Course Name: M.Sc. in Computer Science
Semester – II, Session: 2019-2021
Department of Computer Science
Name of Faculty: Gautam Mahapatra, Associate Professor
Subject Taught: **Advanced Data Structure – Growth Functions with different theorems and operations, Recurrence Relations – Homogeneous & Non-Homogeneous and its solutions, Master Theorem and its applications, Divide-and-Conquer Technique for Binary Search, Max/Min, Merge Sort, Fast Integer Multiplication, Fast Matrix Multiplication**

Class Taken:
Date: 18th June 2020, Time: 11.00AM – 1.40PM

Number of Students Attended: 18 / 25

Software used: Skype
Internet Service: Jio-Fi

Details of the subject taught:

approximate the time required to solve a problem of size n by multiplying the previous time required by a constant. For example, on a supercomputer we might be able to solve a problem of size n a million times faster than we can on a PC. However, this factor of one million will not depend on n (except perhaps in some minor ways). One of the advantages of using **big-O notation**, which we introduce in this section, is that we can estimate the growth of a function without worrying about constant multipliers or smaller order terms. This means that, using big-O notation, we do not have to worry about the hardware and software used to implement an algorithm. Furthermore, using big-O notation, we can assume that the different operations used in an algorithm take the same time, which simplifies the analysis considerably.

Big-O notation is used extensively to estimate the number of operations an algorithm uses as its input grows. With the help of this notation, we can determine whether it is practical to use a particular algorithm to solve a problem as the size of the input increases. Furthermore, using big-O notation, we can compare two algorithms to determine which is more efficient as the size of the input grows. For instance, if we have two algorithms for solving a problem, one using $100n^2 + 17n + 4$ operations and the other using n^3 operations, big-O notation can help us see that the first algorithm uses far fewer operations when n is large, even though it uses more operations for small values of n, such as $n = 10$.

This section introduces big-O notation and the related big-Omega and big-Theta notations. We will explain how big-O, big-Omega, and big-Theta estimates are constructed and establish estimates for some important functions that are used in the analysis of algorithms.

3.2.2 Big-O Notation

The growth of functions is often described using a special notation. Definition 1 describes this notation.

Definition 1

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-oh of $g(x)$."]

Remark: Intuitively, the definition that $f(x)$ is $O(g(x))$ says that $f(x)$ grows slower than some fixed multiple of $g(x)$ as x grows without bound.

Assessment

The constants C and k in the definition of big-O notation are called **witnesses** to the relationship $f(x)$ is $O(g(x))$. To establish that $f(x)$ is $O(g(x))$ we need only one pair of witnesses to this relationship. That is, to show that $f(x)$ is $O(g(x))$, we need find only *one* pair of constants C and k, the witnesses, such that $|f(x)| \leq C|g(x)|$ whenever $x > k$.

Links

Note that when there is one pair of witnesses to the relationship $f(x)$ is $O(g(x))$, there are *infinitely many* pairs of witnesses. To see this, note that if C and k are one pair of witnesses, then any pair C' and k', where $C < C'$ and $k < k'$, is also a pair of witnesses, because $|f(x)| \leq C|g(x)| \leq C'|g(x)|$ whenever $x > k' > k$.

THE HISTORY OF BIG-O NOTATION Big-O notation has been used in mathematics for more than a century. In computer science it is widely used in the analysis of algorithms, as will be seen in Section 3.3. The German mathematician Paul Bachmann first introduced big-O notation in 1892 in an important book on number theory. The big-O symbol is sometimes called a **Landau symbol** after the German mathematician Edmund Landau, who used this notation throughout his work. The use of big-O notation in computer science was popularized by Donald Knuth, who also introduced the big-Ω and big-Θ notations defined later in this section.

WORKING WITH THE DEFINITION OF BIG-*O* NOTATION A useful approach for finding a pair of witnesses is to first select a value of k for which the size of $|f(x)|$ can be readily estimated when $x > k$ and to see whether we can use this estimate to find a value of C for which $|f(x)| \leq C|g(x)|$ for $x > k$. This approach is illustrated in Example 1.

EXAMPLE 1 Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

Extra Examples

Solution: We observe that we can readily estimate the size of $f(x)$ when $x > 1$ because $x < x^2$ and $1 < x^2$ when $x > 1$. It follows that

$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

whenever $x > 1$, as shown in Figure 1. Consequently, we can take $C = 4$ and $k = 1$ as witnesses to show that $f(x)$ is $O(x^2)$. That is, $f(x) = x^2 + 2x + 1 < 4x^2$ whenever $x > 1$. (Note that it is not necessary to use absolute values here because all functions in these equalities are positive when x is positive.)

Alternatively, we can estimate the size of $f(x)$ when $x > 2$. When $x > 2$, we have $2x \leq x^2$ and $1 \leq x^2$. Consequently, if $x > 2$, we have

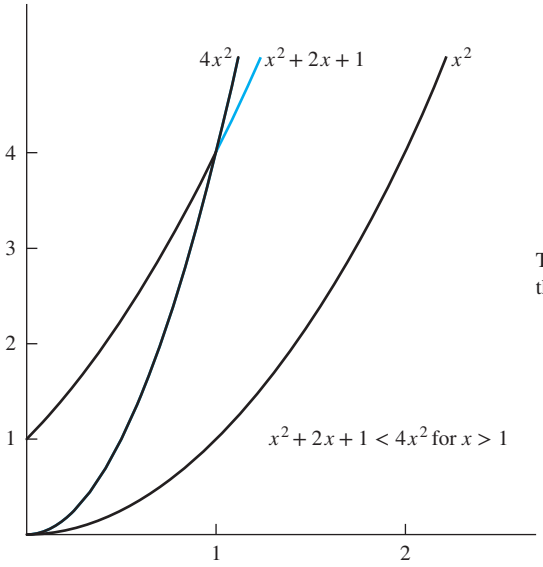
$$0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2.$$

It follows that $C = 3$ and $k = 2$ are also witnesses to the relation $f(x)$ is $O(x^2)$.

Observe that in the relationship "$f(x)$ is $O(x^2)$," x^2 can be replaced by any function that has larger values than x^2 for all $x \geq k$ for some positive real number k. For example, $f(x)$ is $O(x^3)$, $f(x)$ is $O(x^2 + x + 7)$, and so on.

It is also true that x^2 is $O(x^2 + 2x + 1)$, because $x^2 < x^2 + 2x + 1$ whenever $x > 1$. This means that $C = 1$ and $k = 1$ are witnesses to the relationship x^2 is $O(x^2 + 2x + 1)$. ◀

Note that in Example 1 we have two functions, $f(x) = x^2 + 2x + 1$ and $g(x) = x^2$, such that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$—the latter fact following from the inequality $x^2 \leq x^2 + 2x + 1$, which holds for all nonnegative real numbers x. We say that two functions



The part of the graph of $f(x) = x^2 + 2x + 1$ that satisfies $f(x) < 4x^2$ is shown in color.

FIGURE 1 **The function $x^2 + 2x + 1$ is $O(x^2)$.**

$f(x)$ and $g(x)$ that satisfy both of these big-O relationships are of the **same order**. We will return to this notion later in this section.

Remark: The fact that $f(x)$ is $O(g(x))$ is sometimes written $f(x) = O(g(x))$. However, the equals sign in this notation does *not* represent a genuine equality. Rather, this notation tells us that an inequality holds relating the values of the functions f and g for sufficiently large numbers in the domains of these functions. However, it is acceptable to write $f(x) \in O(g(x))$ because $O(g(x))$ represents the set of functions that are $O(g(x))$.

When $f(x)$ is $O(g(x))$, and $h(x)$ is a function that has larger absolute values than $g(x)$ does for sufficiently large values of x, it follows that $f(x)$ is $O(h(x))$. In other words, the function $g(x)$ in the relationship $f(x)$ is $O(g(x))$ can be replaced by a function with larger absolute values. To see this, note that if

$$|f(x)| \leq C|g(x)| \qquad \text{if } x > k,$$

and if $|h(x)| > |g(x)|$ for all $x > k$, then

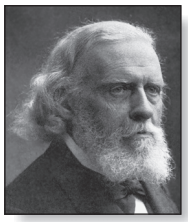
$$|f(x)| \leq C|h(x)| \qquad \text{if } x > k.$$

Hence, $f(x)$ is $O(h(x))$.

When big-O notation is used, the function g in the relationship $f(x)$ is $O(g(x))$ is often chosen to have the smallest growth rate of the functions belonging to a set of reference functions, such as functions of the form x^n, where n is a positive real number. (Important reference functions are discussed later in this section.)

In subsequent discussions, we will almost always deal with functions that take on only positive values. All references to absolute values can be dropped when working with big-O estimates for such functions. Figure 2 illustrates the relationship $f(x)$ is $O(g(x))$.

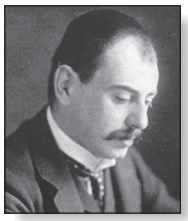
Links



Source: Bachmann, Paul. Die Arithmetik Der Quadratischen Formen. Verlag und Druck von BG Teubner. Leipzig. Berlin. 1923

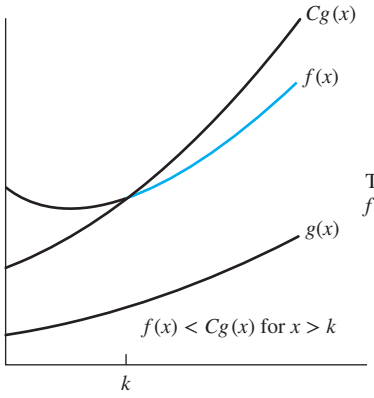
PAUL GUSTAV HEINRICH BACHMANN (1837–1920) Paul Bachmann, the son of a Lutheran pastor, shared his father's pious lifestyle and love of music. His mathematical talent was discovered by one of his teachers, even though he had difficulties with some of his early mathematical studies. After recuperating from tuberculosis in Switzerland, Bachmann studied mathematics, first at the University of Berlin and later at Göttingen, where he attended lectures presented by the famous number theorist Dirichlet. He received his doctorate under the German number theorist Kummer in 1862; his thesis was on group theory. Bachmann was a professor at Breslau and later at Münster. After he retired from his professorship, he continued his mathematical writing, played the piano, and served as a music critic for newspapers. Bachmann's mathematical writings include a five-volume survey of results and methods in number theory, a two-volume work on elementary number theory, a book on irrational numbers, and a book on the famous conjecture known as Fermat's Last Theorem. He introduced big-O notation in his 1892 book *Analytische Zahlentheorie.*

Links



Source: Smith Collection, Rare Book & Manuscript Library, Columbia University in the City of New York

EDMUND LANDAU (1877–1938) Edmund Landau, the son of a Berlin gynecologist, attended high school and university in Berlin. He received his doctorate in 1899, under the direction of Frobenius. Landau first taught at the University of Berlin and then moved to Göttingen, where he was a full professor until the Nazis forced him to stop teaching. Landau's main contributions to mathematics were in the field of analytic number theory. In particular, he established several important results concerning the distribution of primes. He authored a three-volume exposition on number theory as well as other books on number theory and mathematical analysis.



The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.

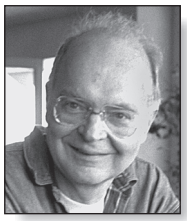
FIGURE 2 **The function $f(x)$ is $O(g(x))$.**

Example 2 illustrates how big-O notation is used to estimate the growth of functions.

EXAMPLE 2 Show that $7x^2$ is $O(x^3)$.

Solution: Note that when $x > 7$, we have $7x^2 < x^3$. (We can obtain this inequality by multiplying both sides of $x > 7$ by x^2.) Consequently, we can take $C = 1$ and $k = 7$ as witnesses to establish the relationship $7x^2$ is $O(x^3)$. Alternatively, when $x > 1$, we have $7x^2 < 7x^3$, so that $C = 7$ and $k = 1$ are also witnesses to the relationship $7x^2$ is $O(x^3)$. ◀

Links



Courtesy of Stanford University News Service

DONALD E. KNUTH (BORN 1938) Knuth grew up in Milwaukee, where his father taught bookkeeping at a Lutheran high school and owned a small printing business. He was an excellent student, earning academic achievement awards. He applied his intelligence in unconventional ways, winning a contest when he was in the eighth grade by finding over 4500 words that could be formed from the letters in "Ziegler's Giant Bar." This won a television set for his school and a candy bar for everyone in his class.

Knuth had a difficult time choosing physics over music as his major at the Case Institute of Technology. He then switched from physics to mathematics, and in 1960 he received his bachelor of science degree, simultaneously receiving a master of science degree by a special award of the faculty who considered his work outstanding. At Case, he managed the basketball team and applied his talents by constructing a formula for the value of each player. This novel approach was covered by *Newsweek* and by Walter Cronkite on the CBS television network. Knuth began graduate work at the California Institute of Technology in 1960 and received his Ph.D. there in 1963. During this time he worked as a consultant, writing compilers for different computers.

Knuth joined the staff of the California Institute of Technology in 1963, where he remained until 1968, when he took a job as a full professor at Stanford University. He retired as Professor Emeritus in 1992 to concentrate on writing. He is especially interested in updating and completing new volumes of his series *The Art of Computer Programming,* a work that has had a profound influence on the development of computer science, which he began writing as a graduate student in 1962, focusing on compilers. In common jargon, "Knuth," referring to *The Art of Computer Programming*, has come to mean the reference that answers all questions about such topics as data structures and algorithms.

Knuth is the founder of the modern study of computational complexity. He has made fundamental contributions to the subject of compilers. His dissatisfaction with mathematics typography sparked him to invent the now widely used TeX and Metafont systems. TeX has become a standard language for computer typography. Two of the many awards Knuth has received are the 1974 Turing Award and the 1979 National Medal of Technology, awarded to him by President Carter.

Knuth has written for a wide range of professional journals in computer science and in mathematics. However, his first publication, in 1957, when he was a college freshman, was a parody of the metric system called "The Potrzebie Systems of Weights and Measures," which appeared in *MAD Magazine* and has been in reprint several times. He is a church organist, as his father was. He is also a composer of music for the organ. Knuth believes that writing computer programs can be an aesthetic experience, much like writing poetry or composing music.

Knuth pays $2.56 for the first person to find each error in his books and $0.32 for significant suggestions. If you send him a letter with an error (you will need to use regular mail, because he has given up reading e-mail), he will eventually inform you whether you were the first person to tell him about this error. Be prepared for a long wait, because he receives an overwhelming amount of mail. (The author received a letter years after sending an error report to Knuth, noting that this report arrived several months after the first report of this error.)

Remark: In Example 2 we did not choose the smallest possible power of x the reference function in the big-O estimate. Note that $7x^2$ is also big-O of x^2 and x^2 grows much slower than x^3. In fact, x^2 would be the smallest possible power of x suitable as the reference function in the big-O estimate.

Example 3 illustrates how to show that a big-O relationship does not hold.

EXAMPLE 3 Show that n^2 is not $O(n)$.

Solution: To show that n^2 is not $O(n)$, we must show that no pair of witnesses C and k exist such that $n^2 \leq Cn$ whenever $n > k$. We will use a proof by contradiction to show this.

Suppose that there are constants C and k for which $n^2 \leq Cn$ whenever $n > k$. Observe that when $n > 0$ we can divide both sides of the inequality $n^2 \leq Cn$ by n to obtain the equivalent inequality $n \leq C$. However, no matter what C and k are, the inequality $n \leq C$ cannot hold for all n with $n > k$. In particular, once we set a value of k, we see that when n is larger than the maximum of k and C, it is not true that $n \leq C$ even though $n > k$. This contradiction shows that n^2 is not $O(n)$. ◀

EXAMPLE 4 Example 2 shows that $7x^2$ is $O(x^3)$. Is it also true that x^3 is $O(7x^2)$?

Solution: To determine whether x^3 is $O(7x^2)$, we need to determine whether witnesses C and k exist, so that $x^3 \leq C(7x^2)$ whenever $x > k$. We will show that no such witnesses exist using a proof by contradiction.

If C and k are witnesses, the inequality $x^3 \leq C(7x^2)$ holds for all $x > k$. Observe that the inequality $x^3 \leq C(7x^2)$ is equivalent to the inequality $x \leq 7C$, which follows by dividing both sides by the positive quantity x^2. However, no matter what C is, it is not the case that $x \leq 7C$ for all $x > k$ no matter what k is, because x can be made arbitrarily large. It follows that no witnesses C and k exist for this proposed big-O relationship. Hence, x^3 is *not* $O(7x^2)$. ◀

3.2.3 Big-O Estimates for Some Important Functions

Polynomials can often be used to estimate the growth of functions. Instead of analyzing the growth of polynomials each time they occur, we would like a result that can always be used to estimate the growth of a polynomial. Theorem 1 does this. It shows that the leading term of a polynomial dominates its growth by asserting that a polynomial of degree n or less is $O(x^n)$.

THEOREM 1 Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, where $a_0, a_1, \ldots, a_{n-1}, a_n$ are real numbers. Then $f(x)$ is $O(x^n)$.

Proof: Using the triangle inequality (see Exercise 9 in Section 1.8), if $x > 1$ we have

$$\begin{aligned} |f(x)| &= |a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0| \\ &\leq |a_n|x^n + |a_{n-1}|x^{n-1} + \cdots + |a_1|x + |a_0| \\ &= x^n \left(|a_n| + |a_{n-1}|/x + \cdots + |a_1|/x^{n-1} + |a_0|/x^n\right) \\ &\leq x^n \left(|a_n| + |a_{n-1}| + \cdots + |a_1| + |a_0|\right). \end{aligned}$$

This shows that

$$|f(x)| \leq Cx^n,$$

where $C = |a_n| + |a_{n-1}| + \cdots + |a_0|$ whenever $x > 1$. Hence, the witnesses $C = |a_n| + |a_{n-1}| + \cdots + |a_0|$ and $k = 1$ show that $f(x)$ is $O(x^n)$. ◁

We now give some examples involving functions that have the set of positive integers as their domains.

EXAMPLE 5 How can big-O notation be used to estimate the sum of the first n positive integers?

Solution: Because each of the integers in the sum of the first n positive integers does not exceed n, it follows that

$$1 + 2 + \cdots + n \leq n + n + \cdots + n = n^2.$$

From this inequality it follows that $1 + 2 + 3 + \cdots + n$ is $O(n^2)$, taking $C = 1$ and $k = 1$ as witnesses. (In this example the domains of the functions in the big-O relationship are the set of positive integers.) ◀

In Example 6 big-O estimates will be developed for the factorial function and its logarithm. These estimates will be important in the analysis of the number of steps used in sorting procedures.

EXAMPLE 6 Give big-O estimates for the factorial function and the logarithm of the factorial function, where the factorial function $f(n) = n!$ is defined by

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$$

whenever n is a positive integer, and $0! = 1$. For example,

$$1! = 1, \quad 2! = 1 \cdot 2 = 2, \quad 3! = 1 \cdot 2 \cdot 3 = 6, \quad 4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24.$$

Note that the function $n!$ grows rapidly. For instance,

$$20! = 2{,}432{,}902{,}008{,}176{,}640{,}000.$$

Solution: A big-O estimate for $n!$ can be obtained by noting that each term in the product does not exceed n. Hence,

$$\begin{aligned} n! &= 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n \\ &\leq n \cdot n \cdot n \cdot \cdots \cdot n \\ &= n^n. \end{aligned}$$

This inequality shows that $n!$ is $O(n^n)$, taking $C = 1$ and $k = 1$ as witnesses. Taking logarithms of both sides of the inequality established for $n!$, we obtain

$$\log n! \leq \log n^n = n \log n.$$

This implies that $\log n!$ is $O(n \log n)$, again taking $C = 1$ and $k = 1$ as witnesses. ◀

EXAMPLE 7 In Section 5.1 , we will show that $n < 2^n$ whenever n is a positive integer. Show that this inequality implies that n is $O(2^n)$, and use this inequality to show that $\log n$ is $O(n)$.

Solution: Using the inequality $n < 2^n$, we quickly can conclude that n is $O(2^n)$ by taking $k = C = 1$ as witnesses. Note that because the logarithm function is increasing, taking logarithms (base 2) of both sides of this inequality shows that

$$\log n < n.$$

It follows that

$$\log n \text{ is } O(n).$$

(Again we take $C = k = 1$ as witnesses.)

If we have logarithms to a base b, where b is different from 2, we still have $\log_b n$ is $O(n)$ because

$$\log_b n = \frac{\log n}{\log b} < \frac{n}{\log b}$$

whenever n is a positive integer. We take $C = 1/\log b$ and $k = 1$ as witnesses. (We have used Theorem 3 in Appendix 2 to see that $\log_b n = \log n / \log b$.) ◀

As mentioned before, big-O notation is used to estimate the number of operations needed to solve a problem using a specified procedure or algorithm. The functions used in these estimates often include the following:

$$1,\ \log n,\ n,\ n \log n,\ n^2,\ 2^n,\ n!$$

Using calculus it can be shown that each function in the list is smaller than the succeeding function, in the sense that the ratio of a function and the succeeding function tends to zero as n grows without bound. Figure 3 displays the graphs of these functions, using a scale for the values of the functions that doubles for each successive marking on the graph. That is, the vertical scale in this graph is logarithmic.

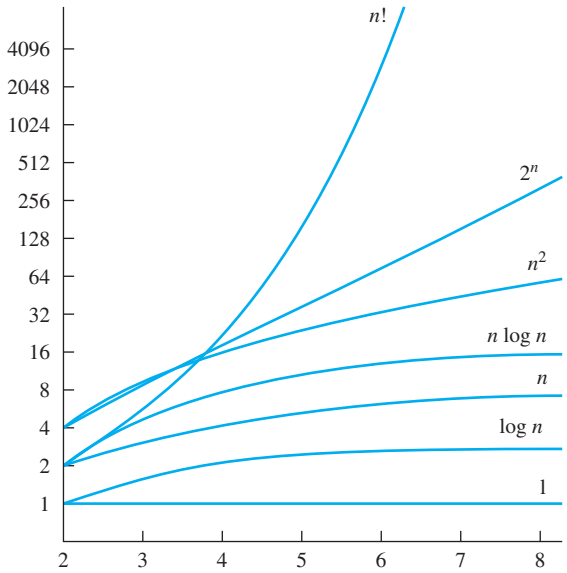


FIGURE 3 **A display of the growth of functions commonly used in big-*O* estimates.**

USEFUL BIG-O ESTIMATES INVOLVING LOGARITHMS, POWERS, AND EXPONENTIAL FUNCTIONS We now give some useful facts that help us determine whether big-O relationships hold between pairs of functions when each of the functions is a power of a logarithm, a power, or an exponential function of the form b^n where $b > 1$. Their proofs are left as Exercises 57–62 for readers skilled with calculus.

Theorem 1 shows that if $f(n)$ is a polynomial of degree d or less, then $f(n)$ is $O(n^d)$. Applying this theorem, we see that if $d > c > 1$, then n^c is $O(n^d)$. We leave it to the reader to show that the reverse of this relationship does not hold. Putting these facts together, we see that if $d > c > 1$, then

$$n^c \text{ is } O(n^d), \text{ but } n^d \text{ is not } O(n^c).$$

In Example 7 we showed that $\log_b n$ is $O(n)$ whenever $b > 1$. More generally, whenever $b > 1$ and c and d are positive, we have

$$(\log_b n)^c \text{ is } O(n^d), \text{ but } n^d \text{ is not } (O(\log_b n)^c).$$

This tells us that every positive power of the logarithm of n to the base b, where $b > 1$, is big-O of every positive power of n, but the reverse relationship never holds.

