1.1 What Is an Algorithm?

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

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

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

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

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

- The nonambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.

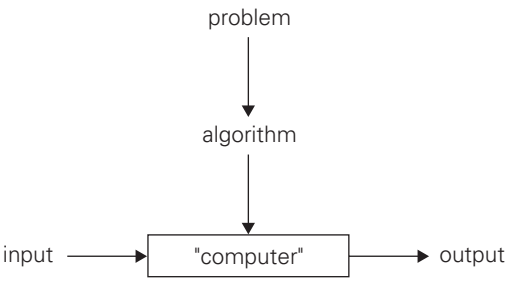


FIGURE 1.1 The notion of the algorithm.

- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

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

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

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

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

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

(If you are not impressed by this algorithm, try finding the greatest common divisor of larger numbers, such as those in Problem 6 in this section's exercises.)

Here is a more structured description of this algorithm:

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

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide m by n and assign the value of the remainder to r.

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

Alternatively, we can express the same algorithm in pseudocode:

ALGORITHM *Euclid*(m, n)

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

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

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

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

Step 1 Assign the value of $\min\{m, n\}$ to t.

Step 2 Divide m by t. If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.

Step 3 Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.

Step 4 Decrease the value of t by 1. Go to Step 2.

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

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

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

Step 1 Find the prime factors of m.

Step 2 Find the prime factors of n.

Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If p is a common factor occurring p_m and p_n times in m and n, respectively, it should be repeated $\min\{p_m, p_n\}$ times.)

Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Thus, for the numbers 60 and 24, we get

$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$
$$24 = 2 \cdot 2 \cdot 2 \cdot 3$$
$$\gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12.$$

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

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

So, let us introduce a simple algorithm for generating consecutive primes not exceeding any given integer $n > 1$. It was probably invented in ancient Greece and is known as the ***sieve of Eratosthenes*** (ca. 200 B.C.). The algorithm starts by initializing a list of prime candidates with consecutive integers from 2 to n. Then, on its first iteration, the algorithm eliminates from the list all multiples of 2, i.e., 4, 6, and so on. Then it moves to the next item on the list, which is 3, and eliminates its multiples. (In this straightforward version, there is an overhead because some numbers, such as 6, are eliminated more than once.) No pass for number 4 is needed: since 4 itself and all its multiples are also multiples of 2, they were already eliminated on a previous pass. The next remaining number on the list, which is used on the third pass, is 5. The algorithm continues in this fashion until no more numbers can be eliminated from the list. The remaining integers of the list are the primes needed.

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

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

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

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

ALGORITHM *Sieve*(n)

//Implements the sieve of Eratosthenes
//Input: A positive integer $n > 1$
//Output: Array L of all prime numbers less than or equal to n

```
for p ← 2 to n do A[p] ← p
for p ← 2 to ⌊√n⌋ do    //see note before pseudocode
    if A[p] ≠ 0          //p hasn't been eliminated on previous passes
        j ← p * p
        while j ≤ n do
            A[j] ← 0     //mark element as eliminated
            j ← j + p
//copy the remaining elements of A to array L of the primes
i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i + 1
return L
```

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

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

1.2 Fundamentals of Algorithmic Problem Solving

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

> We can consider algorithms to be procedural solutions to problems.

