the form "Do you know him/her?" Design an efficient algorithm to identify a celebrity or determine that the group has no such person. How many questions does your algorithm need in the worst case?

2.5 Example: Computing the *n*th Fibonacci Number

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

$$0, \quad 1, \quad 1, \quad 2, \quad 3, \quad 5, \quad 8, \quad 13, \quad 21, \quad 34, \ldots \tag{2.5}$$

that can be defined by the simple recurrence

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

and two initial conditions

$$F(0) = 0, \qquad F(1) = 1. \tag{2.7}$$

The Fibonacci numbers were introduced by Leonardo Fibonacci in 1202 as a solution to a problem about the size of a rabbit population (Problem 2 in this section's exercises). Many more examples of Fibonacci-like numbers have since been discovered in the natural world, and they have even been used in predicting the prices of stocks and commodities. There are some interesting applications of the Fibonacci numbers in computer science as well. For example, worst-case inputs for Euclid's algorithm discussed in Section 1.1 happen to be consecutive elements of the Fibonacci sequence. In this section, we briefly consider algorithms for computing the *n*th element of this sequence. Among other benefits, the discussion will provide us with an opportunity to introduce another method for solving recurrence relations useful for analysis of recursive algorithms.

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

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

where a, b, and c are some fixed real numbers ($a \neq 0$) called the coefficients of the recurrence and $x(n)$ is the generic term of an unknown sequence to be found. Applying this theorem to our recurrence with the initial conditions given—see Appendix B—we obtain the formula

$$F(n) = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n), \tag{2.9}$$

where $\phi = (1 + \sqrt{5})/2 \approx 1.61803$ and $\hat{\phi} = -1/\phi \approx -0.61803$.[6] It is hard to believe that formula (2.9), which includes arbitrary integer powers of irrational numbers, yields nothing else but all the elements of Fibonacci sequence (2.5), but it does!

One of the benefits of formula (2.9) is that it immediately implies that $F(n)$ grows exponentially (remember Fibonacci's rabbits?), i.e., $F(n) \in \Theta(\phi^n)$. This

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

follows from the observation that $\hat{\phi}$ is a fraction between -1 and 0, and hence $\hat{\phi}^n$ gets infinitely small as n goes to infinity. In fact, one can prove that the impact of the second term $\frac{1}{\sqrt{5}}\hat{\phi}^n$ on the value of $F(n)$ can be obtained by rounding off the value of the first term to the nearest integer. In other words, for every nonnegative integer n,

$$F(n) = \frac{1}{\sqrt{5}}\phi^n \quad \text{rounded to the nearest integer.} \tag{2.10}$$

In the algorithms that follow, we consider, for the sake of simplicity, such operations as additions and multiplications at unit cost. Since the Fibonacci numbers grow infinitely large (and grow very rapidly), a more detailed analysis than the one offered here is warranted. In fact, it is the size of the numbers rather than a time-efficient method for computing them that should be of primary concern here. Still, these caveats notwithstanding, the algorithms we outline and their analysis provide useful examples for a student of the design and analysis of algorithms.

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

ALGORITHM $F(n)$

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

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

$$\begin{aligned} &A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1, \\ &A(0) = 0, \qquad A(1) = 0. \end{aligned} \tag{2.11}$$

The recurrence $A(n) - A(n-1) - A(n-2) = 1$ is quite similar to recurrence $F(n) - F(n-1) - F(n-2) = 0$, but its right-hand side is not equal to zero. Such recurrences are called ***inhomogeneous***. There are general techniques for solving inhomogeneous recurrences (see Appendix B or any textbook on discrete mathematics), but for this particular recurrence, a special trick leads to a faster solution. We can reduce our inhomogeneous recurrence to a homogeneous one by rewriting it as

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

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

$$B(n) - B(n-1) - B(n-2) = 0,$$
$$B(0) = 1, \qquad B(1) = 1.$$

This homogeneous recurrence can be solved exactly in the same manner as recurrence (2.6) was solved to find an explicit formula for $F(n)$. But it can actually be avoided by noting that $B(n)$ is, in fact, the same recurrence as $F(n)$ except that it starts with two 1's and thus runs one step ahead of $F(n)$. So $B(n) = F(n+1)$, and

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

Hence, $A(n) \in \Theta(\phi^n)$, and if we measure the size of n by the number of bits $b = \lfloor \log_2 n \rfloor + 1$ in its binary representation, the efficiency class will be even worse, namely, doubly exponential: $A(b) \in \Theta(\phi^{2^b})$.