In Example 7, we also showed that n is $O(2^n)$. More generally, whenever d is positive and $b > 1$, we have

$$n^d \text{ is } O(b^n), \text{ but } b^n \text{ is not } O(n^d).$$

This tells us that every power of n is big-O of every exponential function of n with a base that is greater than one, but the reverse relationship never holds. Furthermore, when $c > b > 1$ we have

$$b^n \text{ is } O(c^n), \text{ but } c^n \text{ is not } O(b^n).$$

This tells us that if we have two exponential functions with different bases greater than one, one of these functions is big-O of the other if and only if its base is smaller or equal.

Finally, we note that if $c > 1$, we have

$$c^n \text{ is } O(n!), \text{ but } n! \text{ is not } O(c^n).$$

We can use the big-O estimates discussed here to help us order the growth of different functions, as Example 8 illustrates.

EXAMPLE 8 Arrange the functions $f_1(n) = 8\sqrt{n}, f_2(n) = (\log n)^2, f_3(n) = 2n \log n, f_4(n) = n!, f_5(n) = (1.1)^n$, and $f_6(n) = n^2$ in a list so that each function is big-O of the next function.

Solution: From the big-O estimates described in this subsection, we see that $f_2(n) = (\log n)^2$ is the slowest growing of these functions. (This follows because $\log n$ grows slower than any positive power of n.) The next three functions, in order, are $f_1(n) = 8\sqrt{n} = f_3(n) = 2n \log n$, and $f_6(n) = n^2$. (We know this because $f_1(n) = 8n^{1/2}$, $f_3(n) = 2n \log n$ is a function that grows faster than n but slower than n^c for every $c > 1$, and $f_6(n) = n^2$ is of the form n^c where $c = 2$.) The next function in the list is $f_5(n) = (1.1)^n$, because it is an exponential function with base 1.1. Finally, $f_4(n) = n!$ is the fastest growing function on the list, because $f(n) = n!$ grows faster than any exponential function of n. ◀

3.2.4 The Growth of Combinations of Functions

Many algorithms are made up of two or more separate subprocedures. The number of steps used by a computer to solve a problem with input of a specified size using such an algorithm is the sum of the number of steps used by these subprocedures. To give a big-O estimate for the number of steps needed, it is necessary to find big-O estimates for the number of steps used by each subprocedure and then combine these estimates.

Big-O estimates of combinations of functions can be provided if care is taken when different big-O estimates are combined. In particular, it is often necessary to estimate the growth of the sum and the product of two functions. What can be said if big-O estimates for each of two functions are known? To see what sort of estimates hold for the sum and the product of two functions, suppose that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$.

From the definition of big-O notation, there are constants C_1, C_2, k_1, and k_2 such that

$$|f_1(x)| \leq C_1|g_1(x)|$$

when $x > k_1$, and

$$|f_2(x)| \leq C_2|g_2(x)|$$

when $x > k_2$. To estimate the sum of $f_1(x)$ and $f_2(x)$, note that

$$\begin{aligned} |(f_1 + f_2)(x)| &= |f_1(x) + f_2(x)| \\ &\leq |f_1(x)| + |f_2(x)| \quad \text{using the triangle inequality } |a+b| \leq |a| + |b|. \end{aligned}$$

When x is greater than both k_1 and k_2, it follows from the inequalities for $|f_1(x)|$ and $|f_2(x)|$ that

$$\begin{aligned} |f_1(x)| + |f_2(x)| &\leq C_1|g_1(x)| + C_2|g_2(x)| \\ &\leq C_1|g(x)| + C_2|g(x)| \\ &= (C_1 + C_2)|g(x)| \\ &= C|g(x)|, \end{aligned}$$

where $C = C_1 + C_2$ and $g(x) = \max(|g_1(x)|, |g_2(x)|)$. [Here $\max(a, b)$ denotes the maximum, or larger, of a and b.]

This inequality shows that $|(f_1 + f_2)(x)| \leq C|g(x)|$ whenever $x > k$, where $k = \max(k_1, k_2)$. We state this useful result as Theorem 2.

THEOREM 2 Suppose that $f_1(x)$ is $O(g_1(x))$ and that $f_2(x)$ is $O(g_2(x))$. Then $(f_1 + f_2)(x)$ is $O(g(x))$, where $g(x) = (\max(|g_1(x)|, |g_2(x)|)$ for all x.

We often have big-O estimates for f_1 and f_2 in terms of the same function g. In this situation, Theorem 2 can be used to show that $(f_1 + f_2)(x)$ is also $O(g(x))$, because $\max(g(x), g(x)) = g(x)$. This result is stated in Corollary 1.

COROLLARY 1 Suppose that $f_1(x)$ and $f_2(x)$ are both $O(g(x))$. Then $(f_1 + f_2)(x)$ is $O(g(x))$.

In a similar way big-O estimates can be derived for the product of the functions f_1 and f_2. When x is greater than $\max(k_1, k_2)$ it follows that

$$\begin{aligned}|(f_1f_2)(x)| &= |f_1(x)||f_2(x)| \\ &\leq C_1|g_1(x)|C_2|g_2(x)| \\ &\leq C_1C_2|(g_1g_2)(x)| \\ &\leq C|(g_1g_2)(x)|,\end{aligned}$$

where $C = C_1C_2$. From this inequality, it follows that $f_1(x)f_2(x)$ is $O(g_1g_2(x))$, because there are constants C and k, namely, $C = C_1C_2$ and $k = \max(k_1, k_2)$, such that $|(f_1f_2)(x)| \leq C|g_1(x)g_2(x)|$ whenever $x > k$. This result is stated in Theorem 3.

THEOREM 3 Suppose that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$. Then $(f_1f_2)(x)$ is $O(g_1(x)g_2(x))$.

The goal in using big-O notation to estimate functions is to choose a function $g(x)$ as simple as possible, that grows relatively slowly so that $f(x)$ is $O(g(x))$. Examples 9 and 10 illustrate how to use Theorems 2 and 3 to do this. The type of analysis given in these examples is often used in the analysis of the time used to solve problems using computer programs.

EXAMPLE 9 Give a big-O estimate for $f(n) = 3n\log(n!) + (n^2 + 3)\log n$, where n is a positive integer.

Solution: First, the product $3n\log(n!)$ will be estimated. From Example 6 we know that $\log(n!)$ is $O(n\log n)$. Using this estimate and the fact that $3n$ is $O(n)$, Theorem 3 gives the estimate that $3n\log(n!)$ is $O(n^2\log n)$.

Next, the product $(n^2 + 3)\log n$ will be estimated. Because $(n^2 + 3) < 2n^2$ when $n > 2$, it follows that $n^2 + 3$ is $O(n^2)$. Thus, from Theorem 3 it follows that $(n^2 + 3)\log n$ is $O(n^2\log n)$. Using Theorem 2 to combine the two big-O estimates for the products shows that $f(n) = 3n\log(n!) + (n^2 + 3)\log n$ is $O(n^2\log n)$. ◀

EXAMPLE 10 Give a big-O estimate for $f(x) = (x + 1)\log(x^2 + 1) + 3x^2$.

Solution: First, a big-O estimate for $(x + 1)\log(x^2 + 1)$ will be found. Note that $(x + 1)$ is $O(x)$. Furthermore, $x^2 + 1 \leq 2x^2$ when $x > 1$. Hence,

$$\log(x^2 + 1) \leq \log(2x^2) = \log 2 + \log x^2 = \log 2 + 2\log x \leq 3\log x,$$

if $x > 2$. This shows that $\log(x^2 + 1)$ is $O(\log x)$.

From Theorem 3 it follows that $(x + 1)\log(x^2 + 1)$ is $O(x\log x)$. Because $3x^2$ is $O(x^2)$, Theorem 2 tells us that $f(x)$ is $O(\max(x\log x, x^2))$. Because $x\log x \leq x^2$, for $x > 1$, it follows that $f(x)$ is $O(x^2)$. ◀

3.2.5 Big-Omega and Big-Theta Notation

Big-O notation is used extensively to describe the growth of functions, but it has limitations. In particular, when $f(x)$ is $O(g(x))$, we have an upper bound, in terms of $g(x)$, for the size of $f(x)$ for large values of x. However, big-O notation does not provide a lower bound for the size of $f(x)$ for large x. For this, we use **big-Omega (big-Ω) notation**. When we want to give both an upper and a lower bound on the size of a function $f(x)$, relative to a reference function $g(x)$, we use **big-Theta (big-Θ) notation**. Both big-Omega and big-Theta notation were introduced

Ω and Θ are the Greek uppercase letters omega and theta, respectively.

by Donald Knuth in the 1970s. His motivation for introducing these notations was the common misuse of big-O notation when both an upper and a lower bound on the size of a function are needed. We now define big-Omega notation and illustrate its use. After doing so, we will do the same for big-Theta notation.

Definition 2

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are constants C and k with C positive such that

$$|f(x)| \geq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-Omega of $g(x)$."]

There is a strong connection between big-O and big-Omega notation. In particular, $f(x)$ is $\Omega(g(x))$ if and only if $g(x)$ is $O(f(x))$. We leave the verification of this fact as a straightforward exercise for the reader.

EXAMPLE 11 The function $f(x) = 8x^3 + 5x^2 + 7$ is $\Omega(g(x))$, where $g(x)$ is the function $g(x) = x^3$. This is easy to see because $f(x) = 8x^3 + 5x^2 + 7 \geq 8x^3$ for all positive real numbers x. This is equivalent to saying that $g(x) = x^3$ is $O(8x^3 + 5x^2 + 7)$, which can be established directly by turning the inequality around. ◀

Often, it is important to know the order of growth of a function in terms of some relatively simple reference function such as x^n when n is a positive integer or c^x, where $c > 1$. Knowing the order of growth requires that we have both an upper bound and a lower bound for the size of the function. That is, given a function $f(x)$, we want a reference function $g(x)$ such that $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$. Big-Theta notation, defined as follows, is used to express both of these relationships, providing both an upper and a lower bound on the size of a function.

Definition 3

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$. When $f(x)$ is $\Theta(g(x))$, we say that f is big-Theta of $g(x)$, that $f(x)$ is of *order* $g(x)$, and that $f(x)$ and $g(x)$ are of the *same order*.

When $f(x)$ is $\Theta(g(x))$, it is also the case that $g(x)$ is $\Theta(f(x))$. Also note that $f(x)$ is $\Theta(g(x))$ if and only if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$ (see Exercise 31). Furthermore, note that $f(x)$ is $\Theta(g(x))$ if and only if there are positive real numbers C_1 and C_2 and a positive real number k such that

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

whenever $x > k$. The existence of the constants C_1, C_2, and k tells us that $f(x)$ is $\Omega(g(x))$ and that $f(x)$ is $O(g(x))$, respectively.

Usually, when big-Theta notation is used, the function $g(x)$ in $\Theta(g(x))$ is a relatively simple reference function, such as x^n, c^x, $\log x$, and so on, while $f(x)$ can be relatively complicated.

EXAMPLE 12 We showed (in Example 5) that the sum of the first n positive integers is $O(n^2)$. Determine whether this sum is of order n^2 without using the summation formula for this sum.

Extra Examples ❯

Solution: Let $f(n) = 1 + 2 + 3 + \cdots + n$. Because we already know that $f(n)$ is $O(n^2)$, to show that $f(n)$ is of order n^2 we need to find a positive constant C such that $f(n) > Cn^2$ for sufficiently

large integers n. To obtain a lower bound for this sum, we can ignore the first half of the terms. Summing only the terms greater than $\lceil n/2 \rceil$, we find that

$$\begin{aligned} 1 + 2 + \cdots + n &\geq \lceil n/2 \rceil + (\lceil n/2 \rceil + 1) + \cdots + n \\ &\geq \lceil n/2 \rceil + \lceil n/2 \rceil + \cdots + \lceil n/2 \rceil \\ &= (n - \lceil n/2 \rceil + 1)\lceil n/2 \rceil \\ &\geq (n/2)(n/2) \\ &= n^2/4. \end{aligned}$$

This shows that $f(n)$ is $\Omega(n^2)$. We conclude that $f(n)$ is of order n^2, or in symbols, $f(n)$ is $\Theta(n^2)$. ◀

Remark: Note that we can also show that $f(n) = \sum_{i=1}^{n} i$ is $\Theta(n^2)$ using the closed formula $\sum_{i=1}^{n} = n(n+1)/2$ from Table 2 in Section 2.4 and derived in Exercise 37(b) of that section.

EXAMPLE 13 Show that $3x^2 + 8x \log x$ is $\Theta(x^2)$.

Extra Examples ➤

Solution: Because $0 \leq 8x \log x \leq 8x^2$, it follows that $3x^2 + 8x \log x \leq 11x^2$ for $x > 1$. Consequently, $3x^2 + 8x \log x$ is $O(x^2)$. Clearly, x^2 is $O(3x^2 + 8x \log x)$. Consequently, $3x^2 + 8x \log x$ is $\Theta(x^2)$. ◀

One useful fact is that the leading term of a polynomial determines its order. For example, if $f(x) = 3x^5 + x^4 + 17x^3 + 2$, then $f(x)$ is of order x^5. This is stated in Theorem 4, whose proof is left as Exercise 50.

THEOREM 4

Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, where $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$. Then $f(x)$ is of order x^n.

EXAMPLE 14 The polynomials $3x^8 + 10x^7 + 221x^2 + 1444$, $x^{19} - 18x^4 - 10{,}112$, and $-x^{99} + 40{,}001x^{98} + 100{,}003x$ are of orders x^8, x^{19}, and x^{99}, respectively. ◀

Unfortunately, as Knuth observed, big-O notation is often used by careless writers and speakers as if it had the same meaning as big-Theta notation. Keep this in mind when you see big-O notation used. The recent trend has been to use big-Theta notation whenever both upper and lower bounds on the size of a function are needed.

Exercises

In Exercises 1–14, to establish a big-O relationship, find witnesses C and k such that $|f(x)| \leq C|g(x)|$ whenever $x > k$.

1. Determine whether each of these functions is $O(x)$.

a) $f(x) = 10$ **b)** $f(x) = 3x + 7$
c) $f(x) = x^2 + x + 1$ **d)** $f(x) = 5 \log x$
e) $f(x) = \lfloor x \rfloor$ **f)** $f(x) = \lceil x/2 \rceil$

2. Determine whether each of these functions is $O(x^2)$.

a) $f(x) = 17x + 11$ **b)** $f(x) = x^2 + 1000$
c) $f(x) = x \log x$ **d)** $f(x) = x^4/2$
e) $f(x) = 2^x$ **f)** $f(x) = \lfloor x \rfloor \cdot \lceil x \rceil$

3. Use the definition of "$f(x)$ is $O(g(x))$" to show that $x^4 + 9x^3 + 4x + 7$ is $O(x^4)$.

4. Use the definition of "$f(x)$ is $O(g(x))$" to show that $2^x + 17$ is $O(3^x)$.

5. Show that $(x^2 + 1)/(x + 1)$ is $O(x)$.

6. Show that $(x^3 + 2x)/(2x + 1)$ is $O(x^2)$.

7. Find the least integer n such that $f(x)$ is $O(x^n)$ for each of these functions.
a) $f(x) = 2x^3 + x^2 \log x$
b) $f(x) = 3x^3 + (\log x)^4$
c) $f(x) = (x^4 + x^2 + 1)/(x^3 + 1)$
d) $f(x) = (x^4 + 5 \log x)/(x^4 + 1)$

8. Find the least integer n such that $f(x)$ is $O(x^n)$ for each of these functions.
a) $f(x) = 2x^2 + x^3 \log x$
b) $f(x) = 3x^5 + (\log x)^4$
c) $f(x) = (x^4 + x^2 + 1)/(x^4 + 1)$
d) $f(x) = (x^3 + 5 \log x)/(x^4 + 1)$

9. Show that $x^2 + 4x + 17$ is $O(x^3)$ but that x^3 is not $O(x^2 + 4x + 17)$.

10. Show that x^3 is $O(x^4)$ but that x^4 is not $O(x^3)$.

11. Show that $3x^4 + 1$ is $O(x^4/2)$ and $x^4/2$ is $O(3x^4 + 1)$.

12. Show that $x \log x$ is $O(x^2)$ but that x^2 is not $O(x \log x)$.

13. Show that 2^n is $O(3^n)$ but that 3^n is not $O(2^n)$. (Note that this is a special case of Exercise 60.)

14. Determine whether x^3 is $O(g(x))$ for each of these functions $g(x)$.
a) $g(x) = x^2$ **b)** $g(x) = x^3$
c) $g(x) = x^2 + x^3$ **d)** $g(x) = x^2 + x^4$
e) $g(x) = 3^x$ **f)** $g(x) = x^3/2$

15. Explain what it means for a function to be $O(1)$.

16. Show that if $f(x)$ is $O(x)$, then $f(x)$ is $O(x^2)$.

17. Suppose that $f(x)$, $g(x)$, and $h(x)$ are functions such that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(h(x))$. Show that $f(x)$ is $O(h(x))$.

18. Let k be a positive integer. Show that $1^k + 2^k + \cdots + n^k$ is $O(n^{k+1})$.

19. Determine whether each of the functions 2^{n+1} and 2^{2n} is $O(2^n)$.

20. Determine whether each of the functions $\log(n + 1)$ and $\log(n^2 + 1)$ is $O(\log n)$.

21. Arrange the functions $\sqrt{n}$, $1000 \log n$, $n \log n$, $2n!$, 2^n, 3^n, and $n^2/1{,}000{,}000$ in a list so that each function is big-O of the next function.

22. Arrange the functions $(1.5)^n$, n^{100}, $(\log n)^3$, $\sqrt{n} \log n$, 10^n, $(n!)^2$, and $n^{99} + n^{98}$ in a list so that each function is big-O of the next function.

23. Suppose that you have two different algorithms for solving a problem. To solve a problem of size n, the first algorithm uses exactly $n(\log n)$ operations and the second algorithm uses exactly $n^{3/2}$ operations. As n grows, which algorithm uses fewer operations?

24. Suppose that you have two different algorithms for solving a problem. To solve a problem of size n, the first algorithm uses exactly $n^2 2^n$ operations and the second algorithm uses exactly $n!$ operations. As n grows, which algorithm uses fewer operations?

25. Give as good a big-O estimate as possible for each of these functions.
a) $(n^2 + 8)(n + 1)$ **b)** $(n \log n + n^2)(n^3 + 2)$
c) $(n! + 2^n)(n^3 + \log(n^2 + 1))$

26. Give a big-O estimate for each of these functions. For the function g in your estimate $f(x)$ is $O(g(x))$, use a simple function g of smallest order.
a) $(n^3 + n^2 \log n)(\log n + 1) + (17 \log n + 19)(n^3 + 2)$
b) $(2^n + n^2)(n^3 + 3^n)$
c) $(n^n + n2^n + 5^n)(n! + 5^n)$

27. Give a big-O estimate for each of these functions. For the function g in your estimate that $f(x)$ is $O(g(x))$, use a simple function g of the smallest order.
a) $n \log(n^2 + 1) + n^2 \log n$
b) $(n \log n + 1)^2 + (\log n + 1)(n^2 + 1)$
c) $n^{2^n} + n^{n^2}$

28. For each function in Exercise 1, determine whether that function is $\Omega(x)$ and whether it is $\Theta(x)$.

29. For each function in Exercise 2, determine whether that function is $\Omega(x^2)$ and whether it is $\Theta(x^2)$.

30. Show that each of these pairs of functions are of the same order.
a) $3x + 7$, x
b) $2x^2 + x - 7$, x^2
c) $\lfloor x + 1/2 \rfloor$, x
d) $\log(x^2 + 1)$, $\log_2 x$
e) $\log_{10} x$, $\log_2 x$

31. Show that $f(x)$ is $\Theta(g(x))$ if and only if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.

32. Show that if $f(x)$ and $g(x)$ are functions from the set of real numbers to the set of real numbers, then $f(x)$ is $O(g(x))$ if and only if $g(x)$ is $\Omega(f(x))$.

33. Show that if $f(x)$ and $g(x)$ are functions from the set of real numbers to the set of real numbers, then $f(x)$ is $\Theta(g(x))$ if and only if there are positive constants k, C_1, and C_2 such that $C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$ whenever $x > k$.

34. **a)** Show that $3x^2 + x + 1$ is $\Theta(3x^2)$ by directly finding the constants k, C_1, and C_2 in Exercise 33.
b) Express the relationship in part (a) using a picture showing the functions $3x^2 + x + 1$, $C_1 \cdot 3x^2$, and $C_2 \cdot 3x^2$, and the constant k on the x-axis, where C_1, C_2, and k are the constants you found in part (a) to show that $3x^2 + x + 1$ is $\Theta(3x^2)$.

35. Express the relationship $f(x)$ is $\Theta(g(x))$ using a picture. Show the graphs of the functions $f(x)$, $C_1|g(x)|$, and $C_2|g(x)|$, as well as the constant k on the x-axis.

36. Explain what it means for a function to be $\Omega(1)$.

37. Explain what it means for a function to be $\Theta(1)$.

38. Give a big-O estimate of the product of the first n odd positive integers.

39. Show that if f and g are real-valued functions such that $f(x)$ is $O(g(x))$, then for every positive integer n, $f^n(x)$ is $O(g^n(x))$. [Note that $f^n(x) = f(x)^n$.]

40. Show that for all real numbers a and b with $a > 1$ and $b > 1$, if $f(x)$ is $O(\log_b x)$, then $f(x)$ is $O(\log_a x)$.

41. Suppose that $f(x)$ is $O(g(x))$, where f and g are increasing and unbounded functions. Show that $\log |f(x)|$ is $O(\log |g(x)|)$.

42. Suppose that $f(x)$ is $O(g(x))$. Does it follow that $2^{f(x)}$ is $O(2^{g(x)})$?

43. Let $f_1(x)$ and $f_2(x)$ be functions from the set of real numbers to the set of positive real numbers. Show that if $f_1(x)$ and $f_2(x)$ are both $\Theta(g(x))$, where $g(x)$ is a function from the set of real numbers to the set of positive real numbers, then $f_1(x) + f_2(x)$ is $\Theta(g(x))$. Is this still true if $f_1(x)$ and $f_2(x)$ can take negative values?

44. Suppose that $f(x)$, $g(x)$, and $h(x)$ are functions such that $f(x)$ is $\Theta(g(x))$ and $g(x)$ is $\Theta(h(x))$. Show that $f(x)$ is $\Theta(h(x))$.

45. If $f_1(x)$ and $f_2(x)$ are functions from the set of positive integers to the set of positive real numbers and $f_1(x)$ and $f_2(x)$ are both $\Theta(g(x))$, is $(f_1 - f_2)(x)$ also $\Theta(g(x))$? Either prove that it is or give a counterexample.

46. Show that if $f_1(x)$ and $f_2(x)$ are functions from the set of positive integers to the set of real numbers and $f_1(x)$ is $\Theta(g_1(x))$ and $f_2(x)$ is $\Theta(g_2(x))$, then $(f_1 f_2)(x)$ is $\Theta((g_1 g_2)(x))$.

47. Find functions f and g from the set of positive integers to the set of real numbers such that $f(n)$ is not $O(g(n))$ and $g(n)$ is not $O(f(n))$.

48. Express the relationship $f(x)$ is $\Omega(g(x))$ using a picture. Show the graphs of the functions $f(x)$ and $Cg(x)$, as well as the constant k on the real axis.

49. Show that if $f_1(x)$ is $\Theta(g_1(x))$, $f_2(x)$ is $\Theta(g_2(x))$, and $f_2(x) \neq 0$ and $g_2(x) \neq 0$ for all real numbers $x > 0$, then $(f_1/f_2)(x)$ is $\Theta((g_1/g_2)(x))$.

50. Show that if $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, where $a_0, a_1, \ldots, a_{n-1}$, and a_n are real numbers and $a_n \neq 0$, then $f(x)$ is $\Theta(x^n)$.