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

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

Understanding the Problem

From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

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

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

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

Ascertaining the Capabilities of the Computational Device

Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. The vast majority of

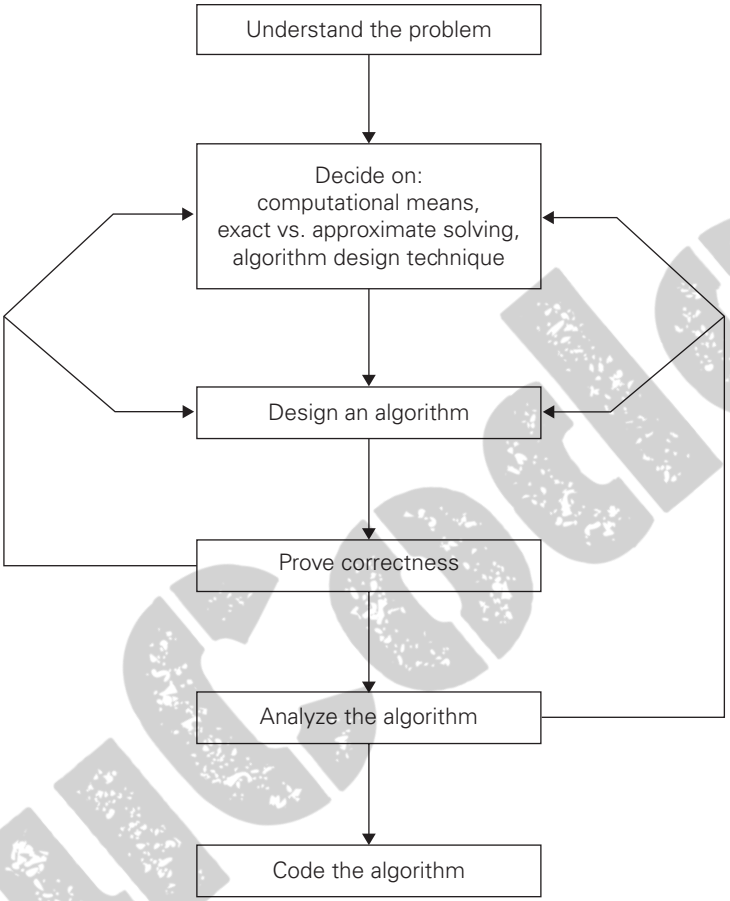


FIGURE 1.2 Algorithm design and analysis process.

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

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

Should you worry about the speed and amount of memory of a computer at your disposal? If you are designing an algorithm as a scientific exercise, the answer is a qualified no. As you will see in Section 2.1, most computer scientists prefer to study algorithms in terms independent of specification parameters for a particular computer. If you are designing an algorithm as a practical tool, the answer may depend on a problem you need to solve. Even the "slow" computers of today are almost unimaginably fast. Consequently, in many situations you need not worry about a computer being too slow for the task. There are important problems, however, that are very complex by their nature, or have to process huge volumes of data, or deal with applications where the time is critical. In such situations, it is imperative to be aware of the speed and memory available on a particular computer system.

Choosing between Exact and Approximate Problem Solving

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

Algorithm Design Techniques

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

What is an algorithm design technique?

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

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

First, they provide guidance for designing algorithms for new problems, i.e., problems for which there is no known satisfactory algorithm. Therefore—to use the language of a famous proverb—learning such techniques is akin to learning

to fish as opposed to being given a fish caught by somebody else. It is not true, of course, that each of these general techniques will be necessarily applicable to every problem you may encounter. But taken together, they do constitute a powerful collection of tools that you will find quite handy in your studies and work.

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

Designing an Algorithm and Data Structures

While the algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem may still be a challenging task. Some design techniques can be simply inapplicable to the problem in question. Sometimes, several techniques need to be combined, and there are algorithms that are hard to pinpoint as applications of the known design techniques. Even when a particular design technique is applicable, getting an algorithm often requires a nontrivial ingenuity on the part of the algorithm designer. With practice, both tasks—choosing among the general techniques and applying them—get easier, but they are rarely easy.

Of course, one should pay close attention to choosing data structures appropriate for the operations performed by the algorithm. For example, the sieve of Eratosthenes introduced in Section 1.1 would run longer if we used a linked list instead of an array in its implementation (why?). Also note that some of the algorithm design techniques discussed in Chapters 6 and 7 depend intimately on structuring or restructuring data specifying a problem's instance. Many years ago, an influential textbook proclaimed the fundamental importance of both algorithms and data structures for computer programming by its very title: *Algorithms + Data Structures = Programs* [Wir76]. In the new world of object-oriented programming, data structures remain crucially important for both design and analysis of algorithms. We review basic data structures in Section 1.4.

Methods of Specifying an Algorithm

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

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

Pseudocode is a mixture of a natural language and programming language-like constructs. Pseudocode is usually more precise than natural language, and its

usage often yields more succinct algorithm descriptions. Surprisingly, computer scientists have never agreed on a single form of pseudocode, leaving textbook authors with a need to design their own "dialects." Fortunately, these dialects are so close to each other that anyone familiar with a modern programming language should be able to understand them all.