The poor efficiency class of the algorithm could be anticipated by the nature of recurrence (2.11). Indeed, it contains two recursive calls with the sizes of smaller instances only slightly smaller than size n. (Have you encountered such a situation before?) We can also see the reason behind the algorithm's inefficiency by looking at a recursive tree of calls tracing the algorithm's execution. An example of such a tree for $n = 5$ is given in Figure 2.6. Note that the same values of the function are being evaluated here again and again, which is clearly extremely inefficient.

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

ALGORITHM *Fib*(n)

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

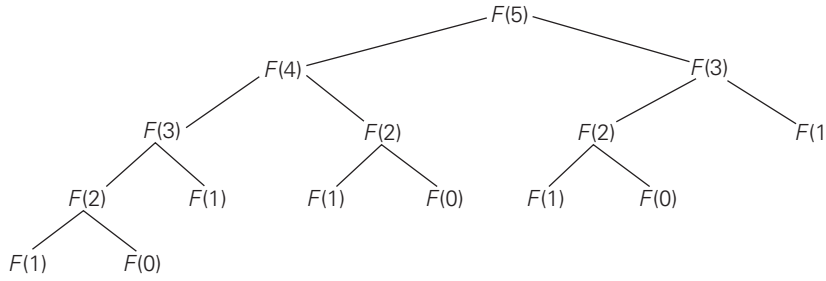


FIGURE 2.6 Tree of recursive calls for computing the 5th Fibonacci number by the definition-based algorithm.

This algorithm clearly makes $n - 1$ additions. Hence, it is linear as a function of n and "only" exponential as a function of the number of bits b in n's binary representation. Note that using an extra array for storing all the preceding elements of the Fibonacci sequence can be avoided: storing just two values is necessary to accomplish the task (see Problem 8 in this section's exercises).

The third alternative for computing the nth Fibonacci number lies in using formula (2.10). The efficiency of the algorithm will obviously be determined by the efficiency of an exponentiation algorithm used for computing ϕ^n. If it is done by simply multiplying ϕ by itself $n - 1$ times, the algorithm will be in $\Theta(n) = \Theta(2^b)$. There are faster algorithms for the exponentiation problem. For example, we will discuss $\Theta(\log n) = \Theta(b)$ algorithms for this problem in Chapters 4 and 6. Note also that special care should be exercised in implementing this approach to computing the nth Fibonacci number. Since all its intermediate results are irrational numbers, we would have to make sure that their approximations in the computer are accurate enough so that the final round-off yields a correct result.

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

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

and an efficient way of computing matrix powers.

Exercises 2.5

1. Find a Web site dedicated to applications of the Fibonacci numbers and study it.

2. *Fibonacci's rabbits problem* A man put a pair of rabbits in a place surrounded by a wall. How many pairs of rabbits will be there in a year if the initial pair of rabbits (male and female) are newborn and all rabbit pairs are not fertile during their first month of life but thereafter give birth to one new male/female pair at the end of every month?

3. *Climbing stairs* Find the number of different ways to climb an n-stair staircase if each step is either one or two stairs. For example, a 3-stair staircase can be climbed three ways: 1-1-1, 1-2, and 2-1.

4. How many even numbers are there among the first n Fibonacci numbers, i.e., among the numbers $F(0), F(1), \ldots, F(n-1)$? Give a closed-form formula valid for every $n > 0$.

5. Check by direct substitutions that the function $\frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$ indeed satisfies recurrence (2.6) and initial conditions (2.7).

6. The maximum values of the Java primitive types `int` and `long` are $2^{31} - 1$ and $2^{63} - 1$, respectively. Find the smallest n for which the nth Fibonacci number is not going to fit in a memory allocated for

a. the type `int`. **b.** the type `long`.

7. Consider the recursive definition-based algorithm for computing the nth Fibonacci number $F(n)$. Let $C(n)$ and $Z(n)$ be the number of times $F(1)$ and $F(0)$ are computed, respectively. Prove that

a. $C(n) = F(n)$. **b.** $Z(n) = F(n-1)$.

8. Improve algorithm Fib of the text so that it requires only $\Theta(1)$ space.

9. Prove the equality

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

10. How many modulo divisions are made by Euclid's algorithm on two consecutive Fibonacci numbers $F(n)$ and $F(n-1)$ as the algorithm's input?