Big-O, big-Theta, and big-Omega notation can be extended to functions in more than one variable. For example, the statement $f(x, y)$ is $O(g(x, y))$ means that there exist constants C, k_1, and k_2 such that $|f(x, y)| \leq C|g(x, y)|$ whenever $x > k_1$ and $y > k_2$.

51. Define the statement $f(x, y)$ is $\Theta(g(x, y))$.

52. Define the statement $f(x, y)$ is $\Omega(g(x, y))$.

53. Show that $(x^2 + xy + x \log y)^3$ is $O(x^6 y^3)$.

54. Show that $x^5 y^3 + x^4 y^4 + x^3 y^5$ is $\Omega(x^3 y^3)$.

55. Show that $\lfloor xy \rfloor$ is $O(xy)$.

56. Show that $\lceil xy \rceil$ is $\Omega(xy)$.

57. (*Requires calculus*) Show that if $c > d > 0$, then n^d is $O(n^c)$, but n^c is not $O(n^d)$.

58. (*Requires calculus*) Show that if $b > 1$ and c and d are positive, then $(\log_b n)^c$ is $O(n^d)$, but n^d is not $O((\log_b n)^c)$.

59. (*Requires calculus*) Show that if d is positive and $b > 1$, then n^d is $O(b^n)$, but b^n is not $O(n^d)$.

60. (*Requires calculus*) Show that if $c > b > 1$, then b^n is $O(c^n)$, but c^n is not $O(b^n)$.

61. (*Requires calculus*) Show that if $c > 1$, then c^n is $O(n!)$, but $n!$ is not $O(c^n)$.

62. (*Requires calculus*) Prove or disprove that $(2n)!$ is $O(n!)$.

The following problems deal with another type of asymptotic notation, called **little-*o*** notation. Because little-o notation is based on the concept of limits, a knowledge of calculus is needed for these problems. We say that $f(x)$ is $o(g(x))$ [read $f(x)$ is "little-oh" of $g(x)$], when

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0.$$

63. (*Requires calculus*) Show that

a) x^2 is $o(x^3)$.
b) $x \log x$ is $o(x^2)$.
c) x^2 is $o(2^x)$.
d) $x^2 + x + 1$ is not $o(x^2)$.

64. (*Requires calculus*)

a) Show that if $f(x)$ and $g(x)$ are functions such that $f(x)$ is $o(g(x))$ and c is a constant, then $cf(x)$ is $o(g(x))$, where $(cf)(x) = cf(x)$.

b) Show that if $f_1(x)$, $f_2(x)$, and $g(x)$ are functions such that $f_1(x)$ is $o(g(x))$ and $f_2(x)$ is $o(g(x))$, then $(f_1 + f_2)(x)$ is $o(g(x))$, where $(f_1 + f_2)(x) = f_1(x) + f_2(x)$.

65. (*Requires calculus*) Represent pictorially that $x \log x$ is $o(x^2)$ by graphing $x \log x$, x^2, and $x \log x / x^2$. Explain how this picture shows that $x \log x$ is $o(x^2)$.

66. (*Requires calculus*) Express the relationship $f(x)$ is $o(g(x))$ using a picture. Show the graphs of $f(x)$, $g(x)$, and $f(x)/g(x)$.

***67.** (*Requires calculus*) Suppose that $f(x)$ is $o(g(x))$. Does it follow that $2^{f(x)}$ is $o(2^{g(x)})$?

***68.** (*Requires calculus*) Suppose that $f(x)$ is $o(g(x))$. Does it follow that $\log |f(x)|$ is $o(\log |g(x)|)$?

69. (*Requires calculus*) The two parts of this exercise describe the relationship between little-o and big-O notation.

a) Show that if $f(x)$ and $g(x)$ are functions such that $f(x)$ is $o(g(x))$, then $f(x)$ is $O(g(x))$.

b) Show that if $f(x)$ and $g(x)$ are functions such that $f(x)$ is $O(g(x))$, then it does not necessarily follow that $f(x)$ is $o(g(x))$.

70. (*Requires calculus*) Show that if $f(x)$ is a polynomial of degree n and $g(x)$ is a polynomial of degree m where $m > n$, then $f(x)$ is $o(g(x))$.

71. (*Requires calculus*) Show that if $f_1(x)$ is $O(g(x))$ and $f_2(x)$ is $o(g(x))$, then $f_1(x) + f_2(x)$ is $O(g(x))$.

72. (*Requires calculus*) Let H_n be the nth **harmonic number**

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

Show that H_n is $O(\log n)$. [*Hint:* First establish the inequality

$$\sum_{j=2}^{n} \frac{1}{j} < \int_1^n \frac{1}{x} dx$$

by showing that the sum of the areas of the rectangles of height $1/j$ with base from $j-1$ to j, for $j = 2, 3, \ldots, n$, is less than the area under the curve $y = 1/x$ from 2 to n.]

***73.** Show that $n \log n$ is $O(\log n!)$.

☞ **74.** Determine whether $\log n!$ is $\Theta(n \log n)$. Justify your answer.

***75.** Show that $\log n!$ is greater than $(n \log n)/4$ for $n > 4$. [*Hint:* Begin with the inequality $n! > n(n-1)(n-2)\cdots\lceil n/2\rceil$.]

Let $f(x)$ and $g(x)$ be functions from the set of real numbers to the set of real numbers. We say that the functions f and g are **asymptotic** and write $f(x) \sim g(x)$ if $\lim_{x\to\infty} f(x)/g(x) = 1$.

76. (*Requires calculus*) For each of these pairs of functions, determine whether f and g are asymptotic.

a) $f(x) = x^2 + 3x + 7$, $g(x) = x^2 + 10$

b) $f(x) = x^2 \log x$, $g(x) = x^3$

c) $f(x) = x^4 + \log(3x^8 + 7)$, $g(x) = (x^2 + 17x + 3)^2$

d) $f(x) = (x^3 + x^2 + x + 1)^4$, $g(x) = (x^4 + x^3 + x^2 + x + 1)^3$.

77. (*Requires calculus*) For each of these pairs of functions, determine whether f and g are asymptotic.

a) $f(x) = \log(x^2 + 1)$, $g(x) = \log x$

b) $f(x) = 2^{x+3}$, $g(x) = 2^{x+7}$

c) $f(x) = 2^{2^x}$, $g(x) = 2^{x^2}$

d) $f(x) = 2^{x^2+x+1}$, $g(x) = 2^{x^2+2x}$

3.3 Complexity of Algorithms

3.3.1 Introduction

When does an algorithm provide a satisfactory solution to a problem? First, it must always produce the correct answer. How this can be demonstrated will be discussed in Chapter 5. Second, it should be efficient. The efficiency of algorithms will be discussed in this section.

How can the efficiency of an algorithm be analyzed? One measure of efficiency is the time used by a computer to solve a problem using the algorithm, when input values are of a specified size. A second measure is the amount of computer memory required to implement the algorithm when input values are of a specified size.

Questions such as these involve the **computational complexity** of the algorithm. An analysis of the time required to solve a problem of a particular size involves the **time complexity** of the algorithm. An analysis of the computer memory required involves the **space complexity** of the algorithm. Considerations of the time and space complexity of an algorithm are essential when algorithms are implemented. It is important to know whether an algorithm will produce an answer in a microsecond, a minute, or a billion years. Likewise, the required memory must be available to solve a problem, so that space complexity must be taken into account.

Considerations of space complexity are tied in with the particular data structures used to implement the algorithm. Because data structures are not dealt with in detail in this book, space complexity will not be considered. We will restrict our attention to time complexity.

3.3.2 Time Complexity

The time complexity of an algorithm can be expressed in terms of the number of operations used by the algorithm when the input has a particular size. The operations used to measure time complexity can be the comparison of integers, the addition of integers, the multiplication of integers, the division of integers, or any other basic operation.

Time complexity is described in terms of the number of operations required instead of actual computer time because of the difference in time needed for different computers to perform basic operations. Moreover, it is quite complicated to break all operations down to the basic bit operations that a computer uses. Furthermore, the fastest computers in existence can perform basic bit operations (for instance, adding, multiplying, comparing, or exchanging two bits) in 10^{-11} second (10 picoseconds), but personal computers may require 10^{-8} second (10 nanoseconds), which is 1000 times as long, to do the same operations.

We illustrate how to analyze the time complexity of an algorithm by considering Algorithm 1 of Section 3.1, which finds the maximum of a finite set of integers.

EXAMPLE 1 Describe the time complexity of Algorithm 1 of Section 3.1 for finding the maximum element in a finite set of integers.

Extra Examples

Solution: The number of comparisons will be used as the measure of the time complexity of the algorithm, because comparisons are the basic operations used.

To find the maximum element of a set with n elements, listed in an arbitrary order, the temporary maximum is first set equal to the initial term in the list. Then, after a comparison $i \leq n$ has been done to determine that the end of the list has not yet been reached, the temporary maximum and second term are compared, updating the temporary maximum to the value of the second term if it is larger. This procedure is continued, using two additional comparisons for each term of the list—one $i \leq n$, to determine that the end of the list has not been reached and another $\max < a_i$, to determine whether to update the temporary maximum. Because two comparisons are used for each of the second through the nth elements and one more comparison is used to exit the loop when $i = n + 1$, exactly $2(n - 1) + 1 = 2n - 1$ comparisons are used whenever this algorithm is applied. Hence, the algorithm for finding the maximum of a set of n elements has time complexity $\Theta(n)$, measured in terms of the number of comparisons used. Note that for this algorithm the number of comparisons is independent of particular input of n numbers. ◀

Next, we will analyze the time complexity of searching algorithms.

EXAMPLE 2 Describe the time complexity of the linear search algorithm (specified as Algortihm 2 in Section 3.1).

Solution: The number of comparisons used by Algorithm 2 in Section 3.1 will be taken as the measure of the time complexity. At each step of the loop in the algorithm, two comparisons are performed—one $i \leq n$, to see whether the end of the list has been reached and one $x \leq a_i$, to compare the element x with a term of the list. Finally, one more comparison $i \leq n$ is made outside the loop. Consequently, if $x = a_i$, $2i + 1$ comparisons are used. The most comparisons, $2n + 2$, are required when the element is not in the list. In this case, $2n$ comparisons are used to determine that x is not a_i, for $i = 1, 2, \ldots, n$, an additional comparison is used to exit the loop, and one comparison is made outside the loop. So when x is not in the list, a total of $2n + 2$ comparisons are used. Hence, a linear search requires $\Theta(n)$ comparisons in the worst case, because $2n + 2$ is $\Theta(n)$. ◀

WORST-CASE COMPLEXITY The type of complexity analysis done in Example 2 is a **worst-case** analysis. By the worst-case performance of an algorithm, we mean the largest number of operations needed to solve the given problem using this algorithm on input of specified size. Worst-case analysis tells us how many operations an algorithm requires to guarantee that it will produce a solution.

EXAMPLE 3 Describe the time complexity of the binary search algorithm (specified as Algorithm 3 in Section 3.1) in terms of the number of comparisons used (and ignoring the time required to compute $m = \lfloor (i + j)/2 \rfloor$ in each iteration of the loop in the algorithm).

Solution: For simplicity, assume there are $n = 2^k$ elements in the list $a_1, a_2, \ldots, a_n$, where k is a nonnegative integer. Note that $k = \log n$. (If n, the number of elements in the list, is not a power of 2, the list can be considered part of a larger list with 2^{k+1} elements, where $2^k < n < 2^{k+1}$. Here 2^{k+1} is the smallest power of 2 larger than n.)

At each stage of the algorithm, i and j, the locations of the first term and the last term of the restricted list at that stage, are compared to see whether the restricted list has more than one term. If $i < j$, a comparison is done to determine whether x is greater than the middle term of the restricted list.

At the first stage the search is restricted to a list with 2^{k-1} terms. So far, two comparisons have been used. This procedure is continued, using two comparisons at each stage to restrict the search to a list with half as many terms. In other words, two comparisons are used at the first stage of the algorithm when the list has 2^k elements, two more when the search has been reduced to a list with 2^{k-1} elements, two more when the search has been reduced to a list with 2^{k-2} elements, and so on, until two comparisons are used when the search has been reduced to a list with $2^1 = 2$ elements. Finally, when one term is left in the list, one comparison tells us that there are no additional terms left, and one more comparison is used to determine if this term is x.

Hence, at most $2k + 2 = 2 \log n + 2$ comparisons are required to perform a binary search when the list being searched has 2^k elements. (If n is not a power of 2, the original list is expanded to a list with 2^{k+1} terms, where $k = \lfloor \log n \rfloor$, and the search requires at most $2 \lceil \log n \rceil + 2$ comparisons.) It follows that in the worst case, binary search requires $O(\log n)$ comparisons.

Note that in the worst case, $2 \log n + 2$ comparisons are used by the binary search. Hence, the binary search uses $\Theta(\log n)$ comparisons in the worst case, because $2 \log n + 2 = \Theta(\log n)$. From this analysis it follows that in the worst case, the binary search algorithm is more efficient than the linear search algorithm, because we know by Example 2 that the linear search algorithm has $\Theta(n)$ worst-case time complexity. ◀

AVERAGE-CASE COMPLEXITY Another important type of complexity analysis, besides worst-case analysis, is called **average-case** analysis. The average number of operations used to solve the problem over all possible inputs of a given size is found in this type of analysis. Average-case time complexity analysis is usually much more complicated than worst-case analysis. However, the average-case analysis for the linear search algorithm can be done without difficulty, as shown in Example 4.

EXAMPLE 4 Describe the average-case performance of the linear search algorithm in terms of the average number of comparisons used, assuming that the integer x is in the list and it is equally likely that x is in any position.

Solution: By hypothesis, the integer x is one of the integers $a_1, a_2, \ldots, a_n$ in the list. If x is the first term a_1 of the list, three comparisons are needed, one $i \leq n$ to determine whether the end of the list has been reached, one $x \neq a_i$ to compare x and the first term, and one $i \leq n$ outside the loop. If x is the second term a_2 of the list, two more comparisons are needed, so that a total of five comparisons are used. In general, if x is the ith term of the list a_i, two comparisons will be used at each of the i steps of the loop, and one outside the loop, so that a total of $2i + 1$ comparisons are needed. Hence, the average number of comparisons used equals

$$\frac{3 + 5 + 7 + \cdots + (2n + 1)}{n} = \frac{2(1 + 2 + 3 + \cdots + n) + n}{n}.$$

Using the formula from line 2 of Table 2 in Section 2.4 (and see Exercise 37(b) of Section 2.4),

$$1 + 2 + 3 + \cdots + n = \frac{n(n + 1)}{2}.$$

Hence, the average number of comparisons used by the linear search algorithm (when x is known to be in the list) is

$$\frac{2[n(n + 1)/2]}{n} + 1 = n + 2,$$

which is $\Theta(n)$. ◀

Remark: In the analysis in Example 4 we assumed that x is in the list being searched. It is also possible to do an average-case analysis of this algorithm when x may not be in the list (see Exercise 23).

Remark: Although we have counted the comparisons needed to determine whether we have reached the end of a loop, these comparisons are often not counted. From this point on we will ignore such comparisons.

WORST-CASE COMPLEXITY OF TWO SORTING ALGORITHMS We analyze the worst-case complexity of the bubble sort and the insertion sort in Examples 5 and 6.

EXAMPLE 5 What is the worst-case complexity of the bubble sort in terms of the number of comparisons made?

Solution: The bubble sort described before Example 4 in Section 3.1 sorts a list by performing a sequence of passes through the list. During each pass the bubble sort successively compares adjacent elements, interchanging them if necessary. When the ith pass begins, the $i-1$ largest elements are guaranteed to be in the correct positions. During this pass, $n-i$ comparisons are used. Consequently, the total number of comparisons used by the bubble sort to order a list of n elements is

$$(n-1)+(n-2)+\cdots+2+1=\frac{(n-1)n}{2}$$

using a summation formula from line 2 in Table 2 in Section 2.4 (and Exercise 37(b) in Section 2.4). Note that the bubble sort always uses this many comparisons, because it continues even if the list becomes completely sorted at some intermediate step. Consequently, the bubble sort uses $(n-1)n/2$ comparisons, so it has $\Theta(n^2)$ worst-case complexity in terms of the number of comparisons used. ◀

EXAMPLE 6 What is the worst-case complexity of the insertion sort in terms of the number of comparisons made?

Solution: The insertion sort (described in Section 3.1) inserts the jth element into the correct position among the first $j-1$ elements that have already been put into the correct order. It does this by using a linear search technique, successively comparing the jth element with successive terms until a term that is greater than or equal to it is found or it compares a_j with itself and stops because a_j is not less than itself. Consequently, in the worst case, j comparisons are required to insert the jth element into the correct position. Therefore, the total number of comparisons used by the insertion sort to sort a list of n elements is

$$2+3+\cdots+n=\frac{n(n+1)}{2}-1,$$

using the summation formula for the sum of consecutive integers in line 2 of Table 2 of Section 2.4 (and see Exercise 37(b) of Section 2.4), and noting that the first term, 1, is missing in this sum. Note that the insertion sort may use considerably fewer comparisons if the smaller elements started out at the end of the list. We conclude that the insertion sort has worst-case complexity $\Theta(n^2)$. ◀

In Examples 5 and 6 we showed that both the bubble sort and the insertion sort have worst-case time complexity $\Theta(n^2)$. However, the most efficient sorting algorithms can sort n items in $O(n \log n)$ time, as we will show in Sections 8.3 and 11.1 using techniques we develop in those sections. From this point on, we will assume that sorting n items can be done in $O(n \log n)$ time.

Links

You can run animations found on many different websites that simultaneously run different sorting algorithms on the same lists. Doing so will help you gain insights into the efficiency of different sorting algorithms. Among the sorting algorithms that you can find are the bubble sort, the insertion sort, the shell sort, the merge sort, and the quick sort. Some of these animations allow you to test the relative performance of these sorting algorithms on lists of randomly selected items, lists that are nearly sorted, and lists that are in reversed order.

3.3.3 Complexity of Matrix Multiplication

The definition of the product of two matrices can be expressed as an algorithm for computing the product of two matrices. Suppose that $\mathbf{C} = [c_{ij}]$ is the $m \times n$ matrix that is the product of the $m \times k$ matrix $\mathbf{A} = [a_{ij}]$ and the $k \times n$ matrix $\mathbf{B} = [b_{ij}]$. The algorithm based on the definition of the matrix product is expressed in pseudocode in Algorithm 1.

ALGORITHM 1 Matrix Multiplication.

```
procedure matrix multiplication(A, B: matrices)
for i := 1 to m
    for j := 1 to n
        c_ij := 0
        for q := 1 to k
            c_ij := c_ij + a_iq b_qj
return C {C = [c_ij] is the product of A and B}
```

We can determine the complexity of this algorithm in terms of the number of additions and multiplications used.

EXAMPLE 7 How many additions of integers and multiplications of integers are used by Algorithm 1 to multiply two $n \times n$ matrices with integer entries?

Solution: There are n^2 entries in the product of $\mathbf{A}$ and $\mathbf{B}$. To find each entry requires a total of n multiplications and $n - 1$ additions. Hence, a total of n^3 multiplications and $n^2(n - 1)$ additions are used. ◀

Surprisingly, there are more efficient algorithms for matrix multiplication than that given in Algorithm 1. As Example 7 shows, multiplying two $n \times n$ matrices directly from the definition requires $O(n^3)$ multiplications and additions. Using other algorithms, two $n \times n$ matrices can be multiplied using $O(n^{\sqrt{7}})$ multiplications and additions. (Details of such algorithms can be found in [CoLeRiSt09].)

We can also analyze the complexity of the algorithm we described in Chapter 2 for computing the Boolean product of two matrices, which we display as Algorithm 2.

ALGORITHM 2 The Boolean Product of Zero–One Matrices.

procedure *Boolean product of Zero–One Matrices* (**A**, **B**: zero–one matrices)
for $i := 1$ **to** m
  for $j := 1$ **to** n
    $c_{ij} := 0$
    for $q := 1$ **to** k
      $c_{ij} := c_{ij} \vee (a_{iq} \wedge b_{qj})$
return C {$\mathbf{C} = [c_{ij}]$ is the Boolean product of **A** and **B**}

The number of bit operations used to find the Boolean product of two $n \times n$ matrices can be easily determined.

EXAMPLE 8 How many bit operations are used to find $\mathbf{A} \odot \mathbf{B}$, where **A** and **B** are $n \times n$ zero–one matrices?

Solution: There are n^2 entries in $\mathbf{A} \odot \mathbf{B}$. Using Algorithm 2, a total of n *OR*s and n *AND*s are used to find an entry of $\mathbf{A} \odot \mathbf{B}$. Hence, $2n$ bit operations are used to find each entry. Therefore, $2n^3$ bit operations are required to compute $\mathbf{A} \odot \mathbf{B}$ using Algorithm 2. ◀

Links

MATRIX-CHAIN MULTIPLICATION There is another important problem involving the complexity of the multiplication of matrices. How should the **matrix-chain** $\mathbf{A}_1\mathbf{A}_2 \cdots \mathbf{A}_n$ be computed using the fewest multiplications of integers, where $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n$ are $m_1 \times m_2, m_2 \times m_3, \ldots, m_n \times m_{n+1}$ matrices, respectively, and each has integers as entries? (Because matrix multiplication is associative, as shown in Exercise 13 in Section 2.6, the order of the multiplication used does not change the product.) Note that $m_1m_2m_3$ multiplications of integers are performed to multiply an $m_1 \times m_2$ matrix and an $m_2 \times m_3$ matrix using Algorithm 1. Example 9 illustrates this problem.

EXAMPLE 9 In which order should the matrices $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$—where $\mathbf{A}_1$ is 30×20, $\mathbf{A}_2$ is 20×40, and $\mathbf{A}_3$ is 40×10, all with integer entries—be multiplied to use the least number of multiplications of integers?

Solution: There are two possible ways to compute $\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3$. These are $\mathbf{A}_1(\mathbf{A}_2\mathbf{A}_3)$ and $(\mathbf{A}_1\mathbf{A}_2)\mathbf{A}_3$.

If $\mathbf{A}_2$ and $\mathbf{A}_3$ are first multiplied, a total of $20 \cdot 40 \cdot 10 = 8000$ multiplications of integers are used to obtain the 20×10 matrix $\mathbf{A}_2\mathbf{A}_3$. Then, to multiply $\mathbf{A}_1$ and $\mathbf{A}_2\mathbf{A}_3$ requires $30 \cdot 20 \cdot 10 = 6000$ multiplications. Hence, a total of

$$8000 + 6000 = 14{,}000$$

multiplications are used. On the other hand, if $\mathbf{A}_1$ and $\mathbf{A}_2$ are first multiplied, then $30 \cdot 20 \cdot 40 = 24{,}000$ multiplications are used to obtain the 30×40 matrix $\mathbf{A}_1\mathbf{A}_2$. Then, to multiply $\mathbf{A}_1\mathbf{A}_2$ and $\mathbf{A}_3$ requires $30 \cdot 40 \cdot 10 = 12{,}000$ multiplications. Hence, a total of

$$24{,}000 + 12{,}000 = 36{,}000$$

multiplications are used.

Clearly, the first method is more efficient. ◀

We will return to this problem in Exercise 57 in Section 8.1. Algorithms for determining the most efficient way to carry out matrix-chain multiplication are discussed in [CoLeRiSt09].

3.3.4 Algorithmic Paradigms

In Section 3.1 we introduced the basic notion of an algorithm. We provided examples of many different algorithms, including searching and sorting algorithms. We also introduced the concept of a greedy algorithm, giving examples of several problems that can be solved by greedy algorithms. Greedy algorithms provide an example of an **algorithmic paradigm**, that is, a general approach based on a particular concept that can be used to construct algorithms for solving a variety of problems.