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

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

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

Proving an Algorithm's Correctness

Once an algorithm has been specified, you have to prove its ***correctness***. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time. For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality $\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$ (which, in turn, needs a proof; see Problem 7 in Exercises 1.1), the simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0.

For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex. A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs. It might be worth mentioning that although tracing the algorithm's performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm's correctness conclusively. But in order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.

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

Analyzing an Algorithm

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

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

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

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

If you are not satisfied with the algorithm's efficiency, simplicity, or generality, you must return to the drawing board and redesign the algorithm. In fact, even if your evaluation is positive, it is still worth searching for other algorithmic solutions. Recall the three different algorithms in the previous section for computing the greatest common divisor: generally, you should not expect to get the best algorithm on the first try. At the very least, you should try to fine-tune the algorithm you

already have. For example, we made several improvements in our implementation of the sieve of Eratosthenes compared with its initial outline in Section 1.1. (Can you identify them?) You will do well if you keep in mind the following observation of Antoine de Saint-Exupéry, the French writer, pilot, and aircraft designer: "A designer knows he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away."[1]

Coding an Algorithm

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

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

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

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

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

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

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

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

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

In the academic world, the question leads to an interesting but usually difficult investigation of an algorithm's ***optimality***. Actually, this question is not about the efficiency of an algorithm but about the complexity of the problem it solves: What is the minimum amount of effort *any* algorithm will need to exert to solve the problem? For some problems, the answer to this question is known. For example, any algorithm that sorts an array by comparing values of its elements needs about $n \log_2 n$ comparisons for some arrays of size n (see Section 11.2). But for many seemingly easy problems such as integer multiplication, computer scientists do not yet have a final answer.

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

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

Fundamentals of the Analysis of Algorithm Efficiency

2.1 The Analysis Framework

- ➤Measuring an Input's Size
- ➤Units for Measuring Running Time
- ➤Orders of Growth
- ➤Worst-Case, Best-Case, and Average-Case Efficiencies
- ➤Recapitulation of the Analysis Framework

2.2 Asymptotic Notations and Basic Efficiency Classes

- ➤Informal Introduction
- ➤O-notation
- ➤Big Omega notation
- ➤Big Theta notation
- ➤Useful Property Involving the Asymptotic Notations
- ➤Using Limits for Comparing Orders of Growth
- ➤Basic Efficiency Classes

2.3 Mathematical Analysis of Non recursive Algorithms

2.4 Mathematical Analysis of Recursive Algorithms

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

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

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.

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 $n_0 = 1$.

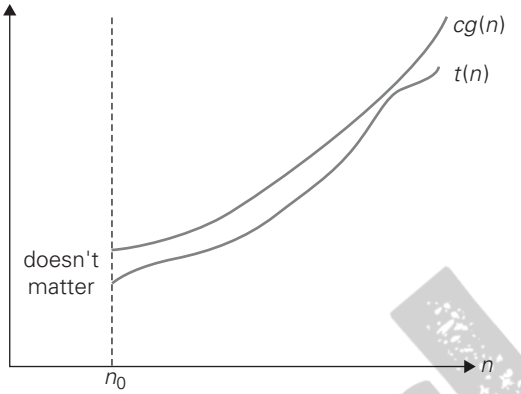


FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$.

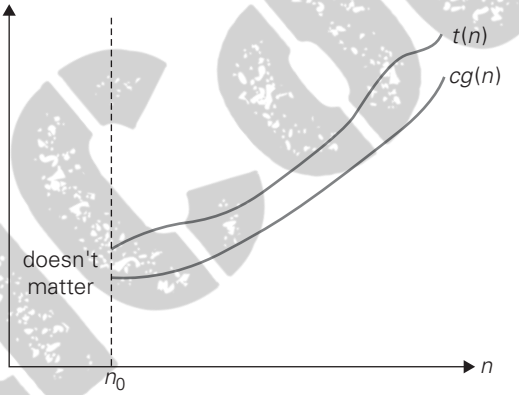


FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$.