11. *Dissecting a Fibonacci rectangle* Given a rectangle whose sides are two consecutive Fibonacci numbers, design an algorithm to dissect it into squares with no more than two squares being the same size. What is the time efficiency class of your algorithm?

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

2.6 Empirical Analysis of Algorithms

In Sections 2.3 and 2.4, we saw how algorithms, both nonrecursive and recursive, can be analyzed mathematically. Though these techniques can be applied successfully to many simple algorithms, the power of mathematics, even when enhanced with more advanced techniques (see [Sed96], [Pur04], [Gra94], and [Gre07]), is far from limitless. In fact, even some seemingly simple algorithms have proved to be very difficult to analyze with mathematical precision and certainty. As we pointed out in Section 2.1, this is especially true for the average-case analysis.

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

General Plan for the Empirical Analysis of Algorithm Time Efficiency

1. Understand the experiment's purpose.

2. Decide on the efficiency metric M to be measured and the measurement unit (an operation count vs. a time unit).

3. Decide on characteristics of the input sample (its range, size, and so on).

4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.

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

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

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

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

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

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

your program. (In UNIX, this time is called the "user time," and it is automatically provided by the `time` command.)

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

Whether you decide to measure the efficiency by basic operation counting or by time clocking, you will need to decide on a sample of inputs for the experiment. Often, the goal is to use a sample representing a "typical" input; so the challenge is to understand what a "typical" input is. For some classes of algorithms—e.g., for algorithms for the traveling salesman problem that we are going to discuss later in the book—researchers have developed a set of instances they use for benchmarking. But much more often than not, an input sample has to be developed by the experimenter. Typically, you will have to make decisions about the sample size (it is sensible to start with a relatively small sample and increase it later if necessary), the range of instance sizes (typically neither trivially small nor excessively large), and a procedure for generating instances in the range chosen. The instance sizes can either adhere to some pattern (e.g., 1000, 2000, 3000, . . . , 10,000 or 500, 1000, 2000, 4000, . . . , 128,000) or be generated randomly within the range chosen.

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

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

Much more often than not, an empirical analysis requires generating random numbers. Even if you decide to use a pattern for input sizes, you will typically want instances themselves generated randomly. Generating random numbers on a digital computer is known to present a difficult problem because, in principle, the problem can be solved only approximately. This is the reason computer scientists prefer to call such numbers ***pseudorandom***. As a practical matter, the easiest and most natural way of getting such numbers is to take advantage of a random number generator available in computer language libraries. Typically, its output will be a value of a (pseudo)random variable uniformly distributed in the interval between 0 and 1. If a different (pseudo)random variable is desired, an appropriate transformation needs to be made. For example, if x is a continuous random variable uniformly distributed on the interval $0 \le x < 1$, the variable $y = l + \lfloor x(r - l) \rfloor$ will be uniformly distributed among the integer values between integers l and $r - 1$ ($l < r$).

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

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

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

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

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

The principal advantage of tabulated data lies in the opportunity to manipulate it easily. For example, one can compute the ratios $M(n)/g(n)$ where $g(n)$ is a candidate to represent the efficiency class of the algorithm in question. If the algorithm is indeed in $\Theta(g(n))$, most likely these ratios will converge to some positive constant as n gets large. (Note that careless novices sometimes assume that this constant must be 1, which is, of course, incorrect according to the definition of $\Theta(g(n))$.) Or one can compute the ratios $M(2n)/M(n)$ and see how the running time reacts to doubling of its input size. As we discussed in Section 2.2, such ratios should change only slightly for logarithmic algorithms and most likely converge to 2, 4, and 8 for linear, quadratic, and cubic algorithms, respectively—to name the most obvious and convenient cases.