In this book we will construct algorithms for solving many different problems based on a variety of algorithmic paradigms, including the most widely used algorithmic paradigms. These paradigms can serve as the basis for constructing efficient algorithms for solving a wide range of problems.

Some of the algorithms we have already studied are based on an algorithmic paradigm known as brute force, which we will describe in this section. Algorithmic paradigms, studied later in this book, include divide-and-conquer algorithms studied in Chapter 8, dynamic programming, also studied in Chapter 8, backtracking, studied in Chapter 10, and probabilistic algorithms, studied in Chapter 7. There are many important algorithmic paradigms besides those described in this book. Consult books on algorithm design such as [KlTa06] to learn more about them.

BRUTE-FORCE ALGORITHMS Brute force is an important, and basic, algorithmic paradigm. In a **brute-force algorithm**, a problem is solved in the most straightforward manner based on the statement of the problem and the definitions of terms. Brute-force algorithms are designed to solve problems without regard to the computing resources required. For example, in some brute-force algorithms the solution to a problem is found by examining every possible solution, looking for the best possible. In general, brute-force algorithms are naive approaches for solving problems that do not take advantage of any special structure of the problem or clever ideas.

Note that Algorithm 1 in Section 3.1 for finding the maximum number in a sequence is a brute-force algorithm because it examines each of the n numbers in a sequence to find the maximum term. The algorithm for finding the sum of n numbers by adding one additional number at a time is also a brute-force algorithm, as is the algorithm for matrix multiplication based on its definition (Algorithm 1). The bubble, insertion, and selection sorts (described in Section 3.1 in Algorithms 4 and 5 and in the preamble of Exercise 43, respectively) are also considered to be brute-force algorithms; all three of these sorting algorithms are straightforward approaches much less efficient than other sorting algorithms such as the merge sort and the quick sort discussed in Chapters 5 and 8.

Although brute-force algorithms are often inefficient, they are often quite useful. A brute-force algorithm may be able to solve practical instances of problems, particularly when the input is not too large, even if it is impractical to use this algorithm for larger inputs. Furthermore, when designing new algorithms to solve a problem, the goal is often to find a new algorithm that is more efficient than a brute-force algorithm. One such problem of this type is described in Example 10.

EXAMPLE 10 Construct a brute-force algorithm for finding the closest pair of points in a set of n points in the plane and provide a worst-case big-O estimate for the number of bit operations used by the algorithm.

Solution: Suppose that we are given as input the points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Recall that the distance between (x_i, y_i) and (x_j, y_j) is $\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$. A brute-force algorithm can find the closest pair of these points by computing the distances between all pairs of the n points and determining the smallest distance. (We can make one small simplification to make the computation easier; we can compute the square of the distance between pairs of points to find the

closest pair, rather than the distance between these points. We can do this because the square of the distance between a pair of points is smallest when the distance between these points is smallest.)

ALGORITHM 3 Brute-Force Algorithm for Closest Pair of Points.

procedure *closest-pair*($(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$: pairs of real numbers)
$min = \infty$
for $i := 2$ **to** n
  for $j := 1$ **to** $i - 1$
    if $(x_j - x_i)^2 + (y_j - y_i)^2 < min$ **then**
      $min := (x_j - x_i)^2 + (y_j - y_i)^2$
      closest pair $:= ((x_i, y_i), (x_j, y_j))$
return *closest pair*

To estimate the number of operations used by the algorithm, first note that there are $n(n-1)/2$ pairs of points $((x_i, y_i), (x_j, y_j))$ that we loop through (as the reader should verify). For each such pair we compute $(x_j - x_i)^2 + (y_j - y_i)^2$, compare it with the current value of *min*, and if it is smaller than *min*, replace the current value of *min* by this new value. It follows that this algorithm uses $\Theta(n^2)$ operations, in terms of arithmetic operations and comparisons.

In Chapter 8 we will devise an algorithm that determines the closest pair of points when given n points in the plane as input that has $O(n \log n)$ worst-case complexity. The original discovery of such an algorithm, much more efficient than the brute-force approach, was considered quite surprising. ◀

3.3.5 Understanding the Complexity of Algorithms

Table 1 displays some common terminology used to describe the time complexity of algorithms. For example, an algorithm that finds the largest of the first 100 terms of a list of n elements by applying Algorithm 1 to the sequence of the first 100 terms, where n is an integer with $n \geq 100$, has **constant complexity** because it uses 99 comparisons no matter what n is (as the reader can verify). The linear search algorithm has **linear** (worst-case or average-case) **complexity** and the binary search algorithm has **logarithmic** (worst-case) **complexity**. Many important algorithms have $n \log n$, or **linearithmic** (worst-case) **complexity**, such as the merge sort, which we will introduce in Chapter 4. (The word *linearithmic* is a combination of the words *linear* and log*arithmic*.)

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

An algorithm has **polynomial complexity** if it has complexity $\Theta(n^b)$, where b is an integer with $b \geq 1$. For example, the bubble sort algorithm is a polynomial-time algorithm because it uses $\Theta(n^2)$ comparisons in the worst case. An algorithm has **exponential complexity** if it has time complexity $\Theta(b^n)$, where $b > 1$. The algorithm that determines whether a compound proposition in n variables is satisfiable by checking all possible assignments of truth variables is an algorithm with exponential complexity, because it uses $\Theta(2^n)$ operations. Finally, an algorithm has **factorial complexity** if it has $\Theta(n!)$ time complexity. The algorithm that finds all orders that a traveling salesperson could use to visit n cities has factorial complexity; we will discuss this algorithm in Chapter 9.

TRACTABILITY A problem that is solvable using an algorithm with polynomial (or better) worst-case complexity is called **tractable**, because the expectation is that the algorithm will produce the solution to the problem for reasonably sized input in a relatively short time. However, if the polynomial in the big-Θ estimate has high degree (such as degree 100) or if the coefficients are extremely large, the algorithm may take an extremely long time to solve the problem. Consequently, that a problem can be solved using an algorithm with polynomial worst-case time complexity is no guarantee that the problem can be solved in a reasonable amount of time for even relatively small input values. Fortunately, in practice, the degree and coefficients of polynomials in such estimates are often small.

The situation is much worse for problems that cannot be solved using an algorithm with worst-case polynomial time complexity. Such problems are called **intractable**. Usually, but not always, an extremely large amount of time is required to solve the problem for the worst cases of even small input values. In practice, however, there are situations where an algorithm with a certain worst-case time complexity may be able to solve a problem much more quickly for most cases than for its worst case. When we are willing to allow that some, perhaps small, number of cases may not be solved in a reasonable amount of time, the average-case time complexity is a better measure of how long an algorithm takes to solve a problem. Many problems important in industry are thought to be intractable but can be practically solved for essentially all sets of input that arise in daily life. Another way that intractable problems are handled when they arise in practical applications is that instead of looking for exact solutions of a problem, approximate solutions are sought. It may be the case that fast algorithms exist for finding such approximate solutions, perhaps even with a guarantee that they do not differ by very much from an exact solution.

Some problems even exist for which it can be shown that no algorithm exists for solving them. Such problems are called **unsolvable** (as opposed to **solvable** problems that can be solved using an algorithm). The first proof that there are unsolvable problems was provided by the great English mathematician and computer scientist Alan Turing when he showed that the halting problem is unsolvable. Recall that we proved that the halting problem is unsolvable in Section 3.1. (A biography of Alan Turing and a description of some of his other work can be found in Chapter 13.)

P VERSUS NP The study of the complexity of algorithms goes far beyond what we can describe here. Note, however, that many solvable problems are believed to have the property that no algorithm with polynomial worst-case time complexity solves them, but that a solution, if known, can be checked in polynomial time. Problems for which a solution can be checked in polynomial time are said to belong to the **class NP** (tractable problems are said to belong to **class P**). The abbreviation NP stands for *nondeterministic polynomial* time. The satisfiability problem, discussed in Section 1.3, is an example of an NP problem—we can quickly verify that an assignment of truth values to the variables of a compound proposition makes it true, but no polynomial time algorithm has been discovered for finding such an assignment of truth values. (For example, an exhaustive search of all possible truth values requires $\Omega(2^n)$ bit operations where n is the number of variables in the compound proposition.)

Links

There is also an important class of problems, called **NP-complete problems**, with the property that if any of these problems can be solved by a polynomial worst-case time algorithm, then

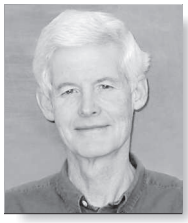
all problems in the class NP can be solved by polynomial worst-case time algorithms. The satisfiability problem is also an example of an NP-complete problem. It is an NP problem and if a polynomial time algorithm for solving it were known, there would be polynomial time algorithms for all problems known to be in this class of problems (and there are many important problems in this class). This last statement follows from the fact that every problem in NP can be reduced in polynomial time to the satisfiability problem. Although more than 3000 NP-complete problems are now known, the satisfiability problem was the first problem shown to be NP-complete. The theorem that asserts this is known as the **Cook-Levin theorem** after Stephen Cook and Leonid Levin, who independently proved it in the early 1970s.

The **P versus NP problem** asks whether NP, the class of problems for which it is possible to check solutions in polynomial time, equals P, the class of tractable problems. If P $\neq$ NP, there would be some problems that cannot be solved in polynomial time, but whose solutions could be verified in polynomial time. The concept of NP-completeness is helpful in research aimed at solving the P versus NP problem, because NP-complete problems are the problems in NP considered most likely not to be in P, as every problem in NP can be reduced to an NP-complete problem in polynomial time. A large majority of theoretical computer scientists believe that P $\neq$ NP, which would mean that no NP-complete problem can be solved in polynomial time. One reason for this belief is that despite extensive research, no one has succeeded in showing that P = NP. In particular, no one has been able to find an algorithm with worst-case polynomial time complexity that solves any NP-complete problem. The P versus NP problem is one of the most famous unsolved problems in the mathematical sciences (which include theoretical computer science). It is one of the seven famous Millennium Prize Problems, of which six remain unsolved. A prize of $1,000,000 is offered by the Clay Mathematics Institute for its solution.

For more information about the complexity of algorithms, consult the references, including [CoLeRiSt09], for this section listed at the end of this book. (Also, for a more formal discussion of computational complexity in terms of Turing machines, see Section 13.5.)

PRACTICAL CONSIDERATIONS Note that a big-Θ estimate of the time complexity of an algorithm expresses how the time required to solve the problem increases as the input grows in size. In practice, the best estimate (that is, with the smallest reference function) that can be shown is used. However, big-Θ estimates of time complexity cannot be directly translated into the actual amount of computer time used. One reason is that a big-Θ estimate $f(n)$ is $\Theta(g(n))$, where $f(n)$ is the time complexity of an algorithm and $g(n)$ is a reference function, means that $C_1 g(n) \leq f(n) \leq C_2 g(n)$ when $n > k$, where C_1, C_2, and k are constants. So without knowing the constants C_1, C_2, and k in the inequality, this estimate cannot be used to determine a lower bound and an upper bound on the number of operations used in the worst case. As remarked before, the time required for an operation depends on the type of operation and the

Links



Courtesy of Dr. Stephen Cook

STEPHEN COOK (BORN 1939) Stephen Cook was born in Buffalo, where his father worked as an industrial chemist and taught university courses. His mother taught English courses in a community college. While in high school Cook developed an interest in electronics through his work with a famous local inventor noted for inventing the first implantable cardiac pacemaker.

Cook was a mathematics major at the University of Michigan, graduating in 1961. He did graduate work at Harvard, receiving a master's degree in 1962 and a Ph.D. in 1966. Cook was appointed an assistant professor in the Mathematics Department at the University of California, Berkeley, in 1966. He was not granted tenure there, possibly because the members of the Mathematics Department did not find his work on what is now considered to be one of the most important areas of theoretical computer science of sufficient interest. In 1970, he joined the University of Toronto as an assistant professor, holding a joint appointment in the Computer Science Department and the Mathematics Department. He has remained at the University of Toronto, where he was appointed a University Professor in 1985.

Cook is considered to be one of the founders of computational complexity theory. His 1971 paper "The Complexity of Theorem Proving Procedures" formalized the notions of NP-completeness and polynomial-time reduction, showed that NP-complete problems exist by showing that the satisfiability problem is such a problem, and introduced the notorious P versus NP problem.

Cook has received many awards, including the 1982 Turing Award. He is married and has two sons. Among his interests are playing the violin and racing sailboats.

TABLE 2 The Computer Time Used by Algorithms.

Problem Size	*Bit Operations Used*					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*

computer being used. Often, instead of a big-Θ estimate on the worst-case time complexity of an algorithm, we have only a big-O estimate. Note that a big-O estimate on the time complexity of an algorithm provides an upper, but not a lower, bound on the worst-case time required for the algorithm as a function of the input size. Nevertheless, for simplicity, we will often use big-O estimates when describing the time complexity of algorithms, with the understanding that big-Θ estimates would provide more information.

Table 2 displays the time needed to solve problems of various sizes with an algorithm using the indicated number n of bit operations, assuming that each bit operation takes 10^{-11} seconds, a reasonable estimate of the time required for a bit operation using the fastest computers available in 2018. Times of more than 10^{100} years are indicated with an asterisk. In the future, these times will decrease as faster computers are developed. We can use the times shown in Table 2 to see whether it is reasonable to expect a solution to a problem of a specified size using an algorithm with known worst-case time complexity when we run this algorithm on a modern computer. Note that we cannot determine the exact time a computer uses to solve a problem with input of a particular size because of a myriad of issues involving computer hardware and the particular software implementation of the algorithm.

It is important to have a reasonable estimate for how long it will take a computer to solve a problem. For instance, if an algorithm requires approximately 10 hours, it may be worthwhile to spend the computer time (and money) required to solve this problem. But, if an algorithm requires approximately 10 billion years to solve a problem, it would be unreasonable to use resources to implement this algorithm. One of the most interesting phenomena of modern technology is the tremendous increase in the speed and memory space of computers. Another important factor that decreases the time needed to solve problems on computers is **parallel processing**, which is the technique of performing sequences of operations simultaneously.

Efficient algorithms, including most algorithms with polynomial time complexity, benefit most from significant technology improvements. However, these technology improvements offer little help in overcoming the complexity of algorithms of exponential or factorial time complexity. Because of the increased speed of computation, increases in computer memory, and the use of algorithms that take advantage of parallel processing, many problems that were considered impossible to solve five years ago are now routinely solved, and certainly five years from now this statement will still be true. This is even true when the algorithms used are intractable.

Exercises

1. Give a big-O estimate for the number of operations (where an operation is an addition or a multiplication) used in this segment of an algorithm.

```
t := 0
for i := 1 to 3
  for j := 1 to 4
    t := t + ij
```

2. Give a big-O estimate for the number additions used in this segment of an algorithm.

```
t := 0
for i := 1 to n
  for j := 1 to n
    t := t + i + j
```

3. Give a big-O estimate for the number of operations, where an operation is a comparison or a multiplication, used in this segment of an algorithm (ignoring comparisons used to test the conditions in the **for** loops, where $a_1, a_2, ..., a_n$ are positive real numbers).

```
m := 0
for i := 1 to n
  for j := i + 1 to n
    m := max(a_i a_j, m)
```

4. Give a big-O estimate for the number of operations, where an operation is an addition or a multiplication, used in this segment of an algorithm (ignoring comparisons used to test the conditions in the **while** loop).

```
i := 1
t := 0
while i ≤ n
    t := t + i
    i := 2i
```

5. How many comparisons are used by the algorithm given in Exercise 16 of Section 3.1 to find the smallest natural number in a sequence of n natural numbers?

6. a) Use pseudocode to describe the algorithm that puts the first four terms of a list of real numbers of arbitrary length in increasing order using the insertion sort.

b) Show that this algorithm has time complexity $O(1)$ in terms of the number of comparisons used.

7. Suppose that an element is known to be among the first four elements in a list of 32 elements. Would a linear search or a binary search locate this element more rapidly?

8. Given a real number x and a positive integer k, determine the number of multiplications used to find x^{2^k} starting with x and successively squaring (to find x^2, x^4, and so on). Is this a more efficient way to find x^{2^k} than by multiplying x by itself the appropriate number of times?

9. Give a big-O estimate for the number of comparisons used by the algorithm that determines the number of 1s in a bit string by examining each bit of the string to determine whether it is a 1 bit (see Exercise 25 of Section 3.1).

***10. a)** Show that this algorithm determines the number of 1 bits in the bit string S:

```
procedure bit count(S: bit string)
count := 0
while S ≠ 0
  count := count + 1
  S := S ∧ (S − 1)
return count {count is the number of 1s in S}
```

Here $S - 1$ is the bit string obtained by changing the rightmost 1 bit of S to a 0 and all the 0 bits to the right of this to 1s. [Recall that $S \wedge (S - 1)$ is the bitwise *AND* of S and $S - 1$.]

b) How many bitwise *AND* operations are needed to find the number of 1 bits in a string S using the algorithm in part (a)?

11. a) Suppose we have n subsets $S_1, S_2, \ldots, S_n$ of the set $\{1, 2, \ldots, n\}$. Express a brute-force algorithm that determines whether there is a disjoint pair of these subsets. [*Hint:* The algorithm should loop through the subsets; for each subset S_i, it should then loop through all other subsets; and for each of these other subsets S_j, it should loop through all elements k in S_i to determine whether k also belongs to S_j.]

b) Give a big-O estimate for the number of times the algorithm needs to determine whether an integer is in one of the subsets.

12. Consider the following algorithm, which takes as input a sequence of n integers $a_1, a_2, \ldots, a_n$ and produces as output a matrix $\mathbf{M} = \{m_{ij}\}$ where m_{ij} is the minimum term in the sequence of integers $a_i, a_{i+1}, \ldots, a_j$ for $j \geq i$ and $m_{ij} = 0$ otherwise.

```
initialize M so that m_ij = a_i if j ≥ i and m_ij = 0
  otherwise
for i := 1 to n
  for j := i + 1 to n
    for k := i + 1 to j
      m_ij := min(m_ij, a_k)
return M= {m_ij} {m_ij is the minimum term of
  a_i, a_{i+1}, ..., a_j}
```

a) Show that this algorithm uses $O(n^3)$ comparisons to compute the matrix $\mathbf{M}$.

b) Show that this algorithm uses $\Omega(n^3)$ comparisons to compute the matrix $\mathbf{M}$. Using this fact and part (a), conclude that the algorithms uses $\Theta(n^3)$ comparisons. [*Hint:* Only consider the cases where $i \leq n/4$ and $j \geq 3n/4$ in the two outer loops in the algorithm.]

13. The conventional algorithm for evaluating a polynomial $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ at $x = c$ can be expressed in pseudocode by

```
procedure polynomial(c, a_0, a_1, ..., a_n: real numbers)
  power := 1
  y := a_0
  for i := 1 to n
    power := power * c
    y := y + a_i * power
  return y {y = a_n c^n + a_{n-1} c^{n-1} + ··· + a_1 c + a_0}
```

where the final value of y is the value of the polynomial at $x = c$.

a) Evaluate $3x^2 + x + 1$ at $x = 2$ by working through each step of the algorithm showing the values assigned at each assignment step.

b) Exactly how many multiplications and additions are used to evaluate a polynomial of degree n at $x = c$? (Do not count additions used to increment the loop variable.)

14. There is a more efficient algorithm (in terms of the number of multiplications and additions used) for evaluating polynomials than the conventional algorithm described in the previous exercise. It is called **Horner's method**. This pseudocode shows how to use this method

to find the value of $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ at $x = c$.

procedure *Horner*$(c, a_0, a_1, a_2, \ldots, a_n$: real numbers)
  $y := a_n$
  for $i := 1$ **to** n
    $y := y * c + a_{n-i}$
  return y $\{y = a_n c^n + a_{n-1}c^{n-1} + \cdots + a_1 c + a_0\}$

a) Evaluate $3x^2 + x + 1$ at $x = 2$ by working through each step of the algorithm showing the values assigned at each assignment step.

b) Exactly how many multiplications and additions are used by this algorithm to evaluate a polynomial of degree n at $x = c$? (Do not count additions used to increment the loop variable.)

15. What is the largest n for which one can solve within one second a problem using an algorithm that requires $f(n)$ bit operations, where each bit operation is carried out in 10^{-9} seconds, with these functions $f(n)$?

a) $\log n$ **b)** n **c)** $n \log n$
d) n^2 **e)** 2^n **f)** $n!$

16. What is the largest n for which one can solve within a day using an algorithm that requires $f(n)$ bit operations, where each bit operation is carried out in 10^{-11} seconds, with these functions $f(n)$?

a) $\log n$ **b)** $1000n$ **c)** n^2
d) $1000n^2$ **e)** n^3 **f)** 2^n
g) 2^{2n} **h)** 2^{2^n}

17. What is the largest n for which one can solve within a minute using an algorithm that requires $f(n)$ bit operations, where each bit operation is carried out in 10^{-12} seconds, with these functions $f(n)$?

a) $\log \log n$ **b)** $\log n$ **c)** $(\log n)^2$
d) $1{,}000{,}000n$ **e)** n^2 **f)** 2^n
g) 2^{n^2}

18. How much time does an algorithm take to solve a problem of size n if this algorithm uses $2n^2 + 2^n$ operations, each requiring 10^{-9} seconds, with these values of n?

a) 10 **b)** 20 **c)** 50 **d)** 100

19. How much time does an algorithm using 2^{50} operations need if each operation takes these amounts of time?

a) 10^{-6} s **b)** 10^{-9} s **c)** 10^{-12} s

20. What is the effect in the time required to solve a problem when you double the size of the input from n to $2n$, assuming that the number of milliseconds the algorithm uses to solve the problem with input size n is each of these functions? [Express your answer in the simplest form possible, either as a ratio or a difference. Your answer may be a function of n or a constant.]

a) $\log \log n$ **b)** $\log n$ **c)** $100n$
d) $n \log n$ **e)** n^2 **f)** n^3
g) 2^n

21. What is the effect in the time required to solve a problem when you increase the size of the input from n to $n + 1$, assuming that the number of milliseconds the algorithm uses to solve the problem with input size n is each of these functions? [Express your answer in the simplest form possible, either as a ratio or a difference. Your answer may be a function of n or a constant.]

a) $\log n$ **b)** $100n$ **c)** n^2
d) n^3 **e)** 2^n **f)** 2^{n^2}
g) $n!$

22. Determine the least number of comparisons, or best-case performance,

a) required to find the maximum of a sequence of n integers, using Algorithm 1 of Section 3.1.

b) used to locate an element in a list of n terms with a linear search.

c) used to locate an element in a list of n terms using a binary search.

23. Analyze the average-case performance of the linear search algorithm, if exactly half the time the element x is not in the list, and if x is in the list, it is equally likely to be in any position.

24. An algorithm is called **optimal** for the solution of a problem with respect to a specified operation if there is no algorithm for solving this problem using fewer operations.

a) Show that Algorithm 1 in Section 3.1 is an optimal algorithm with respect to the number of comparisons of integers. [*Note:* Comparisons used for bookkeeping in the loop are not of concern here.]

b) Is the linear search algorithm optimal with respect to the number of comparisons of integers (not including comparisons used for bookkeeping in the loop)?

25. Describe the worst-case time complexity, measured in terms of comparisons, of the ternary search algorithm described in Exercise 27 of Section 3.1.

26. Describe the worst-case time complexity, measured in terms of comparisons, of the search algorithm described in Exercise 28 of Section 3.1.

27. Analyze the worst-case time complexity of the algorithm you devised in Exercise 29 of Section 3.1 for locating a mode in a list of nondecreasing integers.

28. Analyze the worst-case time complexity of the algorithm you devised in Exercise 30 of Section 3.1 for locating all modes in a list of nondecreasing integers.

29. Analyze the worst-case time complexity of the algorithm you devised in Exercise 33 of Section 3.1 for finding the first term of a sequence of integers equal to some previous term.