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 $n_0 = 0$.

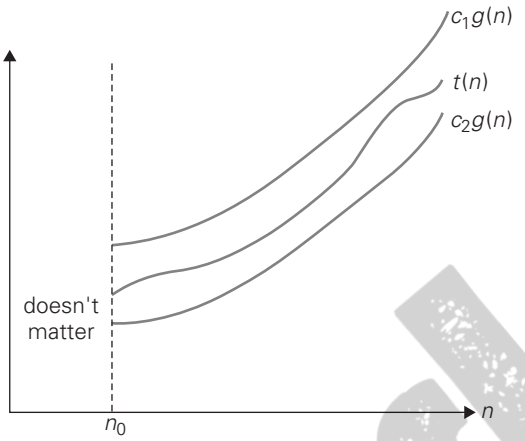


FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$.

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.) ■

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.

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,

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

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

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

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

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

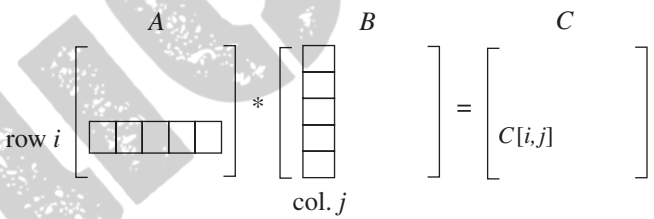
$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$
$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2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

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. ■

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

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

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.

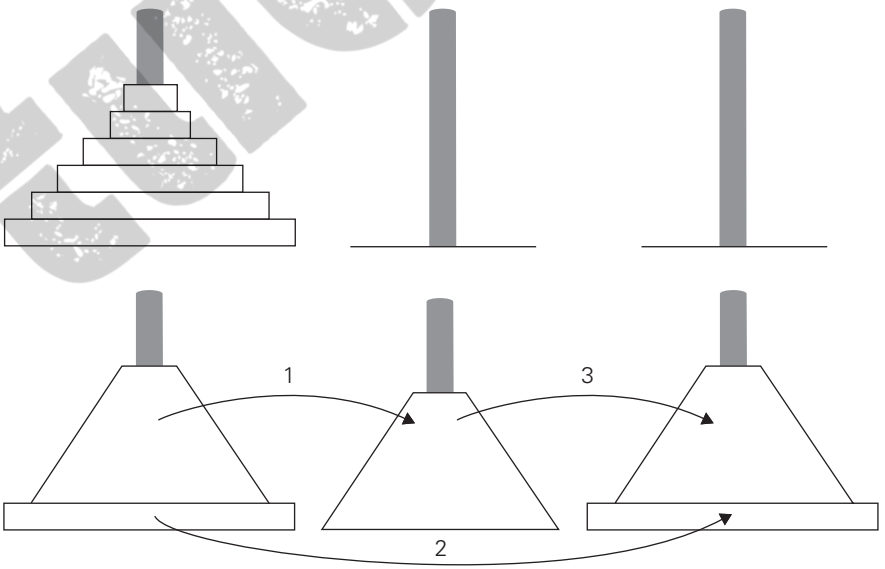


FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle.

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:

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

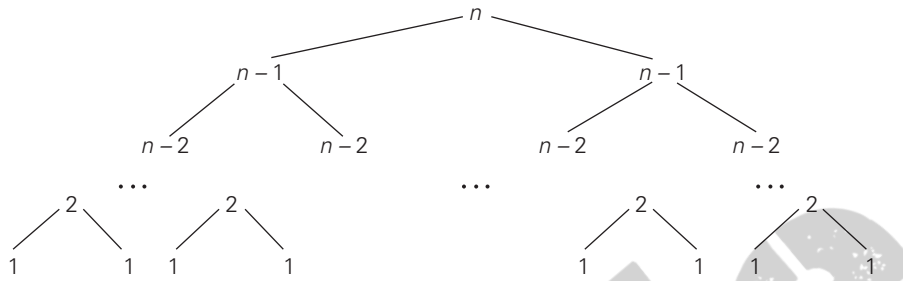


FIGURE 2.5 Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.