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

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

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

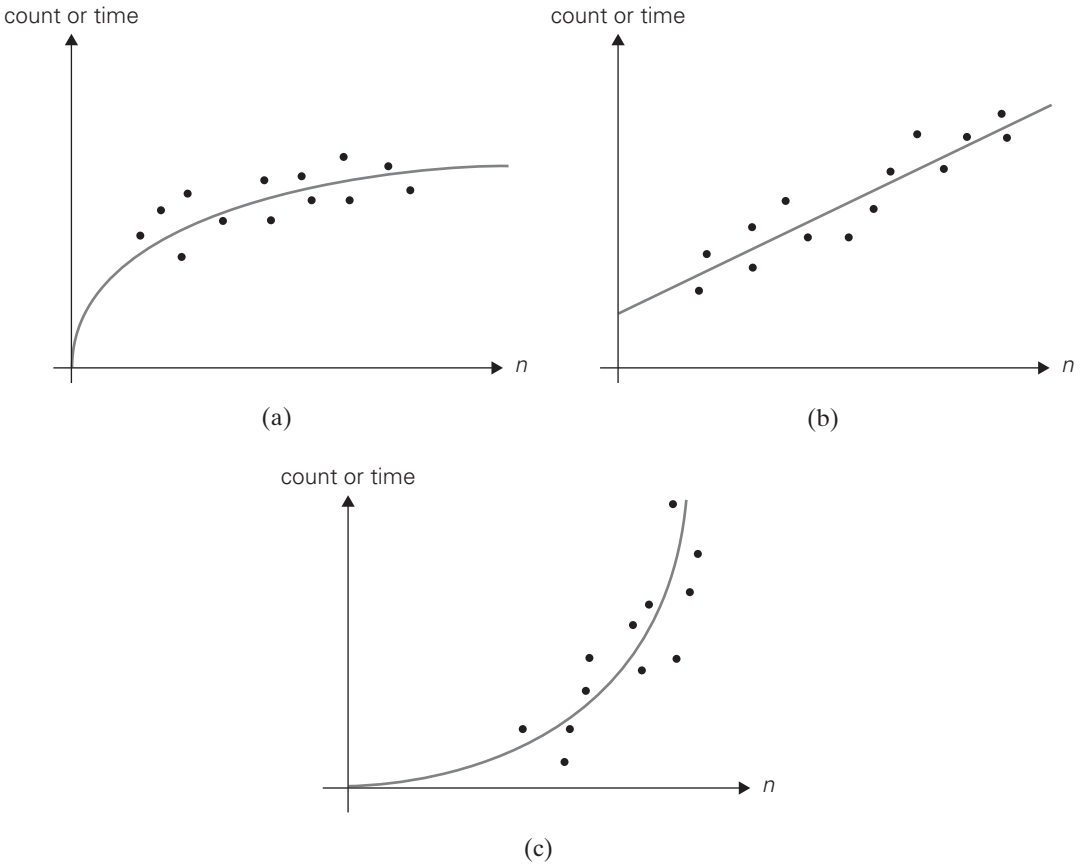


FIGURE 2.7 Typical scatter plots. (a) Logarithmic. (b) Linear. (c) One of the convex functions.

Exercises 2.6

1. Consider the following well-known sorting algorithm, which is studied later in the book, with a counter inserted to count the number of key comparisons.

ALGORITHM *SortAnalysis*$(A[0..n-1])$
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: The total number of key comparisons made
$count \leftarrow 0$
for $i \leftarrow 1$ **to** $n-1$ **do**

```
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        count ← count + 1
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
return count
```

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

2. **a.** Run the program of Problem 1, with a properly inserted counter (or counters) for the number of key comparisons, on 20 random arrays of sizes 1000, 2000, 3000, . . . , 20,000.

b. Analyze the data obtained to form a hypothesis about the algorithm's average-case efficiency.

c. Estimate the number of key comparisons we should expect for a randomly generated array of size 25,000 sorted by the same algorithm.

3. Repeat Problem 2 by measuring the program's running time in milliseconds.

4. Hypothesize a likely efficiency class of an algorithm based on the following empirical observations of its basic operation's count:

size	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
count	11,966	24,303	39,992	53,010	67,272	78,692	91,274	113,063	129,799	140,538

5. What scale transformation will make a logarithmic scatterplot look like a linear one?

6. How can one distinguish a scatterplot for an algorithm in $\Theta(\lg \lg n)$ from a scatterplot for an algorithm in $\Theta(\lg n)$?

7. **a.** Find empirically the largest number of divisions made by Euclid's algorithm for computing $\gcd(m, n)$ for $1 \leq n \leq m \leq 100$.

b. For each positive integer k, find empirically the smallest pair of integers $1 \leq n \leq m \leq 100$ for which Euclid's algorithm needs to make k divisions in order to find $\gcd(m, n)$.