30. Analyze the worst-case time complexity of the algorithm you devised in Exercise 34 of Section 3.1 for finding all terms of a sequence that are greater than the sum of all previous terms.

31. Analyze the worst-case time complexity of the algorithm you devised in Exercise 35 of Section 3.1 for finding the first term of a sequence less than the immediately preceding term.

32. Determine the worst-case complexity in terms of comparisons of the algorithm from Exercise 5 in Section 3.1 for determining all values that occur more than once in a sorted list of integers.

33. Determine the worst-case complexity in terms of comparisons of the algorithm from Exercise 9 in Section 3.1 for determining whether a string of n characters is a palindrome.

34. How many comparisons does the selection sort (see preamble to Exercise 43 in Section 3.1) use to sort n items? Use your answer to give a big-O estimate of the complexity of the selection sort in terms of number of comparisons for the selection sort.

35. Determine a big-O estimate for the worst-case complexity in terms of number of comparisons used and the number of terms swapped by the binary insertion sort described in the preamble to Exercise 49 in Section 3.1.

36. Determine the number of character comparisons used by the naive string matcher to look for a pattern of m characters in a text with n characters if the first character of the pattern does not occur in the text.

37. Determine a big-O estimate of the number of character comparisons used by the naive string matcher to find all occurrences of a pattern of m characters in a text with n characters, in terms of the parameters m and n.

38. Determine big-O estimates for the algorithms for deciding whether two strings are anagrams from parts (a) and (b) of Exercise 31 of Section 3.1.

39. Determine big-O estimates for the algorithms for finding the closest of n real numbers from parts (a) and (b) of Exercise 32 of Section 3.1.

40. Show that the greedy algorithm for making change for n cents using quarters, dimes, nickels, and pennies has $O(n)$ complexity measured in terms of comparisons needed.

Exercises 41 and 42 deal with the problem of scheduling the most talks possible given the start and end times of n talks.

41. Find the complexity of a brute-force algorithm for scheduling the talks by examining all possible subsets of the talks. [*Hint:* Use the fact that a set with n elements has 2^n subsets.]

42. Find the complexity of the greedy algorithm for scheduling the most talks by adding at each step the talk with the earliest end time compatible with those already scheduled (Algorithm 7 in Section 3.1). Assume that the talks are not already sorted by earliest end time and assume that the worst-case time complexity of sorting is $O(n \log n)$.

43. Describe how the number of comparisons used in the worst case changes when these algorithms are used to search for an element of a list when the size of the list doubles from n to $2n$, where n is a positive integer.

a) linear search **b)** binary search

44. Describe how the number of comparisons used in the worst case changes when the size of the list to be sorted doubles from n to $2n$, where n is a positive integer when these sorting algorithms are used.

a) bubble sort **b)** insertion sort
c) selection sort (described in the preamble to Exercise 43 in Section 3.1)
d) binary insertion sort (described in the preamble to Exercise 49 in Section 3.1)

An $n \times n$ matrix is called **upper triangular** if $a_{ij} = 0$ whenever $i > j$.

45. From the definition of the matrix product, describe an algorithm in English for computing the product of two upper triangular matrices that ignores those products in the computation that are automatically equal to zero.

46. Give a pseudocode description of the algorithm in Exercise 45 for multiplying two upper triangular matrices.

47. How many multiplications of entries are used by the algorithm found in Exercise 45 for multiplying two $n \times n$ upper triangular matrices?

In Exercises 48–49 assume that the number of multiplications of entries used to multiply a $p \times q$ matrix and a $q \times r$ matrix is pqr.

48. What is the best order to form the product **ABC** if **A**, **B**, and **C** are matrices with dimensions 3×9, 9×4, and 4×2, respectively?

49. What is the best order to form the product **ABCD** if **A**, **B**, **C**, and **D** are matrices with dimensions 30×10, 10×40, 40×50, and 50×30, respectively?

Key Terms and Results

TERMS

algorithm: a finite sequence of precise instructions for performing a computation or solving a problem

searching algorithm: the problem of locating an element in a list

linear search algorithm: a procedure for searching a list element by element

binary search algorithm: a procedure for searching an ordered list by successively splitting the list in half

sorting: the reordering of the elements of a list into prescribed order

string searching: given a string, determining all the occurrences where this string occurs within a longer string

$f(x)$ is $O(g(x))$: the fact that $|f(x)| \leq C|g(x)|$ for all $x > k$ for some constants C and k

witness to the relationship $f(x)$ is $O(g(x))$: a pair C and k such that $|f(x)| \leq C|g(x)|$ whenever $x > k$

$f(x)$ is $\Omega(g(x))$: the fact that $|f(x)| \geq C|g(x)|$ for all $x > k$ for some positive constants C and k

$f(x)$ is $\Theta(g(x))$: the fact that $f(x)$ is both $O(g(x))$ and $\Omega(g(x))$

time complexity: the amount of time required for an algorithm to solve a problem

space complexity: the amount of space in computer memory required for an algorithm to solve a problem

worst-case time complexity: the greatest amount of time required for an algorithm to solve a problem of a given size

average-case time complexity: the average amount of time required for an algorithm to solve a problem of a given size

algorithmic paradigm: a general approach for constructing algorithms based on a particular concept

brute force: the algorithmic paradigm based on constructing algorithms for solving problems in a naive manner from the statement of the problem and definitions

greedy algorithm: an algorithm that makes the best choice at each step according to some specified condition

tractable problem: a problem for which there is a worst-case polynomial-time algorithm that solves it

intractable problem: a problem for which no worst-case polynomial-time algorithm exists for solving it

solvable problem: a problem that can be solved by an algorithm

unsolvable problem: a problem that cannot be solved by an algorithm

RESULTS

linear and binary search algorithms: (given in Section 3.1)

bubble sort: a sorting that uses passes where successive items are interchanged if they are in the wrong order

insertion sort: a sorting that at the jth step inserts the jth element into the correct position in the list, when the first $j-1$ elements of the list are already sorted

The linear search has $O(n)$ worst case time complexity.

The binary search has $O(\log n)$ worst case time complexity.

The bubble and insertion sorts have $O(n^2)$ worst case time complexity.

$\log n!$ is $O(n \log n)$.

If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then $(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x)))$ and $(f_1 f_2)(x)$ is $O((g_1 g_2(x))$.

If $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$, then $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ is $\Theta(x^n)$, and hence $O(n)$ and $\Omega(n)$.

Review Questions

1. a) Define the term *algorithm*.

b) What are the different ways to describe algorithms?

c) What is the difference between an algorithm for solving a problem and a computer program that solves this problem?

2. a) Describe, using English, an algorithm for finding the largest integer in a list of n integers.

b) Express this algorithm in pseudocode.

c) How many comparisons does the algorithm use?

3. a) State the definition of the fact that $f(n)$ is $O(g(n))$, where $f(n)$ and $g(n)$ are functions from the set of positive integers to the set of real numbers.

b) Use the definition of the fact that $f(n)$ is $O(g(n))$ directly to prove or disprove that $n^2 + 18n + 107$ is $O(n^3)$.

c) Use the definition of the fact that $f(n)$ is $O(g(n))$ directly to prove or disprove that n^3 is $O(n^2 + 18n + 107)$.

4. List these functions so that each function is big-O of the next function in the list: $(\log n)^3$, $n^3/1{,}000{,}000$, $\sqrt{n}$, $100n + 101$, 3^n, $n!$, $2^n n^2$.

5. a) How can you produce a big-O estimate for a function that is the sum of different terms where each term is the product of several functions?

b) Give a big-O estimate for the function $f(n) = (n! + 1)(2^n + 1) + (n^{n-2} + 8n^{n-3})(n^3 + 2^n)$. For the function g in your estimate $f(x)$ is $O(g(x))$, use a simple function of smallest possible order.

6. a) Define what the worst-case time complexity, average-case time complexity, and best-case time complexity (in terms of comparisons) mean for an algorithm that finds the smallest integer in a list of n integers.

b) What are the worst-case, average-case, and best-case time complexities, in terms of comparisons, of the algorithm that finds the smallest integer in a list of n integers by comparing each of the integers with the smallest integer found so far?

7. a) Describe the linear search and binary search algorithm for finding an integer in a list of integers in increasing order.

b) Compare the worst-case time complexities of these two algorithms.

c) Is one of these algorithms always faster than the other (measured in terms of comparisons)?

8. a) Describe the bubble sort algorithm.

b) Use the bubble sort algorithm to sort the list 5, 2, 4, 1, 3.

c) Give a big-O estimate for the number of comparisons used by the bubble sort.

9. a) Describe the insertion sort algorithm.

b) Use the insertion sort algorithm to sort the list 2, 5, 1, 4, 3.

c) Give a big-O estimate for the number of comparisons used by the insertion sort.

10. a) Explain the concept of a greedy algorithm.

b) Provide an example of a greedy algorithm that produces an optimal solution and explain why it produces an optimal solution.

c) Provide an example of a greedy algorithm that does not always produce an optimal solution and explain why it fails to do so.

11. Define what it means for a problem to be tractable and what it means for a problem to be solvable.

Supplementary Exercises

1. a) Describe an algorithm for locating the last occurrence of the largest number in a list of integers.
b) Estimate the number of comparisons used.

2. a) Describe an algorithm for finding the first and second largest elements in a list of integers.
b) Estimate the number of comparisons used.

3. a) Give an algorithm to determine whether a bit string contains a pair of consecutive zeros.
b) How many comparisons does the algorithm use?

4. a) Suppose that a list contains integers that are in order of largest to smallest and an integer can appear repeatedly in this list. Devise an algorithm that locates all occurrences of an integer x in the list.
b) Estimate the number of comparisons used.

5. a) Adapt Algorithm 1 in Section 3.1 to find the maximum and the minimum of a sequence of n elements by employing a temporary maximum and a temporary minimum that is updated as each successive element is examined.
b) Describe the algorithm from part (a) in pseudocode.
c) How many comparisons of elements in the sequence are carried out by this algorithm? (Do not count comparisons used to determine whether the end of the sequence has been reached.)

6. a) Describe in detail (and in English) the steps of an algorithm that finds the maximum and minimum of a sequence of n elements by examining pairs of successive elements, keeping track of a temporary maximum and a temporary minimum. If n is odd, both the temporary maximum and temporary minimum should initially equal the first term, and if n is even, the temporary minimum and temporary maximum should be found by comparing the initial two elements. The temporary maximum and temporary minimum should be updated by comparing them with the maximum and minimum of the pair of elements being examined.
b) Express the algorithm described in part (a) in pseudocode.
c) How many comparisons of elements of the sequence are carried out by this algorithm? (Do not count comparisons used to determine whether the end of the sequence has been reached.) How does this compare to the number of comparisons used by the algorithm in Exercise 5?

***7.** Show that the worst-case complexity in terms of comparisons of an algorithm that finds the maximum and minimum of n elements is at least $\lceil 3n/2 \rceil - 2$.

8. Devise an efficient algorithm for finding the second largest element in a sequence of n elements and determine the worst-case complexity of your algorithm.

9. Devise an algorithm that finds all equal pairs of sums of two terms of a sequence of n numbers, and determine the worst-case complexity of your algorithm.

10. Devise an algorithm that finds the closest pair of integers in a sequence of n integers, and determine the worst-case complexity of your algorithm. [*Hint:* Sort the sequence. Use the fact that sorting can be done with worst-case time complexity $O(n \log n)$.]

The **shaker sort** (or **bidirectional bubble sort**) successively compares pairs of adjacent elements, exchanging them if they are out of order, and alternately passing through the list from the beginning to the end and then from the end to the beginning until no exchanges are needed.

11. Show the steps used by the shaker sort to sort the list 3, 5, 1, 4, 6, 2.

12. Express the shaker sort in pseudocode.

13. Show that the shaker sort has $O(n^2)$ complexity measured in terms of the number of comparisons it uses.

14. Explain why the shaker sort is efficient for sorting lists that are already in close to the correct order.

15. Show that $(n \log n + n^2)^3$ is $O(n^6)$.

16. Show that $8x^3 + 12x + 100 \log x$ is $O(x^3)$.

17. Give a big-O estimate for $(x^2 + x(\log x)^3) \cdot (2^x + x^3)$.

18. Find a big-O estimate for $\sum_{j=1}^{n} j(j+1)$.

***19.** Show that $n!$ is not $O(2^n)$.

***20.** Show that n^n is not $O(n!)$.

21. Find all pairs of functions of the same order in this list of functions: $n^2 + (\log n)^2$, $n^2 + n$, $n^2 + \log 2^n + 1$, $(n+1)^3 - (n-1)^3$, and $(n + \log n)^2$.

22. Find all pairs of functions of the same order in this list of functions $n^2 + 2^n$, $n^2 + 2^{100}$, $n^2 + 2^{2n}$, $n^2 + n!$, $n^2 + 3^n$, and $(n^2 + 1)^2$.

23. Find an integer n with $n > 2$ for which $n^{2^{100}} < 2^n$.

24. Find an integer n with $n > 2$ for which $(\log n)^{2^{100}} < \sqrt{n}$.

***25.** Arrange the functions n^n, $(\log n)^2$, $n^{1.0001}$, $(1.0001)^n$, $2^{\sqrt{\log_2 n}}$, and $n(\log n)^{1001}$ in a list so that each function is big-O of the next function. [*Hint:* To determine the relative size of some of these functions, take logarithms.]

***26.** Arrange the function 2^{100n}, 2^{n^2}, $2^{n!}$, 2^{2^n}, $n^{\log n}$, $n \log n \log \log n$, $n^{3/2}$, $n(\log n)^{3/2}$, and $n^{4/3}(\log n)^2$ in a list so that each function is big-O of the next function. [*Hint:* To determine the relative size of some of these functions, take logarithms.]

***27.** Give an example of two increasing functions $f(n)$ and $g(n)$ from the set of positive integers to the set of positive integers such that neither $f(n)$ is $O(g(n))$ nor $g(n)$ is $O(f(n))$.

28. Show that if the denominations of coins are $c^0, c^1, \ldots, c^k$, where k is a positive integer and c is a positive integer, $c > 1$, the greedy algorithm always produces change using the fewest coins possible.

29. a) Use pseudocode to specify a brute-force algorithm that determines when given as input a sequence of n positive integers whether there are two distinct terms of the sequence that have as sum a third term. The algorithm should loop through all triples of terms of the sequence, checking whether the sum of the first two terms equals the third.

b) Give a big-O estimate for the complexity of the brute-force algorithm from part (a).

30. a) Devise a more efficient algorithm for solving the problem described in Exercise 29 that first sorts the input sequence and then checks for each pair of terms whether their difference is in the sequence.

b) Give a big-O estimate for the complexity of this algorithm. Is it more efficient than the brute-force algorithm from Exercise 29?

Suppose we have s men and s women each with their preference lists for the members of the opposite gender, as described in the preamble to Exercise 64 in Section 3.1. We say that a woman w is a **valid partner** for a man m if there is some stable matching in which they are paired. Similarly, a man m is a **valid partner** for a woman w if there is some stable matching in which they are paired. A matching in which each man is assigned his valid partner ranking highest on his preference list is called **male optimal**, and a matching in which each woman is assigned her valid partner ranking lowest on her preference list is called **female pessimal**.

31. Find all valid partners for each man and each woman if there are three men m_1, m_2, and m_3 and three women w_1, w_2, w_3 with these preference rankings of the men for the women, from highest to lowest: m_1: w_3, w_1, w_2; m_2: w_3, w_2, w_1; m_3: w_2, w_3, w_1; and with these preference rankings of the women for the men, from highest to lowest: w_1: m_3, m_2, m_1; w_2: m_1, m_3, m_2; w_3: m_3, m_2, m_1.

***32.** Show that the deferred acceptance algorithm given in the preamble to Exercise 65 of Section 3.1, always produces a male optimal and female pessimal matching.

33. Define what it means for a matching to be female optimal and for a matching to be male pessimal.

***34.** Show that when woman do the proposing in the deferred acceptance algorithm, the matching produced is female optimal and male pessimal.

In Exercises 35 and 36 we consider variations on the problem of finding stable matchings of men and women described in the preamble to Exercise 65 in Section 3.1.

***35.** In this exercise we consider matching problems where there may be different numbers of men and women, so that it is impossible to match everyone with a member of the opposite gender.

a) Extend the definition of a stable matching from that given in the preamble to Exercise 64 in Section 3.1 to cover the case where there are unequal numbers of men and women. Avoid all cases where a man and a woman would prefer each other to their current situation, including those involving unmatched people. (Assume that an unmatched person prefers a match with a member of the opposite gender to remaining unmatched.)

b) Adapt the deferred acceptance algorithm to find stable matchings, using the definition of stable matchings from part (a), when there are different numbers of men and women.

c) Prove that all matchings produced by the algorithm from part (b) are stable, according to the definition from part (a).

***36.** In this exercise we consider matching problems where some man-woman pairs are not allowed.

a) Extend the definition of a stable matching to cover the situation where there are the same number of men and women, but certain pairs of men and women are forbidden. Avoid all cases where a man and a woman would prefer each other to their current situation, including those involving unmatched people.

b) Adapt the deferred acceptance algorithm to find stable matchings when there are the same number of men and women, but certain man-woman pairs are forbidden. Be sure to consider people who are unmatched at the end of the algorithm. (Assume that an unmatched person prefers a match with a member of the opposite gender who is not a forbidden partner to remaining unmatched.)

c) Prove that all matchings produced by the algorithm from (b) are stable, according to the definition in part (a).

Exercises 37–40 deal with the problem of scheduling n jobs on a single processor. To complete job j, the processor must run job j for time t_j without interruption. Each job has a deadline d_j. If we start job j at time s_j, it will be completed at time $e_j = s_j + t_j$. The **lateness** of the job measures how long it finishes after its deadline, that is, the lateness of job j is $\max(0, e_j - d_j)$. We wish to devise a greedy algorithm that minimizes the maximum lateness of a job among the n jobs.

37. Suppose we have five jobs with specified required times and deadlines: $t_1 = 25$, $d_1 = 50$; $t_2 = 15$, $d_2 = 60$; $t_3 = 20$, $d_3 = 60$; $t_4 = 5$, $d_4 = 55$; $t_5 = 10$, $d_5 = 75$. Find the maximum lateness of any job when the jobs are scheduled in this order (and they start at time 0): Job 3, Job 1, Job 4, Job 2, Job 5. Answer the same question for the schedule Job 5, Job 4, Job 3, Job 1, Job 2.

38. The **slackness** of a job requiring time t and with deadline d is $d - t$, the difference between its deadline and the time it requires. Find an example that shows that scheduling jobs by increasing slackness does not always yield a schedule with the smallest possible maximum lateness.

39. Find an example that shows that scheduling jobs in order of increasing time required does not always yield a schedule with the smallest possible maximum lateness.

*40. Prove that scheduling jobs in order of increasing deadlines always produces a schedule that minimizes the maximum lateness of a job. [*Hint:* First show that for a schedule to be optimal, jobs must be scheduled with no idle time between them and so that no job is scheduled before another with an earlier deadline.]

41. Suppose that we have a knapsack with total capacity of W kg. We also have n items where item j has mass w_j. The **knapsack problem** asks for a subset of these n items with the largest possible total mass not exceeding W.
   a) Devise a brute-force algorithm for solving the knapsack problem.
   b) Solve the knapsack problem when the capacity of the knapsack is 18 kg and there are five items: a 5-kg sleeping bag, an 8-kg tent, a 7-kg food pack, a 4-kg container of water, and an 11-kg portable stove.

In Exercises 42–46 we will study the problem of load balancing. The input to the problem is a collection of p processors and n jobs, t_j is the time required to run job j, jobs run without interruption on a single machine until finished, and a processor can run only one job at a time. The **load** L_k of processor k is the sum over all jobs assigned to processor k of the times required to run these jobs. The **makespan** is the maximum load over all the p processors. The load balancing problem asks for an assignment of jobs to processors to minimize the makespan.

42. Suppose we have three processors and five jobs requiring times $t_1 = 3$, $t_2 = 5$, $t_3 = 4$, $t_4 = 7$, and $t_5 = 8$. Solve the load balancing problem for this input by finding the assignment of the five jobs to the three processors that minimizes the makespan.

43. Suppose that L^* is the minimum makespan when p processors are given n jobs, where t_j is the time required to run job j.
   a) Show that $L^* \geq \max_{j=1,2,\ldots,n} t_j$.
   b) Show that $L^* \geq \frac{1}{p} \sum_{j=1}^{n} t_j$.

44. Write out in pseudocode the greedy algorithm that goes through the jobs in order and assigns each job to the processor with the smallest load at that point in the algorithm.

45. Run the algorithm from Exercise 44 on the input given in Exercise 42.

An **approximation algorithm** for an optimization problem produces a solution guaranteed to be close to an optimal solution. More precisely, suppose that the optimization problem asks for an input S that minimizes $F(X)$ where F is some function of the input X. If an algorithm always finds an input T with $F(T) \leq cF(S)$, where c is a fixed positive real number, the algorithm is called a ***c*-approximation algorithm** for the problem.

*46. Prove that the algorithm from Exercise 44 is a 2-approximation algorithm for the load balancing problem. [*Hint:* Use both parts of Exercise 43.]

Computer Projects

Write programs with these inputs and outputs.

1. Given a list of n integers, find the largest integer in the list.
2. Given a list of n integers, find the first and last occurrences of the largest integer in the list.
3. Given a list of n distinct integers, determine the position of an integer in the list using a linear search.
4. Given an ordered list of n distinct integers, determine the position of an integer in the list using a binary search.
5. Given a list of n integers, sort them using a bubble sort.
6. Given a list of n integers, sort them using an insertion sort.
7. Given two strings of characters use the naive string matching algorithm to determine whether the shorter string occurs in the longer string.
8. Given an integer n, use the cashier's algorithm to find the change for n cents using quarters, dimes, nickels, and pennies.
9. Given the starting and ending times of n talks, use the appropriate greedy algorithm to schedule the most talks possible in a single lecture hall.
10. Given an ordered list of n integers and an integer x in the list, find the number of comparisons used to determine the position of x in the list using a linear search and using a binary search.
11. Given a list of integers, determine the number of comparisons used by the bubble sort and by the insertion sort to sort this list.

Computations and Explorations

Use a computational program or programs you have written to do these exercises.

1. We know that n^b is $O(d^n)$ when b and d are positive numbers with $d \geq 2$. Give values of the constants C and k such that $n^b \leq Cd^n$ whenever $x > k$ for each of these sets of values: $b = 10$, $d = 2$; $b = 20$, $d = 3$; $b = 1000$, $d = 7$.

2. Compute the change for different values of n with coins of different denominations using the cashier's algorithm and determine whether the smallest number of coins was used. Can you find conditions so that the cashier's algorithm is guaranteed to use the fewest coins possible?
3. Using a generator of random orderings of the integers $1, 2, \ldots, n$, find the number of comparisons used by the bubble sort, insertion sort, binary insertion sort, and selection sort to sort these integers.
4. Collect experimental evidence comparing the number of comparisons used by the sorting algorithms in Question 3 when used to sort sequences that only have a small fraction of terms out of order.

*5. Write a program that animates the progress of all the sorting algorithms in Question 3 when given the numbers from 1 to 100 in random order.

Writing Projects

Respond to these with essays using outside sources.