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:

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

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.

Brute Force Approaches

3.1 Selection Sort and Bubble Sort

- ➤**Selection Sort**
- ➤**Bubble Sort**

3.2 Sequential Search and Brute-Force String Matching

- ➤**Sequential Search**
- ➤**Brute-Force String Matching**

3.1 Selection Sort and Bubble Sort

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

Now, after your mind is unburdened of previous knowledge of sorting algorithms, ask yourself a question: "What would be the most straightforward method for solving the sorting problem?" Reasonable people may disagree on the answer to this question. The two algorithms discussed here—selection sort and bubble sort—seem to be the two prime candidates.

Selection Sort

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

ith pass through the list, which we number from 0 to $n-2$, the algorithm searches for the smallest item among the last $n-i$ elements and swaps it with A_i:

$$\underbrace{A_0 \le A_1 \le \cdots \le A_{i-1}}_{\text{in their final positions}} \;\big|\; \underbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}_{\text{the last } n-i \text{ elements}}$$

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

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

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

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

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

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

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

| 89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

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

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

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

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

Bubble Sort

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

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \le \cdots \le A_{n-1}}_{\text{in their final positions}}$$

Here is pseudocode of this algorithm.

ALGORITHM *BubbleSort*$(A[0..n-1])$
//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

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

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

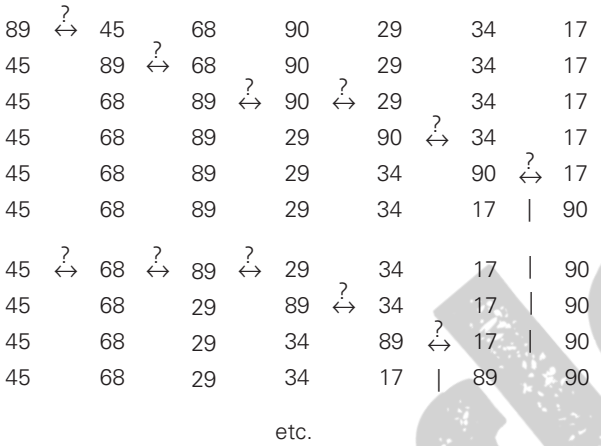


FIGURE 3.2 First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

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

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

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

As is often the case with an application of the brute-force strategy, the first version of an algorithm obtained can often be improved upon with a modest amount of effort. Specifically, we can improve the crude version of bubble sort given above by exploiting the following observation: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm (Problem 12a in this section's exercises). Though the new version runs faster on some inputs, it is still in $\Theta(n^2)$ in the worst and average cases. In fact, even among elementary sorting methods, bubble sort is an inferior choice, and if it were not for its catchy name, you would probably have never heard of it. However, the general lesson you just learned is important and worth repeating:

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

3.2 Sequential Search and Brute-Force String Matching

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

Sequential Search

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

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

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

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

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

Brute-Force String Matching

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

$$\begin{array}{ccccccccccc} t_0 & \ldots & t_i & \ldots & t_{i+j} & \ldots & t_{i+m-1} & \ldots & t_{n-1} & \text{text } T \\ & & \updownarrow & & \updownarrow & & \updownarrow & & & \\ & & p_0 & \ldots & p_j & \ldots & p_{m-1} & \text{pattern } P & & \end{array}$$

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

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

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

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

An operation of the algorithm is illustrated in Figure 3.3. Note that for this example, the algorithm shifts the pattern almost always after a single character comparison. The worst case is much worse: the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the $n - m + 1$ tries. (Problem 6 in this section's exercises asks you to give a specific example of such a situation.) Thus, in the worst case, the algorithm makes

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

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

$m(n - m + 1)$ character comparisons, which puts it in the $O(nm)$ class. For a typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons (check the example again). Therefore, the average-case efficiency should be considerably better than the worst-case efficiency. Indeed it is: for searching in random texts, it has been shown to be linear, i.e., $\Theta(n)$. There are several more sophisticated and more efficient algorithms for string searching. The most widely known of them—by R. Boyer and J. Moore—is outlined in Section 7.2 along with its simplification suggested by R. Horspool.