8. The average-case efficiency of Euclid's algorithm on inputs of size n can be measured by the average number of divisions $D_{avg}(n)$ made by the algorithm in computing $\gcd(n, 1)$, $\gcd(n, 2)$, . . . , $\gcd(n, n)$. For example,

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

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

9. Run an experiment to ascertain the efficiency class of the sieve of Eratosthenes (see Section 1.1).

10. Run a timing experiment for the three algorithms for computing $\gcd(m, n)$ presented in Section 1.1.

2.7 Algorithm Visualization

In addition to the mathematical and empirical analyses of algorithms, there is yet a third way to study algorithms. It is called ***algorithm visualization*** and can be defined as the use of images to convey some useful information about algorithms. That information can be a visual illustration of an algorithm's operation, of its performance on different kinds of inputs, or of its execution speed versus that of other algorithms for the same problem. To accomplish this goal, an algorithm visualization uses graphic elements—points, line segments, two- or three-dimensional bars, and so on—to represent some "interesting events" in the algorithm's operation.

There are two principal variations of algorithm visualization:

- Static algorithm visualization
- Dynamic algorithm visualization, also called ***algorithm animation***

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

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

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

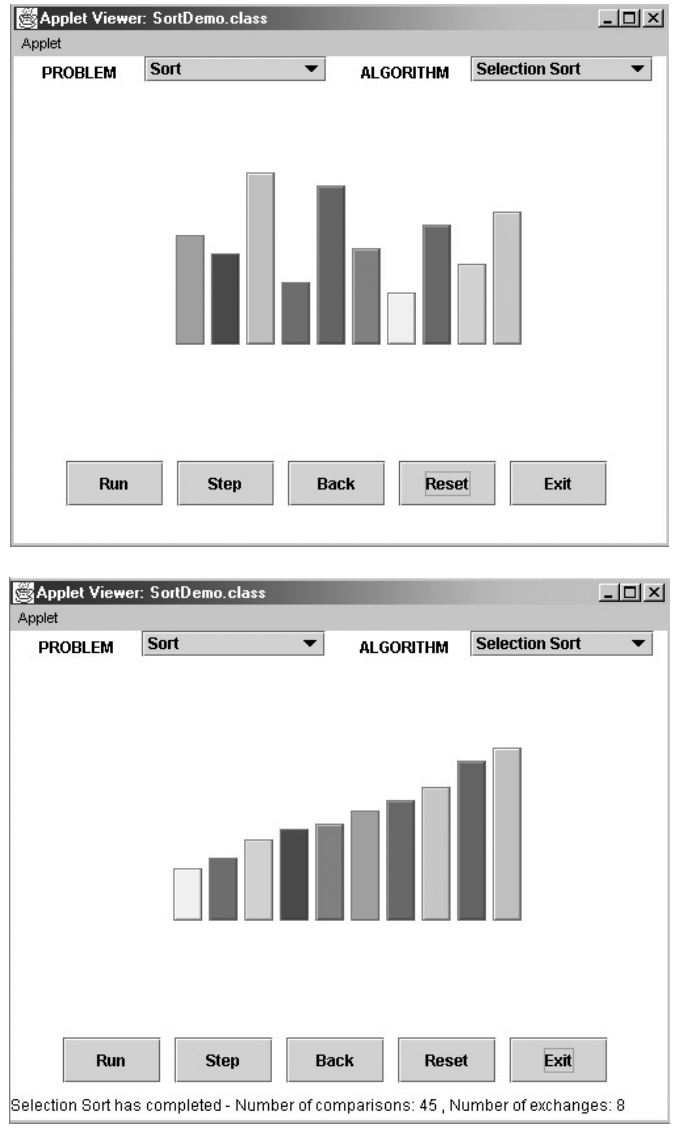


FIGURE 2.8 Initial and final screens of a typical visualization of a sorting algorithm using the bar representation.

work by comparing and exchanging two given items at a time—an event that can be animated relatively easily.

Since the appearance of *Sorting Out Sorting*, a great number of algorithm animations have been created, especially after the appearance of Java and the