1. Examine the history of the word *algorithm* and describe the use of this word in early writings.
2. Look up Bachmann's original introduction of big-O notation. Explain how he and others have used this notation.
3. Explain how sorting algorithms can be classified into a taxonomy based on the underlying principle on which they are based.
4. Describe the radix sort algorithm.
5. Describe some of the different algorithms for string matching.
6. Describe some of the different applications of string matching in bioinformatics.
7. Describe the historic trends in how quickly processors can perform operations and use these trends to estimate how quickly processors will be able to perform operations in the next 20 years.
8. Develop a detailed list of algorithmic paradigms and provide examples using each of these paradigms.
9. Explain what the Turing Award is and describe the criteria used to select winners. List six past winners of the award and why they received the award.
10. Describe what is meant by a parallel algorithm. Explain how the pseudocode used in this book can be extended to handle parallel algorithms.
11. Explain how the complexity of parallel algorithms can be measured. Give some examples to illustrate this concept, showing how a parallel algorithm can work more quickly than one that does not operate in parallel.
12. Describe six different NP-complete problems.
13. Demonstrate how one of the many different NP-complete problems can be reduced to the satisfiability problem.

***57.** Dynamic programming can be used to develop an algorithm for solving the matrix-chain multiplication problem introduced in Section 3.3. This is the problem of determining how the product $\mathbf{A}_1\mathbf{A}_2 \cdots \mathbf{A}_n$ can be computed using the fewest integer multiplications, where $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n$ are $m_1 \times m_2, m_2 \times m_3, \ldots, m_n \times m_{n+1}$ matrices, respectively, and each matrix has integer entries. Recall that by the associative law, the product does not depend on the order in which the matrices are multiplied.

a) Show that the brute-force method of determining the minimum number of integer multiplications needed to solve a matrix-chain multiplication problem has exponential worst-case complexity. [*Hint:* Do this by first showing that the order of multiplication of matrices is specified by parenthesizing the product. Then, use Example 5 and the result of part (c) of Exercise 43 in Section 8.4.]

b) Denote by $\mathbf{A}_{ij}$ the product $\mathbf{A}_i\mathbf{A}_{i+1} \ldots, \mathbf{A}_j$, and $M(i, j)$ the minimum number of integer multiplications required to find $\mathbf{A}_{ij}$. Show that if the least number of integer multiplications are used to compute $\mathbf{A}_{ij}$, where $i < j$, by splitting the product into the product of $\mathbf{A}_i$ through $\mathbf{A}_k$ and the product of $\mathbf{A}_{k+1}$ through $\mathbf{A}_j$, then the first k terms must be parenthesized so that $\mathbf{A}_{ik}$ is computed in the optimal way using $M(i, k)$ integer multiplications, and $\mathbf{A}_{k+1,j}$ must be parenthesized so that $\mathbf{A}_{k+1,j}$ is computed in the optimal way using $M(k + 1, j)$ integer multiplications.

c) Explain why part (b) leads to the recurrence relation $M(i, j) = \min_{i \le k < j}(M(i, k) + M(k + 1, j) + m_i m_{k+1} m_{j+1})$ if $1 \le i \le j < j \le n$.

d) Use the recurrence relation in part (c) to construct an efficient algorithm for determining the order the n matrices should be multiplied to use the minimum number of integer multiplications. Store the partial results $M(i, j)$ as you find them so that your algorithm will not have exponential complexity.

e) Show that your algorithm from part (d) has $O(n^3)$ worst-case complexity in terms of multiplications of integers.

8.2 Solving Linear Recurrence Relations

8.2.1 Introduction

Links

A wide variety of recurrence relations occur in models. Some of these recurrence relations can be solved using iteration or some other ad hoc technique. However, one important class of recurrence relations can be explicitly solved in a systematic way. These are recurrence relations that express the terms of a sequence as linear combinations of previous terms.

Definition 1

A *linear homogeneous recurrence relation of degree k with constant coefficients* is a recurrence relation of the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k},$$

where $c_1, c_2, \ldots, c_k$ are real numbers, and $c_k \neq 0$.

The recurrence relation in the definition is **linear** because the right-hand side is a sum of previous terms of the sequence each multiplied by a function of n. The recurrence relation is **homogeneous** because no terms occur that are not multiples of the a_js. The coefficients of the terms of the sequence are all **constants**, rather than functions that depend on n. The **degree** is k because a_n is expressed in terms of the previous k terms of the sequence.

A consequence of the second principle of mathematical induction is that a sequence satisfying the recurrence relation in the definition is uniquely determined by this recurrence relation and the k initial conditions

$$a_0 = C_0, a_1 = C_1, \ldots, a_{k-1} = C_{k-1}.$$

EXAMPLE 1 The recurrence relation $P_n = (1.11)P_{n-1}$ is a linear homogeneous recurrence relation of degree one. The recurrence relation $f_n = f_{n-1} + f_{n-2}$ is a linear homogeneous recurrence relation of

degree two. The recurrence relation $a_n = a_{n-5}$ is a linear homogeneous recurrence relation of degree five. ◀

To help clarify the definition of linear homogeneous recurrence relations with constant coefficients, we will now provide examples of recurrence relations each lacking one of the defining properties.

EXAMPLE 2 The recurrence relation $a_n = a_{n-1} + a_{n-2}^2$ is not linear. The recurrence relation $H_n = 2H_{n-1} + 1$ is not homogeneous. The recurrence relation $B_n = nB_{n-1}$ does not have constant coefficients. ◀

Linear homogeneous recurrence relations are studied for two reasons. First, they often occur in modeling of problems. Second, they can be systematically solved.

8.2.2 Solving Linear Homogeneous Recurrence Relations with Constant Coefficients

Recurrence relations may be difficult to solve, but fortunately this is not the case for linear homogenous recurrence relations with constant coefficients. We can use two key ideas to find all their solutions. First, these recurrence relations have solutions of the form $a_n = r^n$, where r is a constant. To see this, observe that $a_n = r^n$ is a solution of the recurrence relation $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$ if and only if

$$r^n = c_1 r^{n-1} + c_2 r^{n-2} + \cdots + c_k r^{n-k}.$$

When both sides of this equation are divided by r^{n-k} (when $r \neq 0$) and the right-hand side is subtracted from the left, we obtain the equation

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - \cdots - c_{k-1} r - c_k = 0.$$

Consequently, the sequence $\{a_n\}$ with $a_n = r^n$ where $r \neq 0$ is a solution if and only if r is a solution of this last equation. We call this the **characteristic equation** of the recurrence relation. The solutions of this equation are called the **characteristic roots** of the recurrence relation. As we will see, these characteristic roots can be used to give an explicit formula for all the solutions of the recurrence relation.

The other key observation is that a linear combination of two solutions of a linear homogeneous recurrence relation is also a solution. To see this, suppose that s_n and t_n are both solutions of $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$. Then

$$s_n = c_1 s_{n-1} + c_2 s_{n-2} + \cdots + c_k s_{n-k}$$

and

$$t_n = c_1 t_{n-1} + c_2 t_{n-2} + \cdots + c_k t_{n-k}.$$

Now suppose that b_1 and b_2 are real numbers. Then

$$\begin{aligned} b_1 s_n + b_2 t_n &= b_1(c_1 s_{n-1} + c_2 s_{n-2} + \cdots + c_k s_{n-k}) + b_2(c_1 t_{n-1} + c_2 t_{n-2} + \cdots + c_k t_{n-k}) \\ &= c_1(b_1 s_{n-1} + b_2 t_{n-1}) + c_2(b_1 s_{n-2} + b_2 t_{n-2}) + \cdots + c_k(b_1 s_{n-k} + b_k t_{n-k}). \end{aligned}$$

This means that $b_1 s_n + b_2 t_n$ is also a solution of the same linear homogeneous recurrence relation.

Using these key observations, we will show how to solve linear homogeneous recurrence relations with constant coefficients.

THE DEGREE TWO CASE We now turn our attention to linear homogeneous recurrence relations of degree two. First, consider the case when there are two distinct characteristic roots.

THEOREM 1

Let c_1 and c_2 be real numbers. Suppose that $r^2 - c_1 r - c_2 = 0$ has two distinct roots r_1 and r_2. Then the sequence $\{a_n\}$ is a solution of the recurrence relation $a_n = c_1 a_{n-1} + c_2 a_{n-2}$ if and only if $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$ for $n = 0, 1, 2, \ldots$, where α_1 and α_2 are constants.

Proof: We must do two things to prove the theorem. First, it must be shown that if r_1 and r_2 are the roots of the characteristic equation, and α_1 and α_2 are constants, then the sequence $\{a_n\}$ with $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$ is a solution of the recurrence relation. Second, it must be shown that if the sequence $\{a_n\}$ is a solution, then $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$ for some constants α_1 and α_2.

We now show that if $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$, then the sequence $\{a_n\}$ is a solution of the recurrence relation. Because r_1 and r_2 are roots of $r^2 - c_1 r - c_2 = 0$, it follows that $r_1^2 = c_1 r_1 + c_2$ and $r_2^2 = c_1 r_2 + c_2$.

From these equations, we see that

$$\begin{aligned} c_1 a_{n-1} + c_2 a_{n-2} &= c_1(\alpha_1 r_1^{n-1} + \alpha_2 r_2^{n-1}) + c_2(\alpha_1 r_1^{n-2} + \alpha_2 r_2^{n-2}) \\ &= \alpha_1 r_1^{n-2}(c_1 r_1 + c_2) + \alpha_2 r_2^{n-2}(c_1 r_2 + c_2) \\ &= \alpha_1 r_1^{n-2} r_1^2 + \alpha_2 r_2^{n-2} r_2^2 \\ &= \alpha_1 r_1^n + \alpha_2 r_2^n \\ &= a_n. \end{aligned}$$

This shows that the sequence $\{a_n\}$ with $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$ is a solution of the recurrence relation.

To show that every solution $\{a_n\}$ of the recurrence relation $a_n = c_1 a_{n-1} + c_2 a_{n-2}$ has $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$ for $n = 0, 1, 2, \ldots$, for some constants α_1 and α_2, suppose that $\{a_n\}$ is a solution of the recurrence relation, and the initial conditions $a_0 = C_0$ and $a_1 = C_1$ hold. It will be shown that there are constants α_1 and α_2 such that the sequence $\{a_n\}$ with $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$ satisfies these same initial conditions. This requires that

$$\begin{aligned} a_0 &= C_0 = \alpha_1 + \alpha_2, \\ a_1 &= C_1 = \alpha_1 r_1 + \alpha_2 r_2. \end{aligned}$$

We can solve these two equations for α_1 and α_2. From the first equation it follows that $\alpha_2 = C_0 - \alpha_1$. Inserting this expression into the second equation gives

$$C_1 = \alpha_1 r_1 + (C_0 - \alpha_1) r_2.$$

Hence,

$$C_1 = \alpha_1 (r_1 - r_2) + C_0 r_2.$$

This shows that

$$\alpha_1 = \frac{C_1 - C_0 r_2}{r_1 - r_2}$$

and

$$\alpha_2 = C_0 - \alpha_1 = C_0 - \frac{C_1 - C_0 r_2}{r_1 - r_2} = \frac{C_0 r_1 - C_1}{r_1 - r_2},$$

where these expressions for α_1 and α_2 depend on the fact that $r_1 \neq r_2$. (When $r_1 = r_2$, this theorem is not true.) Hence, with these values for α_1 and α_2, the sequence $\{a_n\}$ with $\alpha_1 r_1^n + \alpha_2 r_2^n$ satisfies the two initial conditions.

We know that $\{a_n\}$ and $\{\alpha_1 r_1^n + \alpha_2 r_2^n\}$ are both solutions of the recurrence relation $a_n = c_1 a_{n-1} + c_2 a_{n-2}$ and both satisfy the initial conditions when $n = 0$ and $n = 1$. Because there is a unique solution of a linear homogeneous recurrence relation of degree two with two initial conditions, it follows that the two solutions are the same, that is, $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$ for all nonnegative integers n. We have completed the proof by showing that a solution of the linear homogeneous recurrence relation with constant coefficients of degree two must be of the form $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$, where α_1 and α_2 are constants. ◁

The characteristic roots of a linear homogeneous recurrence relation with constant coefficients may be complex numbers. Theorem 1 (and also subsequent theorems in this section) still applies in this case. Recurrence relations with complex characteristic roots will not be discussed in the text. Readers familiar with complex numbers may wish to solve Exercises 38 and 39.

Examples 3 and 4 show how to use Theorem 1 to solve recurrence relations.

EXAMPLE 3 What is the solution of the recurrence relation

Extra Examples

$$a_n = a_{n-1} + 2a_{n-2}$$

with $a_0 = 2$ and $a_1 = 7$?

Solution: Theorem 1 can be used to solve this problem. The characteristic equation of the recurrence relation is $r^2 - r - 2 = 0$. Its roots are $r = 2$ and $r = -1$. Hence, the sequence $\{a_n\}$ is a solution to the recurrence relation if and only if

$$a_n = \alpha_1 2^n + \alpha_2 (-1)^n,$$

for some constants α_1 and α_2. From the initial conditions, it follows that

$$a_0 = 2 = \alpha_1 + \alpha_2,$$
$$a_1 = 7 = \alpha_1 \cdot 2 + \alpha_2 \cdot (-1).$$

Solving these two equations shows that $\alpha_1 = 3$ and $\alpha_2 = -1$. Hence, the solution to the recurrence relation and initial conditions is the sequence $\{a_n\}$ with

$$a_n = 3 \cdot 2^n - (-1)^n.$$ ◀

EXAMPLE 4 Find an explicit formula for the Fibonacci numbers.

Solution: Recall that the sequence of Fibonacci numbers satisfies the recurrence relation $f_n = f_{n-1} + f_{n-2}$ and also satisfies the initial conditions $f_0 = 0$ and $f_1 = 1$. The roots of the characteristic equation $r^2 - r - 1 = 0$ are $r_1 = (1 + \sqrt{5})/2$ and $r_2 = (1 - \sqrt{5})/2$. Therefore, from Theorem 1 it follows that the Fibonacci numbers are given by

$$f_n = \alpha_1 \left(\frac{1 + \sqrt{5}}{2} \right)^n + \alpha_2 \left(\frac{1 - \sqrt{5}}{2} \right)^n,$$

for some constants α_1 and α_2. The initial conditions $f_0 = 0$ and $f_1 = 1$ can be used to find these constants. We have

$$f_0 = \alpha_1 + \alpha_2 = 0,$$
$$f_1 = \alpha_1\left(\frac{1+\sqrt{5}}{2}\right) + \alpha_2\left(\frac{1-\sqrt{5}}{2}\right) = 1.$$

The solution to these simultaneous equations for α_1 and α_2 is

$$\alpha_1 = 1/\sqrt{5}, \qquad \alpha_2 = -1/\sqrt{5}.$$

Consequently, the Fibonacci numbers are given by

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

◀

Theorem 1 does not apply when there is one characteristic root of multiplicity two. If this happens, then $a_n = nr_0^n$ is another solution of the recurrence relation when r_0 is a root of multiplicity two of the characteristic equation. Theorem 2 shows how to handle this case.

THEOREM 2

Let c_1 and c_2 be real numbers with $c_2 \neq 0$. Suppose that $r^2 - c_1 r - c_2 = 0$ has only one root r_0. A sequence $\{a_n\}$ is a solution of the recurrence relation $a_n = c_1 a_{n-1} + c_2 a_{n-2}$ if and only if $a_n = \alpha_1 r_0^n + \alpha_2 n r_0^n$, for $n = 0, 1, 2, \ldots$, where α_1 and α_2 are constants.

The proof of Theorem 2 is left as Exercise 10. Example 5 illustrates the use of this theorem.

EXAMPLE 5 What is the solution of the recurrence relation

$$a_n = 6a_{n-1} - 9a_{n-2}$$

with initial conditions $a_0 = 1$ and $a_1 = 6$?

Solution: The only root of $r^2 - 6r + 9 = 0$ is $r = 3$. Hence, the solution to this recurrence relation is

$$a_n = \alpha_1 3^n + \alpha_2 n 3^n$$

for some constants α_1 and α_2. Using the initial conditions, it follows that

$$a_0 = 1 = \alpha_1,$$
$$a_1 = 6 = \alpha_1 \cdot 3 + \alpha_2 \cdot 3.$$

Solving these two equations shows that $\alpha_1 = 1$ and $\alpha_2 = 1$. Consequently, the solution to this recurrence relation and the initial conditions is

$$a_n = 3^n + n3^n.$$

◀

THE GENERAL CASE We will now state the general result about the solution of linear homogeneous recurrence relations with constant coefficients, where the degree may be greater than two, under the assumption that the characteristic equation has distinct roots. The proof of this result will be left as Exercise 16.

THEOREM 3

Let $c_1, c_2, \ldots, c_k$ be real numbers. Suppose that the characteristic equation

$$r^k - c_1 r^{k-1} - \cdots - c_k = 0$$

has k distinct roots $r_1, r_2, \ldots, r_k$. Then a sequence $\{a_n\}$ is a solution of the recurrence relation

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$$

if and only if

$$a_n = \alpha_1 r_1^n + \alpha_2 r_2^n + \cdots + \alpha_k r_k^n$$

for $n = 0, 1, 2, \ldots$, where $\alpha_1, \alpha_2, \ldots, \alpha_k$ are constants.

We illustrate the use of the theorem with Example 6.

EXAMPLE 6 Find the solution to the recurrence relation

$$a_n = 6a_{n-1} - 11a_{n-2} + 6a_{n-3}$$

with the initial conditions $a_0 = 2$, $a_1 = 5$, and $a_2 = 15$.

Solution: The characteristic polynomial of this recurrence relation is

$$r^3 - 6r^2 + 11r - 6.$$

The characteristic roots are $r = 1$, $r = 2$, and $r = 3$, because $r^3 - 6r^2 + 11r - 6 = (r - 1)(r - 2)(r - 3)$. Hence, the solutions to this recurrence relation are of the form

$$a_n = \alpha_1 \cdot 1^n + \alpha_2 \cdot 2^n + \alpha_3 \cdot 3^n.$$

To find the constants α_1, α_2, and α_3, use the initial conditions. This gives

$$\begin{aligned} a_0 &= 2 = \alpha_1 + \alpha_2 + \alpha_3, \\ a_1 &= 5 = \alpha_1 + \alpha_2 \cdot 2 + \alpha_3 \cdot 3, \\ a_2 &= 15 = \alpha_1 + \alpha_2 \cdot 4 + \alpha_3 \cdot 9. \end{aligned}$$

When these three simultaneous equations are solved for α_1, α_2, and α_3, we find that $\alpha_1 = 1$, $\alpha_2 = -1$, and $\alpha_3 = 2$. Hence, the unique solution to this recurrence relation and the given initial conditions is the sequence $\{a_n\}$ with

$$a_n = 1 - 2^n + 2 \cdot 3^n.$$ ◀

We now state the most general result about linear homogeneous recurrence relations with constant coefficients, allowing the characteristic equation to have multiple roots. The key point is that for each root r of the characteristic equation, the general solution has a summand of the

form $P(n)r^n$, where $P(n)$ is a polynomial of degree $m - 1$, with m the multiplicity of this root. We leave the proof of this result as Exercise 51.

THEOREM 4

Let $c_1, c_2, \ldots, c_k$ be real numbers. Suppose that the characteristic equation

$$r^k - c_1 r^{k-1} - \cdots - c_k = 0$$

has t distinct roots $r_1, r_2, \ldots, r_t$ with multiplicities $m_1, m_2, \ldots, m_t$, respectively, so that $m_i \geq 1$ for $i = 1, 2, \ldots, t$ and $m_1 + m_2 + \cdots + m_t = k$. Then a sequence $\{a_n\}$ is a solution of the recurrence relation

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$$

if and only if

$$\begin{aligned} a_n = {} & (\alpha_{1,0} + \alpha_{1,1} n + \cdots + \alpha_{1,m_1-1} n^{m_1-1}) r_1^n \\ & + (\alpha_{2,0} + \alpha_{2,1} n + \cdots + \alpha_{2,m_2-1} n^{m_2-1}) r_2^n \\ & + \cdots + (\alpha_{t,0} + \alpha_{t,1} n + \cdots + \alpha_{t,m_t-1} n^{m_t-1}) r_t^n \end{aligned}$$

for $n = 0, 1, 2, \ldots$, where $\alpha_{i,j}$ are constants for $1 \leq i \leq t$ and $0 \leq j \leq m_i - 1$.

Example 7 illustrates how Theorem 4 is used to find the general form of a solution of a linear homogeneous recurrence relation when the characteristic equation has several repeated roots.

EXAMPLE 7 Suppose that the roots of the characteristic equation of a linear homogeneous recurrence relation are 2, 2, 2, 5, 5, and 9 (that is, there are three roots, the root 2 with multiplicity three, the root 5 with multiplicity two, and the root 9 with multiplicity one). What is the form of the general solution?

Solution: By Theorem 4, the general form of the solution is

$$(\alpha_{1,0} + \alpha_{1,1} n + \alpha_{1,2} n^2) 2^n + (\alpha_{2,0} + \alpha_{2,1} n) 5^n + \alpha_{3,0} 9^n.$$

◀

We now illustrate the use of Theorem 4 to solve a linear homogeneous recurrence relation with constant coefficients when the characteristic equation has a root of multiplicity three.

EXAMPLE 8 Find the solution to the recurrence relation

$$a_n = -3a_{n-1} - 3a_{n-2} - a_{n-3}$$

with initial conditions $a_0 = 1$, $a_1 = -2$, and $a_2 = -1$.

Solution: The characteristic equation of this recurrence relation is

$$r^3 + 3r^2 + 3r + 1 = 0.$$

Because $r^3 + 3r^2 + 3r + 1 = (r + 1)^3$, there is a single root $r = -1$ of multiplicity three of the characteristic equation. By Theorem 4 the solutions of this recurrence relation are of the form

$$a_n = \alpha_{1,0}(-1)^n + \alpha_{1,1} n(-1)^n + \alpha_{1,2} n^2 (-1)^n.$$

To find the constants $\alpha_{1,0}$, $\alpha_{1,1}$, and $\alpha_{1,2}$, use the initial conditions. This gives

$$a_0 = 1 = \alpha_{1,0},$$
$$a_1 = -2 = -\alpha_{1,0} - \alpha_{1,1} - \alpha_{1,2},$$
$$a_2 = -1 = \alpha_{1,0} + 2\alpha_{1,1} + 4\alpha_{1,2}.$$

The simultaneous solution of these three equations is $\alpha_{1,0} = 1$, $\alpha_{1,1} = 3$, and $\alpha_{1,2} = -2$. Hence, the unique solution to this recurrence relation and the given initial conditions is the sequence $\{a_n\}$ with

$$a_n = (1 + 3n - 2n^2)(-1)^n.$$ ◀

8.2.3 Linear Nonhomogeneous Recurrence Relations with Constant Coefficients

We have seen how to solve linear homogeneous recurrence relations with constant coefficients. Is there a relatively simple technique for solving a linear, but not homogeneous, recurrence relation with constant coefficients, such as $a_n = 3a_{n-1} + 2n$? We will see that the answer is yes for certain families of such recurrence relations.

The recurrence relation $a_n = 3a_{n-1} + 2n$ is an example of a **linear nonhomogeneous recurrence relation with constant coefficients**, that is, a recurrence relation of the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + F(n),$$

where $c_1, c_2, \ldots, c_k$ are real numbers and $F(n)$ is a function not identically zero depending only on n. The recurrence relation

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$$

is called the **associated homogeneous recurrence relation**. It plays an important role in the solution of the nonhomogeneous recurrence relation.

EXAMPLE 9 Each of the recurrence relations $a_n = a_{n-1} + 2^n$, $a_n = a_{n-1} + a_{n-2} + n^2 + n + 1$, $a_n = 3a_{n-1} + n3^n$, and $a_n = a_{n-1} + a_{n-2} + a_{n-3} + n!$ is a linear nonhomogeneous recurrence relation with constant coefficients. The associated linear homogeneous recurrence relations are $a_n = a_{n-1}$, $a_n = a_{n-1} + a_{n-2}$, $a_n = 3a_{n-1}$, and $a_n = a_{n-1} + a_{n-2} + a_{n-3}$, respectively. ◀

The key fact about linear nonhomogeneous recurrence relations with constant coefficients is that every solution is the sum of a particular solution and a solution of the associated linear homogeneous recurrence relation, as Theorem 5 shows.