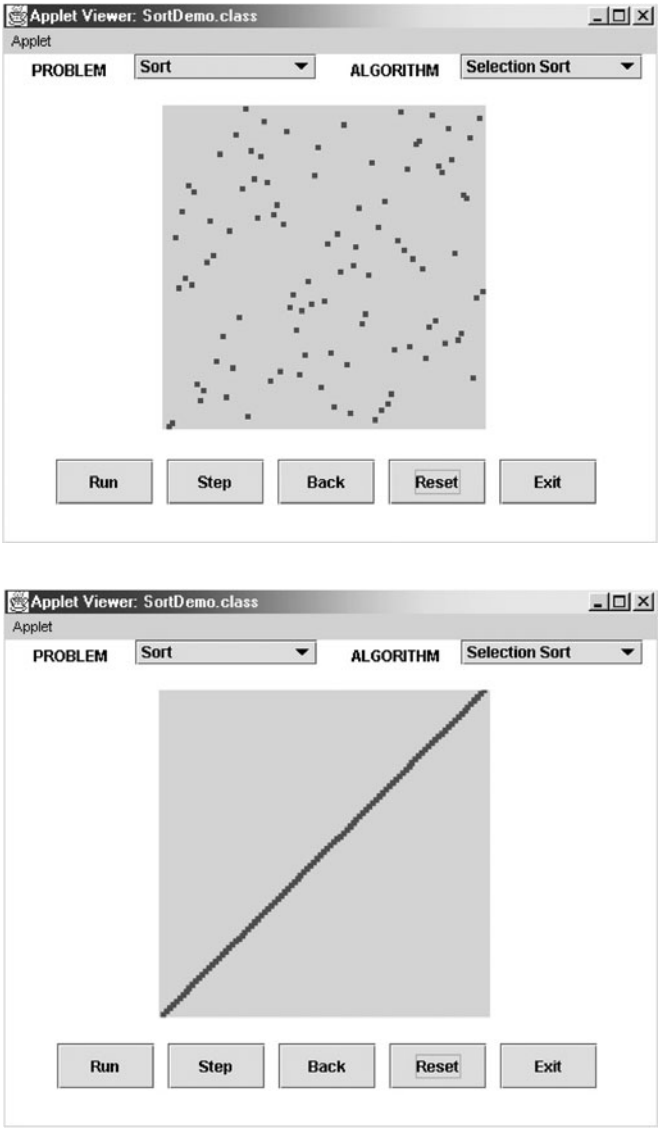


FIGURE 2.9 Initial and final screens of a typical visualization of a sorting algorithm using the scatterplot representation.

World Wide Web in the 1990s. They range in scope from one particular algorithm to a group of algorithms for the same problem (e.g., sorting) or the same application area (e.g., geometric algorithms) to general-purpose animation systems. At the end of 2010, a catalog of links to existing visualizations, maintained under the

NSF-supported AlgoVizProject, contained over 500 links. Unfortunately, a survey of existing visualizations found most of them to be of low quality, with the content heavily skewed toward easier topics such as sorting [Sha07].

There are two principal applications of algorithm visualization: research and education. Potential benefits for researchers are based on expectations that algorithm visualization may help uncover some unknown features of algorithms. For example, one researcher used a visualization of the recursive Tower of Hanoi algorithm in which odd- and even-numbered disks were colored in two different colors. He noticed that two disks of the same color never came in direct contact during the algorithm's execution. This observation helped him in developing a better nonrecursive version of the classic algorithm. To give another example, Bentley and McIlroy [Ben93] mentioned using an algorithm animation system in their work on improving a library implementation of a leading sorting algorithm.

The application of algorithm visualization to education seeks to help students learning algorithms. The available evidence of its effectiveness is decisively mixed. Although some experiments did register positive learning outcomes, others failed to do so. The increasing body of evidence indicates that creating sophisticated software systems is not going to be enough. In fact, it appears that the level of student involvement with visualization might be more important than specific features of visualization software. In some experiments, low-tech visualizations prepared by students were more effective than passive exposure to sophisticated software systems.

To summarize, although some successes in both research and education have been reported in the literature, they are not as impressive as one might expect. A deeper understanding of human perception of images will be required before the true potential of algorithm visualization is fulfilled.

SUMMARY

- There are two kinds of algorithm efficiency: time efficiency and space efficiency. *Time efficiency* indicates how fast the algorithm runs; *space efficiency* deals with the extra space it requires.
- An algorithm's time efficiency is principally measured as a function of its input size by counting the number of times its basic operation is executed. A *basic operation* is the operation that contributes the most to running time. Typically, it is the most time-consuming operation in the algorithm's innermost loop.
- For some algorithms, the running time can differ considerably for inputs of the same size, leading to *worst-case* efficiency, *average-case* efficiency, and *best-case* efficiency.
- The established framework for analyzing time efficiency is primarily grounded in the order of growth of the algorithm's running time as its input size goes to infinity.

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

This page intentionally left blank