THEOREM 5 If $\{a_n^{(p)}\}$ is a particular solution of the nonhomogeneous linear recurrence relation with constant coefficients

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + F(n),$$

then every solution is of the form $\{a_n^{(p)} + a_n^{(h)}\}$, where $\{a_n^{(h)}\}$ is a solution of the associated homogeneous recurrence relation

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}.$$

Proof: Because $\{a_n^{(p)}\}$ is a particular solution of the nonhomogeneous recurrence relation, we know that

$$a_n^{(p)} = c_1 a_{n-1}^{(p)} + c_2 a_{n-2}^{(p)} + \cdots + c_k a_{n-k}^{(p)} + F(n).$$

Now suppose that $\{b_n\}$ is a second solution of the nonhomogeneous recurrence relation, so that

$$b_n = c_1 b_{n-1} + c_2 b_{n-2} + \cdots + c_k b_{n-k} + F(n).$$

Subtracting the first of these two equations from the second shows that

$$b_n - a_n^{(p)} = c_1(b_{n-1} - a_{n-1}^{(p)}) + c_2(b_{n-2} - a_{n-2}^{(p)}) + \cdots + c_k(b_{n-k} - a_{n-k}^{(p)}).$$

It follows that $\{b_n - a_n^p\}$ is a solution of the associated homogeneous linear recurrence, say, $\{a_n^{(h)}\}$. Consequently, $b_n = a_n^{(p)} + a_n^{(h)}$ for all n. ◁

By Theorem 5, we see that the key to solving nonhomogeneous recurrence relations with constant coefficients is finding a particular solution. Then every solution is a sum of this solution and a solution of the associated homogeneous recurrence relation. Although there is no general method for finding such a solution that works for every function $F(n)$, there are techniques that work for certain types of functions $F(n)$, such as polynomials and powers of constants. This is illustrated in Examples 10 and 11.

EXAMPLE 10 Find all solutions of the recurrence relation $a_n = 3a_{n-1} + 2n$. What is the solution with $a_1 = 3$?

Solution: To solve this linear nonhomogeneous recurrence relation with constant coefficients, we need to solve its associated linear homogeneous equation and to find a particular solution for the given nonhomogeneous equation. The associated linear homogeneous equation is $a_n = 3a_{n-1}$. Its solutions are $a_n^{(h)} = \alpha 3^n$, where α is a constant.

We now find a particular solution. Because $F(n) = 2n$ is a polynomial in n of degree one, a reasonable trial solution is a linear function in n, say, $p_n = cn + d$, where c and d are constants. To determine whether there are any solutions of this form, suppose that $p_n = cn + d$ is such a solution. Then the equation $a_n = 3a_{n-1} + 2n$ becomes $cn + d = 3(c(n-1) + d) + 2n$. Simplifying and combining like terms gives $(2 + 2c)n + (2d - 3c) = 0$. It follows that $cn + d$ is a solution if and only if $2 + 2c = 0$ and $2d - 3c = 0$. This shows that $cn + d$ is a solution if and only if $c = -1$ and $d = -3/2$. Consequently, $a_n^{(p)} = -n - 3/2$ is a particular solution.

By Theorem 5 all solutions are of the form

$$a_n = a_n^{(p)} + a_n^{(h)} = -n - \frac{3}{2} + \alpha \cdot 3^n,$$

where α is a constant.

To find the solution with $a_1 = 3$, let $n = 1$ in the formula we obtained for the general solution. We find that $3 = -1 - 3/2 + 3\alpha$, which implies that $\alpha = 11/6$. The solution we seek is $a_n = -n - 3/2 + (11/6)3^n$. ◀

EXAMPLE 11 Find all solutions of the recurrence relation

Extra Examples

$$a_n = 5a_{n-1} - 6a_{n-2} + 7^n.$$

Solution: This is a linear nonhomogeneous recurrence relation. The solutions of its associated homogeneous recurrence relation

$$a_n = 5a_{n-1} - 6a_{n-2}$$

are $a_n^{(h)} = \alpha_1 \cdot 3^n + \alpha_2 \cdot 2^n$, where α_1 and α_2 are constants. Because $F(n) = 7^n$, a reasonable trial solution is $a_n^{(p)} = C \cdot 7^n$, where C is a constant. Substituting the terms of this sequence into the recurrence relation implies that $C \cdot 7^n = 5C \cdot 7^{n-1} - 6C \cdot 7^{n-2} + 7^n$. Factoring out 7^{n-2}, this equation becomes $49C = 35C - 6C + 49$, which implies that $20C = 49$, or that $C = 49/20$. Hence, $a_n^{(p)} = (49/20)7^n$ is a particular solution. By Theorem 5, all solutions are of the form

$$a_n = \alpha_1 \cdot 3^n + \alpha_2 \cdot 2^n + (49/20)7^n.$$ ◀

In Examples 10 and 11, we made an educated guess that there are solutions of a particular form. In both cases we were able to find particular solutions. This was not an accident. Whenever $F(n)$ is the product of a polynomial in n and the nth power of a constant, we know exactly what form a particular solution has, as stated in Theorem 6. We leave the proof of Theorem 6 as Exercise 52.

THEOREM 6

Suppose that $\{a_n\}$ satisfies the linear nonhomogeneous recurrence relation

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + F(n),$$

where $c_1, c_2, \ldots, c_k$ are real numbers, and

$$F(n) = (b_t n^t + b_{t-1} n^{t-1} + \cdots + b_1 n + b_0)s^n,$$

where $b_0, b_1, \ldots, b_t$ and s are real numbers. When s is not a root of the characteristic equation of the associated linear homogeneous recurrence relation, there is a particular solution of the form

$$(p_t n^t + p_{t-1} n^{t-1} + \cdots + p_1 n + p_0)s^n.$$

When s is a root of this characteristic equation and its multiplicity is m, there is a particular solution of the form

$$n^m(p_t n^t + p_{t-1} n^{t-1} + \cdots + p_1 n + p_0)s^n.$$

Note that in the case when s is a root of multiplicity m of the characteristic equation of the associated linear homogeneous recurrence relation, the factor n^m ensures that the proposed particular solution will not already be a solution of the associated linear homogeneous recurrence relation. We next provide Example 12 to illustrate the form of a particular solution provided by Theorem 6.

EXAMPLE 12 What form does a particular solution of the linear nonhomogeneous recurrence relation $a_n = 6a_{n-1} - 9a_{n-2} + F(n)$ have when $F(n) = 3^n$, $F(n) = n3^n$, $F(n) = n^2 2^n$, and $F(n) = (n^2 + 1)3^n$?

Solution: The associated linear homogeneous recurrence relation is $a_n = 6a_{n-1} - 9a_{n-2}$. Its characteristic equation, $r^2 - 6r + 9 = (r - 3)^2 = 0$, has a single root, 3, of multiplicity two. To apply Theorem 6, with $F(n)$ of the form $P(n)s^n$, where $P(n)$ is a polynomial and s is a constant, we need to ask whether s is a root of this characteristic equation.

Because $s = 3$ is a root with multiplicity $m = 2$ but $s = 2$ is not a root, Theorem 6 tells us that a particular solution has the form $p_0 n^2 3^n$ if $F(n) = 3^n$, the form $n^2(p_1 n + p_0)3^n$ if $F(n) =$

$n3^n$, the form $(p_2n^2 + p_1n + p_0)2^n$ if $F(n) = n^2 2^n$, and the form $n^2(p_2n^2 + p_1n + p_0)3^n$ if $F(n) = (n^2 + 1)3^n$. ◀

Care must be taken when $s = 1$ when solving recurrence relations of the type covered by Theorem 6. In particular, to apply this theorem with $F(n) = b_t n_t + b_{t-1}n_{t-1} + \cdots + b_1 n + b_0$, the parameter s takes the value $s = 1$ (even though the term 1^n does not explicitly appear). By the theorem, the form of the solution then depends on whether 1 is a root of the characteristic equation of the associated linear homogeneous recurrence relation. This is illustrated in Example 13, which shows how Theorem 6 can be used to find a formula for the sum of the first n positive integers.

EXAMPLE 13 Let a_n be the sum of the first n positive integers, so that

$$a_n = \sum_{k=1}^{n} k.$$

Note that a_n satisfies the linear nonhomogeneous recurrence relation

$$a_n = a_{n-1} + n.$$

(To obtain a_n, the sum of the first n positive integers, from a_{n-1}, the sum of the first $n - 1$ positive integers, we add n.) Note that the initial condition is $a_1 = 1$.

The associated linear homogeneous recurrence relation for a_n is

$$a_n = a_{n-1}.$$

The solutions of this homogeneous recurrence relation are given by $a_n^{(h)} = c(1)^n = c$, where c is a constant. To find all solutions of $a_n = a_{n-1} + n$, we need find only a single particular solution. By Theorem 6, because $F(n) = n = n \cdot (1)^n$ and $s = 1$ is a root of degree one of the characteristic equation of the associated linear homogeneous recurrence relation, there is a particular solution of the form $n(p_1n + p_0) = p_1n^2 + p_0n$.

Inserting this into the recurrence relation gives $p_1n^2 + p_0n = p_1(n - 1)^2 + p_0(n - 1) + n$. Simplifying, we see that $n(2p_1 - 1) + (p_0 - p_1) = 0$, which means that $2p_1 - 1 = 0$ and $p_0 - p_1 = 0$, so $p_0 = p_1 = 1/2$. Hence,

$$a_n^{(p)} = \frac{n^2}{2} + \frac{n}{2} = \frac{n(n+1)}{2}$$

is a particular solution. Hence, all solutions of the original recurrence relation $a_n = a_{n-1} + n$ are given by $a_n = a_n^{(h)} + a_n^{(p)} = c + n(n + 1)/2$. Because $a_1 = 1$, we have $1 = a_1 = c + 1 \cdot 2/2 = c + 1$, so $c = 0$. It follows that $a_n = n(n + 1)/2$. (This is the same formula given in Table 2 in Section 2.4 and derived previously.) ◀

Exercises

1. Determine which of these are linear homogeneous recurrence relations with constant coefficients. Also, find the degree of those that are.

a) $a_n = 3a_{n-1} + 4a_{n-2} + 5a_{n-3}$
b) $a_n = 2na_{n-1} + a_{n-2}$
c) $a_n = a_{n-1} + a_{n-4}$
d) $a_n = a_{n-1} + 2$
e) $a_n = a_{n-1}^2 + a_{n-2}$
f) $a_n = a_{n-2}$
g) $a_n = a_{n-1} + n$

2. Determine which of these are linear homogeneous recurrence relations with constant coefficients. Also, find the degree of those that are.

a) $a_n = 3a_{n-2}$
b) $a_n = 3$
c) $a_n = a_{n-1}^2$
d) $a_n = a_{n-1} + 2a_{n-3}$
e) $a_n = a_{n-1}/n$
f) $a_n = a_{n-1} + a_{n-2} + n + 3$
g) $a_n = 4a_{n-2} + 5a_{n-4} + 9a_{n-7}$

3. Solve these recurrence relations together with the initial conditions given.

a) $a_n = 2a_{n-1}$ for $n \geq 1$, $a_0 = 3$
b) $a_n = a_{n-1}$ for $n \geq 1$, $a_0 = 2$
c) $a_n = 5a_{n-1} - 6a_{n-2}$ for $n \geq 2$, $a_0 = 1$, $a_1 = 0$
d) $a_n = 4a_{n-1} - 4a_{n-2}$ for $n \geq 2$, $a_0 = 6$, $a_1 = 8$
e) $a_n = -4a_{n-1} - 4a_{n-2}$ for $n \geq 2$, $a_0 = 0$, $a_1 = 1$
f) $a_n = 4a_{n-2}$ for $n \geq 2$, $a_0 = 0$, $a_1 = 4$
g) $a_n = a_{n-2}/4$ for $n \geq 2$, $a_0 = 1$, $a_1 = 0$

4. Solve these recurrence relations together with the initial conditions given.

a) $a_n = a_{n-1} + 6a_{n-2}$ for $n \geq 2$, $a_0 = 3$, $a_1 = 6$
b) $a_n = 7a_{n-1} - 10a_{n-2}$ for $n \geq 2$, $a_0 = 2$, $a_1 = 1$
c) $a_n = 6a_{n-1} - 8a_{n-2}$ for $n \geq 2$, $a_0 = 4$, $a_1 = 10$
d) $a_n = 2a_{n-1} - a_{n-2}$ for $n \geq 2$, $a_0 = 4$, $a_1 = 1$
e) $a_n = a_{n-2}$ for $n \geq 2$, $a_0 = 5$, $a_1 = -1$
f) $a_n = -6a_{n-1} - 9a_{n-2}$ for $n \geq 2$, $a_0 = 3$, $a_1 = -3$
g) $a_{n+2} = -4a_{n+1} + 5a_n$ for $n \geq 0$, $a_0 = 2$, $a_1 = 8$

5. How many different messages can be transmitted in n microseconds using the two signals described in Exercise 19 in Section 8.1?

6. How many different messages can be transmitted in n microseconds using three different signals if one signal requires 1 microsecond for transmittal, the other two signals require 2 microseconds each for transmittal, and a signal in a message is followed immediately by the next signal?

7. In how many ways can a $2 \times n$ rectangular checkerboard be tiled using 1×2 and 2×2 pieces?

8. A model for the number of lobsters caught per year is based on the assumption that the number of lobsters caught in a year is the average of the number caught in the two previous years.

a) Find a recurrence relation for $\{L_n\}$, where L_n is the number of lobsters caught in year n, under the assumption for this model.
b) Find L_n if 100,000 lobsters were caught in year 1 and 300,000 were caught in year 2.

9. A deposit of \$100,000 is made to an investment fund at the beginning of a year. On the last day of each year two dividends are awarded. The first dividend is 20% of the amount in the account during that year. The second dividend is 45% of the amount in the account in the previous year.

a) Find a recurrence relation for $\{P_n\}$, where P_n is the amount in the account at the end of n years if no money is ever withdrawn.
b) How much is in the account after n years if no money has been withdrawn?

***10.** Prove Theorem 2.

11. The **Lucas numbers** satisfy the recurrence relation

Links

$$L_n = L_{n-1} + L_{n-2},$$

and the initial conditions $L_0 = 2$ and $L_1 = 1$.

a) Show that $L_n = f_{n-1} + f_{n+1}$ for $n = 2, 3, \ldots$, where f_n is the nth Fibonacci number.
b) Find an explicit formula for the Lucas numbers.

12. Find the solution to $a_n = 2a_{n-1} + a_{n-2} - 2a_{n-3}$ for $n = 3, 4, 5, \ldots$, with $a_0 = 3$, $a_1 = 6$, and $a_2 = 0$.

13. Find the solution to $a_n = 7a_{n-2} + 6a_{n-3}$ with $a_0 = 9$, $a_1 = 10$, and $a_2 = 32$.

14. Find the solution to $a_n = 5a_{n-2} - 4a_{n-4}$ with $a_0 = 3$, $a_1 = 2$, $a_2 = 6$, and $a_3 = 8$.

15. Find the solution to $a_n = 2a_{n-1} + 5a_{n-2} - 6a_{n-3}$ with $a_0 = 7$, $a_1 = -4$, and $a_2 = 8$.

***16.** Prove Theorem 3.

17. Prove this identity relating the Fibonacci numbers and the binomial coefficients:

$$f_{n+1} = C(n, 0) + C(n-1, 1) + \cdots + C(n-k, k),$$

where n is a positive integer and $k = \lfloor n/2 \rfloor$. [*Hint:* Let $a_n = C(n, 0) + C(n-1, 1) + \cdots + C(n-k, k)$. Show that the sequence $\{a_n\}$ satisfies the same recurrence relation and initial conditions satisfied by the sequence of Fibonacci numbers.]

18. Solve the recurrence relation $a_n = 6a_{n-1} - 12a_{n-2} + 8a_{n-3}$ with $a_0 = -5$, $a_1 = 4$, and $a_2 = 88$.

19. Solve the recurrence relation $a_n = -3a_{n-1} - 3a_{n-2} - a_{n-3}$ with $a_0 = 5$, $a_1 = -9$, and $a_2 = 15$.

20. Find the general form of the solutions of the recurrence relation $a_n = 8a_{n-2} - 16a_{n-4}$.

21. What is the general form of the solutions of a linear homogeneous recurrence relation if its characteristic equation has roots 1, 1, 1, 1, −2, −2, −2, 3, 3, −4?

22. What is the general form of the solutions of a linear homogeneous recurrence relation if its characteristic equation has the roots −1, −1, −1, 2, 2, 5, 5, 7?

23. Consider the nonhomogeneous linear recurrence relation $a_n = 3a_{n-1} + 2^n$.

a) Show that $a_n = -2^{n+1}$ is a solution of this recurrence relation.
b) Use Theorem 5 to find all solutions of this recurrence relation.
c) Find the solution with $a_0 = 1$.

24. Consider the nonhomogeneous linear recurrence relation $a_n = 2a_{n-1} + 2^n$.

a) Show that $a_n = n2^n$ is a solution of this recurrence relation.
b) Use Theorem 5 to find all solutions of this recurrence relation.
c) Find the solution with $a_0 = 2$.

25. **a)** Determine values of the constants A and B such that $a_n = An + B$ is a solution of recurrence relation $a_n = 2a_{n-1} + n + 5$.
b) Use Theorem 5 to find all solutions of this recurrence relation.
c) Find the solution of this recurrence relation with $a_0 = 4$.

26. What is the general form of the particular solution guaranteed to exist by Theorem 6 of the linear nonhomogeneous recurrence relation $a_n = 6a_{n-1} - 12a_{n-2} + 8a_{n-3} + F(n)$ if

a) $F(n) = n^2$? **b)** $F(n) = 2^n$?
c) $F(n) = n2^n$? **d)** $F(n) = (-2)^n$?
e) $F(n) = n^2 2^n$? **f)** $F(n) = n^3(-2)^n$?
g) $F(n) = 3$?

27. What is the general form of the particular solution guaranteed to exist by Theorem 6 of the linear nonhomogeneous recurrence relation $a_n = 8a_{n-2} - 16a_{n-4} + F(n)$ if

a) $F(n) = n^3$? **b)** $F(n) = (-2)^n$?
c) $F(n) = n2^n$? **d)** $F(n) = n^2 4^n$?
e) $F(n) = (n^2 - 2)(-2)^n$? **f)** $F(n) = n^4 2^n$?
g) $F(n) = 2$?

28. a) Find all solutions of the recurrence relation $a_n = 2a_{n-1} + 2n^2$.
b) Find the solution of the recurrence relation in part (a) with initial condition $a_1 = 4$.

29. a) Find all solutions of the recurrence relation $a_n = 2a_{n-1} + 3^n$.
b) Find the solution of the recurrence relation in part (a) with initial condition $a_1 = 5$.

30. a) Find all solutions of the recurrence relation $a_n = -5a_{n-1} - 6a_{n-2} + 42 \cdot 4^n$.
b) Find the solution of this recurrence relation with $a_1 = 56$ and $a_2 = 278$.

31. Find all solutions of the recurrence relation $a_n = 5a_{n-1} - 6a_{n-2} + 2^n + 3n$. [*Hint:* Look for a particular solution of the form $qn2^n + p_1 n + p_2$, where q, p_1, and p_2 are constants.]

32. Find the solution of the recurrence relation $a_n = 2a_{n-1} + 3 \cdot 2^n$.

33. Find all solutions of the recurrence relation $a_n = 4a_{n-1} - 4a_{n-2} + (n+1)2^n$.

34. Find all solutions of the recurrence relation $a_n = 7a_{n-1} - 16a_{n-2} + 12a_{n-3} + n4^n$ with $a_0 = -2$, $a_1 = 0$, and $a_2 = 5$.

35. Find the solution of the recurrence relation $a_n = 4a_{n-1} - 3a_{n-2} + 2^n + n + 3$ with $a_0 = 1$ and $a_1 = 4$.

36. Let a_n be the sum of the first n perfect squares, that is, $a_n = \sum_{k=1}^{n} k^2$. Show that the sequence $\{a_n\}$ satisfies the linear nonhomogeneous recurrence relation $a_n = a_{n-1} + n^2$ and the initial condition $a_1 = 1$. Use Theorem 6 to determine a formula for a_n by solving this recurrence relation.

37. Let a_n be the sum of the first n triangular numbers, that is, $a_n = \sum_{k=1}^{n} t_k$, where $t_k = k(k+1)/2$. Show that $\{a_n\}$ satisfies the linear nonhomogeneous recurrence relation $a_n = a_{n-1} + n(n+1)/2$ and the initial condition $a_1 = 1$. Use Theorem 6 to determine a formula for a_n by solving this recurrence relation.

38. a) Find the characteristic roots of the linear homogeneous recurrence relation $a_n = 2a_{n-1} - 2a_{n-2}$. [*Note:* These are complex numbers.]
b) Find the solution of the recurrence relation in part (a) with $a_0 = 1$ and $a_1 = 2$.

***39. a)** Find the characteristic roots of the linear homogeneous recurrence relation $a_n = a_{n-4}$. [*Note:* These include complex numbers.]
b) Find the solution of the recurrence relation in part (a) with $a_0 = 1$, $a_1 = 0$, $a_2 = -1$, and $a_3 = 1$.

***40.** Solve the simultaneous recurrence relations

$$a_n = 3a_{n-1} + 2b_{n-1}$$
$$b_n = a_{n-1} + 2b_{n-1}$$

with $a_0 = 1$ and $b_0 = 2$.

***41. a)** Use the formula found in Example 4 for f_n, the nth Fibonacci number, to show that f_n is the integer closest to

$$\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n.$$

b) Determine for which n f_n is greater than

$$\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$$

and for which n f_n is less than

$$\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n.$$

42. Show that if $a_n = a_{n-1} + a_{n-2}$, $a_0 = s$ and $a_1 = t$, where s and t are constants, then $a_n = sf_{n-1} + tf_n$ for all positive integers n.

43. Express the solution of the linear nonhomogenous recurrence relation $a_n = a_{n-1} + a_{n-2} + 1$ for $n \geq 2$ where $a_0 = 0$ and $a_1 = 1$ in terms of the Fibonacci numbers. [*Hint:* Let $b_n = a_n + 1$ and apply Exercise 42 to the sequence b_n.]

***44.** (*Linear algebra required*) Let $\mathbf{A}_n$ be the $n \times n$ matrix with 2s on its main diagonal, 1s in all positions next to a diagonal element, and 0s everywhere else. Find a recurrence relation for d_n, the determinant of $\mathbf{A}_n$. Solve this recurrence relation to find a formula for d_n.

45. Suppose that each pair of a genetically engineered species of rabbits left on an island produces two new pairs of rabbits at the age of 1 month and six new pairs of rabbits at the age of 2 months and every month afterward. None of the rabbits ever die or leave the island.

a) Find a recurrence relation for the number of pairs of rabbits on the island n months after one newborn pair is left on the island.
b) By solving the recurrence relation in (a) determine the number of pairs of rabbits on the island n months after one pair is left on the island.

46. Suppose that there are two goats on an island initially. The number of goats on the island doubles every year by natural reproduction, and some goats are either added or removed each year.

a) Construct a recurrence relation for the number of goats on the island at the start of the nth year, assuming that during each year an extra 100 goats are put on the island.
b) Solve the recurrence relation from part (a) to find the number of goats on the island at the start of the nth year.
c) Construct a recurrence relation for the number of goats on the island at the start of the nth year, assuming that n goats are removed during the nth year for each $n \geq 3$.
d) Solve the recurrence relation in part (c) for the number of goats on the island at the start of the nth year.

47. A new employee at an exciting new software company starts with a salary of \$50,000 and is promised that at the end of each year her salary will be double her salary of the previous year, with an extra increment of \$10,000 for each year she has been with the company.
a) Construct a recurrence relation for her salary for her nth year of employment.
b) Solve this recurrence relation to find her salary for her nth year of employment.

Some linear recurrence relations that do not have constant coefficients can be systematically solved. This is the case for recurrence relations of the form $f(n)a_n = g(n)a_{n-1} + h(n)$. Exercises 48–50 illustrate this.

***48.** **a)** Show that the recurrence relation

$$f(n)a_n = g(n)a_{n-1} + h(n),$$

for $n \geq 1$, and with $a_0 = C$, can be reduced to a recurrence relation of the form

$$b_n = b_{n-1} + Q(n)h(n),$$

where $b_n = g(n+1)Q(n+1)a_n$, with

$$Q(n) = (f(1)f(2)\cdots f(n-1))/(g(1)g(2)\cdots g(n)).$$

b) Use part (a) to solve the original recurrence relation to obtain

$$a_n = \frac{C + \sum_{i=1}^{n} Q(i)h(i)}{g(n+1)Q(n+1)}.$$

***49.** Use Exercise 48 to solve the recurrence relation $(n+1)a_n = (n+3)a_{n-1} + n$, for $n \geq 1$, with $a_0 = 1$.

50. It can be shown that C_n, the average number of comparisons made by the quick sort algorithm (described in preamble to Exercise 50 in Section 5.4), when sorting n elements in random order, satisfies the recurrence relation

$$C_n = n + 1 + \frac{2}{n}\sum_{k=0}^{n-1} C_k$$

for $n = 1, 2, \ldots$, with initial condition $C_0 = 0$.
a) Show that $\{C_n\}$ also satisfies the recurrence relation $nC_n = (n+1)C_{n-1} + 2n$ for $n = 1, 2, \ldots$.
b) Use Exercise 48 to solve the recurrence relation in part (a) to find an explicit formula for C_n.

****51.** Prove Theorem 4.

****52.** Prove Theorem 6.

53. Solve the recurrence relation $T(n) = nT^2(n/2)$ with initial condition $T(1) = 6$ when $n = 2^k$ for some integer k. [*Hint:* Let $n = 2^k$ and then make the substitution $a_k = \log T(2^k)$ to obtain a linear nonhomogeneous recurrence relation.]

8.3 Divide-and-Conquer Algorithms and Recurrence Relations

8.3.1 Introduction

Links

Many recursive algorithms take a problem with a given input and divide it into one or more smaller problems. This reduction is successively applied until the solutions of the smaller problems can be found quickly. For instance, we perform a binary search by reducing the search for an element in a list to the search for this element in a list half as long. We successively apply this reduction until one element is left. When we sort a list of integers using the merge sort, we split the list into two halves of equal size and sort each half separately. We then merge the two sorted halves. Another example of this type of recursive algorithm is a procedure for multiplying integers that reduces the problem of the multiplication of two integers to three multiplications of pairs of integers with half as many bits. This reduction is successively applied until integers with one bit are obtained. There procedures follow an important algorithmic paradigm known as **divide-and-conquer**, and are called **divide-and-conquer algorithms**, because they *divide* a problem into one or more instances of the same problem of smaller size and they *conquer* the problem by using the solutions of the smaller problems to find a solution of the original problem, perhaps with some additional work.

"Divide et impera" (translation: *"Divide and conquer"*) —Julius Caesar

In this section we will show how recurrence relations can be used to analyze the computational complexity of divide-and-conquer algorithms. We will use these recurrence relations

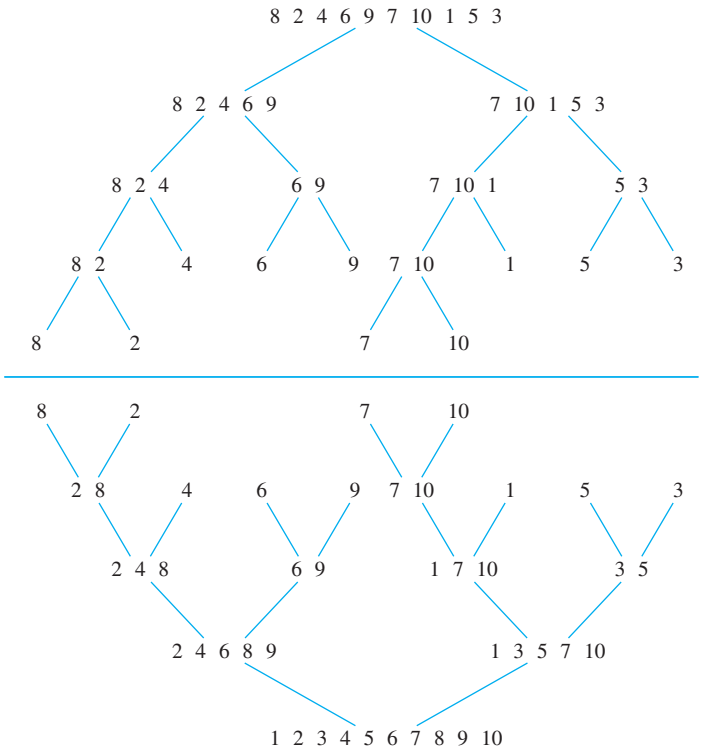


FIGURE 2 **The merge sort of 8, 2, 4, 6, 9, 7, 10, 1, 5, 3.**

5.4.4 The Merge Sort

Links

We now describe a recursive sorting algorithm called the **merge sort** algorithm. We will demonstrate how the merge sort algorithm works with an example before describing it in generality.

EXAMPLE 9 Use the merge sort to put the terms of the list 8, 2, 4, 6, 9, 7, 10, 1, 5, 3 in increasing order.

Solution: A merge sort begins by splitting the list into individual elements by successively splitting lists in two. The progression of sublists for this example is represented with the balanced binary tree of height 4 shown in the upper half of Figure 2.

Sorting is done by successively merging pairs of lists. At the first stage, pairs of individual elements are merged into lists of length two in increasing order. Then successive merges of pairs of lists are performed until the entire list is put into increasing order. The succession of merged lists in increasing order is represented by the balanced binary tree of height 4 shown in the lower half of Figure 2 (note that this tree is displayed "upside down"). ◀

Demo

In general, a merge sort proceeds by iteratively splitting lists into two sublists of equal length (or where one sublist has one more element than the other) until each sublist contains one element. This succession of sublists can be represented by a balanced binary tree. The procedure continues by successively merging pairs of lists, where both lists are in increasing order, into a larger list with elements in increasing order, until the original list is put into increasing order. The succession of merged lists can be represented by a balanced binary tree.

We can also describe the merge sort recursively. To do a merge sort, we split a list into two sublists of equal, or approximately equal, size, sorting each sublist using the merge sort

algorithm, and then merging the two lists. The recursive version of the merge sort is given in Algorithm 9. This algorithm uses the subroutine *merge*, which is described in Algorithm 10.

ALGORITHM 9 A Recursive Merge Sort.

procedure *mergesort*($L = a_1, \ldots, a_n$)
if $n > 1$ **then**
    $m := \lfloor n/2 \rfloor$
    $L_1 := a_1, a_2, \ldots, a_m$
    $L_2 := a_{m+1}, a_{m+2}, \ldots, a_n$
    $L :=$ *merge*(*mergesort*(L_1), *mergesort*(L_2))
{L is now sorted into elements in nondecreasing order}

An efficient algorithm for merging two ordered lists into a larger ordered list is needed to implement the merge sort. We will now describe such a procedure.

EXAMPLE 10 Merge the two lists 2, 3, 5, 6 and 1, 4.

Solution: Table 1 illustrates the steps we use. First, compare the smallest elements in the two lists, 2 and 1, respectively. Because 1 is the smaller, put it at the beginning of the merged list and remove it from the second list. At this stage, the first list is 2, 3, 5, 6, the second is 4, and the combined list is 1.

Next, compare 2 and 4, the smallest elements of the two lists. Because 2 is the smaller, add it to the combined list and remove it from the first list. At this stage the first list is 3, 5, 6, the second is 4, and the combined list is 1, 2.

Continue by comparing 3 and 4, the smallest elements of their respective lists. Because 3 is the smaller of these two elements, add it to the combined list and remove it from the first list. At this stage the first list is 5, 6, and the second is 4. The combined list is 1, 2, 3.

Then compare 5 and 4, the smallest elements in the two lists. Because 4 is the smaller of these two elements, add it to the combined list and remove it from the second list. At this stage the first list is 5, 6, the second list is empty, and the combined list is 1, 2, 3, 4.

Finally, because the second list is empty, all elements of the first list can be appended to the end of the combined list in the order they occur in the first list. This produces the ordered list 1, 2, 3, 4, 5, 6. ◀

We will now consider the general problem of merging two ordered lists L_1 and L_2 into an ordered list L. We will describe an algorithm for solving this problem. Start with an empty list L. Compare the smallest elements of the two lists. Put the smaller of these two elements

TABLE 1 Merging the Two Sorted Lists 2, 3, 5, 6 and 1, 4.

First List	*Second List*	*Merged List*	*Comparison*
2 3 5 6	1 4		1 < 2
2 3 5 6	4	1	2 < 4
3 5 6	4	1 2	3 < 4
5 6	4	1 2 3	4 < 5
5 6		1 2 3 4	
		1 2 3 4 5 6	

at the right end of L, and remove it from the list it was in. Next, if one of L_1 and L_2 is empty, append the other (nonempty) list to L, which completes the merging. If neither L_1 nor L_2 is empty, repeat this process. Algorithm 10 gives a pseudocode description of this procedure.

We will need estimates for the number of comparisons used to merge two ordered lists in the analysis of the merge sort. We can easily obtain such an estimate for Algorithm 10. Each time a comparison of an element from L_1 and an element from L_2 is made, an additional element is added to the merged list L. However, when either L_1 or L_2 is empty, no more comparisons are needed. Hence, Algorithm 10 is least efficient when $m + n - 2$ comparisons are carried out, where m and n are the number of elements in L_1 and L_2, respectively, leaving one element in each of L_1 and L_2. The next comparison will be the last one needed, because it will make one of these lists empty. Hence, Algorithm 10 uses no more than $m + n - 1$ comparisons. Lemma 1 summarizes this estimate.

ALGORITHM 10 Merging Two Lists.

procedure *merge*(L_1, L_2: sorted lists)
L := empty list
while L_1 and L_2 are both nonempty
  remove smaller of first elements of L_1 and L_2 from its list; put it at the right end of L
  if this removal makes one list empty **then** remove all elements from the other list and append them to L
return L {L is the merged list with elements in increasing order}

LEMMA 1 Two sorted lists with m elements and n elements can be merged into a sorted list using no more than $m + n - 1$ comparisons.

Sometimes two sorted lists of length m and n can be merged using far fewer than $m + n - 1$ comparisons. For instance, when $m = 1$, a binary search procedure can be applied to put the one element in the first list into the second list. This requires only $\lceil \log n \rceil$ comparisons, which is much smaller than $m + n - 1 = n$, for $m = 1$. On the other hand, for some values of m and n, Lemma 1 gives the best possible bound. That is, there are lists with m and n elements that cannot be merged using fewer than $m + n - 1$ comparisons. (See Exercise 47.)

We can now analyze the complexity of the merge sort. Instead of studying the general problem, we will assume that n, the number of elements in the list, is a power of 2, say 2^m. This will make the analysis less complicated, but when this is not the case, various modifications can be applied that will yield the same estimate.

At the first stage of the splitting procedure, the list is split into two sublists, of 2^{m-1} elements each, at level 1 of the tree generated by the splitting. This process continues, splitting the two sublists with 2^{m-1} elements into four sublists of 2^{m-2} elements each at level 2, and so on. In general, there are 2^{k-1} lists at level $k - 1$, each with 2^{m-k+1} elements. These lists at level $k - 1$ are split into 2^k lists at level k, each with 2^{m-k} elements. At the end of this process, we have 2^m lists each with one element at level m.

We start merging by combining pairs of the 2^m lists of one element into 2^{m-1} lists, at level $m - 1$, each with two elements. To do this, 2^{m-1} pairs of lists with one element each are merged. The merger of each pair requires exactly one comparison.

The procedure continues, so that at level k ($k = m, m - 1, m - 2, \ldots, 3, 2, 1$), 2^k lists each with 2^{m-k} elements are merged into 2^{k-1} lists, each with 2^{m-k+1} elements, at level $k - 1$. To do this a total of 2^{k-1} mergers of two lists, each with 2^{m-k} elements, are needed. But, by Lemma 1,

each of these mergers can be carried out using at most $2^{m-k} + 2^{m-k} - 1 = 2^{m-k+1} - 1$ comparisons. Hence, going from level k to $k-1$ can be accomplished using at most $2^{k-1}(2^{m-k+1} - 1)$ comparisons.

Summing all these estimates shows that the number of comparisons required for the merge sort is at most

$$\sum_{k=1}^{m} 2^{k-1}(2^{m-k+1} - 1) = \sum_{k=1}^{m} 2^m - \sum_{k=1}^{m} 2^{k-1} = m2^m - (2^m - 1) = n \log n - n + 1,$$

because $m = \log n$ and $n = 2^m$. (We evaluated $\sum_{k=1}^{m} 2^m$ by noting that it is the sum of m identical terms, each equal to 2^m. We evaluated $\sum_{k=1}^{m} 2^{k-1}$ using the formula for the sum of the terms of a geometric progression from Theorem 1 of Section 2.4.)

Theorem 1 summarizes what we have discovered about the worst-case complexity of the merge sort algorithm.

THEOREM 1 The number of comparisons needed to merge sort a list with n elements is $O(n \log n)$.

In Chapter 11 we will show that the fastest comparison-based sorting algorithm have $O(n \log n)$ time complexity. (A comparison-based sorting algorithm has the comparison of two elements as its basic operation.) Theorem 1 tells us that the merge sort achieves this best possible big-O estimate for the complexity of a sorting algorithm. We describe another efficient algorithm, the quick sort, in the preamble to Exercise 50.

Exercises

1. Trace Algorithm 1 when it is given $n = 5$ as input. That is, show all steps used by Algorithm 1 to find 5!, as is done in Example 1 to find 4!.
2. Trace Algorithm 1 when it is given $n = 6$ as input. That is, show all steps used by Algorithm 1 to find 6!, as is done in Example 1 to find 4!.
3. Trace Algorithm 3 when it finds gcd(8, 13). That is, show all the steps used by Algorithm 3 to find gcd(8, 13).
4. Trace Algorithm 3 when it finds gcd(12, 17). That is, show all the steps used by Algorithm 3 to find gcd(12, 17).
5. Trace Algorithm 4 when it is given $m = 5$, $n = 11$, and $b = 3$ as input. That is, show all the steps Algorithm 4 uses to find 3^{11} **mod** 5.
6. Trace Algorithm 4 when it is given $m = 7$, $n = 10$, and $b = 2$ as input. That is, show all the steps Algorithm 4 uses to find 2^{10} **mod** 7.
7. Give a recursive algorithm for computing nx whenever n is a positive integer and x is an integer, using just addition.
8. Give a recursive algorithm for finding the sum of the first n positive integers.
9. Give a recursive algorithm for finding the sum of the first n odd positive integers.
10. Give a recursive algorithm for finding the maximum of a finite set of integers, making use of the fact that the maximum of n integers is the larger of the last integer in the list and the maximum of the first $n - 1$ integers in the list.
11. Give a recursive algorithm for finding the minimum of a finite set of integers, making use of the fact that the minimum of n integers is the smaller of the last integer in the list and the minimum of the first $n - 1$ integers in the list.
12. Devise a recursive algorithm for finding x^n **mod** m whenever n, x, and m are positive integers based on the fact that x^n **mod** $m = (x^{n-1}$ **mod** $m \cdot x$ **mod** $m)$ **mod** m.
13. Give a recursive algorithm for finding $n!$ **mod** m whenever n and m are positive integers.
14. Give a recursive algorithm for finding a **mode** of a list of integers. (A **mode** is an element in the list that occurs at least as often as every other element.)
15. Devise a recursive algorithm for computing the greatest common divisor of two nonnegative integers a and b with $a < b$ using the fact that $\gcd(a, b) = \gcd(a, b - a)$.
16. Prove that the recursive algorithm for finding the sum of the first n positive integers you found in Exercise 8 is correct.

a) Construct a recurrence relation for the number of goats on the island at the start of the nth year, assuming that during each year an extra 100 goats are put on the island.
b) Solve the recurrence relation from part (a) to find the number of goats on the island at the start of the nth year.
c) Construct a recurrence relation for the number of goats on the island at the start of the nth year, assuming that n goats are removed during the nth year for each $n \geq 3$.
d) Solve the recurrence relation in part (c) for the number of goats on the island at the start of the nth year.

47. A new employee at an exciting new software company starts with a salary of \$50,000 and is promised that at the end of each year her salary will be double her salary of the previous year, with an extra increment of \$10,000 for each year she has been with the company.
a) Construct a recurrence relation for her salary for her nth year of employment.
b) Solve this recurrence relation to find her salary for her nth year of employment.

Some linear recurrence relations that do not have constant coefficients can be systematically solved. This is the case for recurrence relations of the form $f(n)a_n = g(n)a_{n-1} + h(n)$. Exercises 48–50 illustrate this.

***48.** **a)** Show that the recurrence relation

$$f(n)a_n = g(n)a_{n-1} + h(n),$$

for $n \geq 1$, and with $a_0 = C$, can be reduced to a recurrence relation of the form

$$b_n = b_{n-1} + Q(n)h(n),$$

where $b_n = g(n+1)Q(n+1)a_n$, with

$$Q(n) = (f(1)f(2)\cdots f(n-1))/(g(1)g(2)\cdots g(n)).$$

b) Use part (a) to solve the original recurrence relation to obtain

$$a_n = \frac{C + \sum_{i=1}^{n} Q(i)h(i)}{g(n+1)Q(n+1)}.$$

***49.** Use Exercise 48 to solve the recurrence relation $(n+1)a_n = (n+3)a_{n-1} + n$, for $n \geq 1$, with $a_0 = 1$.

50. It can be shown that C_n, the average number of comparisons made by the quick sort algorithm (described in preamble to Exercise 50 in Section 5.4), when sorting n elements in random order, satisfies the recurrence relation

$$C_n = n + 1 + \frac{2}{n}\sum_{k=0}^{n-1} C_k$$

for $n = 1, 2, \ldots$, with initial condition $C_0 = 0$.
a) Show that $\{C_n\}$ also satisfies the recurrence relation $nC_n = (n+1)C_{n-1} + 2n$ for $n = 1, 2, \ldots$.
b) Use Exercise 48 to solve the recurrence relation in part (a) to find an explicit formula for C_n.

****51.** Prove Theorem 4.

****52.** Prove Theorem 6.

53. Solve the recurrence relation $T(n) = nT^2(n/2)$ with initial condition $T(1) = 6$ when $n = 2^k$ for some integer k. [*Hint:* Let $n = 2^k$ and then make the substitution $a_k = \log T(2^k)$ to obtain a linear nonhomogeneous recurrence relation.]

8.3 Divide-and-Conquer Algorithms and Recurrence Relations

8.3.1 Introduction

Links

Many recursive algorithms take a problem with a given input and divide it into one or more smaller problems. This reduction is successively applied until the solutions of the smaller problems can be found quickly. For instance, we perform a binary search by reducing the search for an element in a list to the search for this element in a list half as long. We successively apply this reduction until one element is left. When we sort a list of integers using the merge sort, we split the list into two halves of equal size and sort each half separately. We then merge the two sorted halves. Another example of this type of recursive algorithm is a procedure for multiplying integers that reduces the problem of the multiplication of two integers to three multiplications of pairs of integers with half as many bits. This reduction is successively applied until integers with one bit are obtained. There procedures follow an important algorithmic paradigm known as **divide-and-conquer**, and are called **divide-and-conquer algorithms**, because they *divide* a problem into one or more instances of the same problem of smaller size and they *conquer* the problem by using the solutions of the smaller problems to find a solution of the original problem, perhaps with some additional work.

"Divide et impera" (translation: *"Divide and conquer"*) —Julius Caesar

In this section we will show how recurrence relations can be used to analyze the computational complexity of divide-and-conquer algorithms. We will use these recurrence relations

to estimate the number of operations used by many different divide-and-conquer algorithms, including several that we introduce in this section.

8.3.2 Divide-and-Conquer Recurrence Relations

Suppose that a recursive algorithm divides a problem of size n into a subproblems, where each subproblem is of size n/b (for simplicity, assume that n is a multiple of b; in reality, the smaller problems are often of size equal to the nearest integers either less than or equal to, or greater than or equal to, n/b). Also, suppose that a total of $g(n)$ extra operations are required in the conquer step of the algorithm to combine the solutions of the subproblems into a solution of the original problem. Then, if $f(n)$ represents the number of operations required to solve the problem of size n, it follows that f satisfies the recurrence relation

$$f(n) = af(n/b) + g(n).$$

This is called a **divide-and-conquer recurrence relation**.

We will first set up the divide-and-conquer recurrence relations that can be used to study the complexity of some important algorithms. Then we will show how to use these divide-and-conquer recurrence relations to estimate the complexity of these algorithms.

EXAMPLE 1 **Binary Search** We introduced a binary search algorithm in Section 3.1. This binary search algorithm reduces the search for an element in a search sequence of size n to the binary search for this element in a search sequence of size $n/2$, when n is even. (Hence, the problem of size n has been reduced to *one* problem of size $n/2$.) Two comparisons are needed to implement this reduction (one to determine which half of the list to use and the other to determine whether any terms of the list remain). Hence, if $f(n)$ is the number of comparisons required to search for an element in a search sequence of size n, then

Extra Examples

$$f(n) = f(n/2) + 2$$

when n is even. ◀

EXAMPLE 2 **Finding the Maximum and Minimum of a Sequence** Consider the following algorithm for locating the maximum and minimum elements of a sequence $a_1, a_2, \ldots, a_n$. If $n = 1$, then a_1 is the maximum and the minimum. If $n > 1$, split the sequence into two sequences, either where both have the same number of elements or where one of the sequences has one more element than the other. The problem is reduced to finding the maximum and minimum of each of the two smaller sequences. The solution to the original problem results from the comparison of the separate maxima and minima of the two smaller sequences to obtain the overall maximum and minimum.

Let $f(n)$ be the total number of comparisons needed to find the maximum and minimum elements of the sequence with n elements. We have shown that a problem of size n can be reduced into two problems of size $n/2$, when n is even, using two comparisons, one to compare the maxima of the two sequences and the other to compare the minima of the two sequences. This gives the recurrence relation

$$f(n) = 2f(n/2) + 2$$

when n is even. ◀

EXAMPLE 3 **Merge Sort** The merge sort algorithm (introduced in Section 5.4) splits a list to be sorted with n items, where n is even, into two lists with $n/2$ elements each, and uses fewer than n

comparisons to merge the two sorted lists of $n/2$ items each into one sorted list. Consequently, the number of comparisons used by the merge sort to sort a list of n elements is less than $M(n)$, where the function $M(n)$ satisfies the divide-and-conquer recurrence relation

$$M(n) = 2M(n/2) + n.$$ ◀

EXAMPLE 4 **Fast Multiplication of Integers** Surprisingly, there are more efficient algorithms than the conventional algorithm (described in Section 4.2) for multiplying integers. One of these algorithms, which uses a divide-and-conquer technique, will be described here. This fast multiplication algorithm proceeds by splitting each of two $2n$-bit integers into two blocks, each with n bits. Then, the original multiplication is reduced from the multiplication of two $2n$-bit integers to three multiplications of n-bit integers, plus shifts and additions.

Links

Suppose that a and b are integers with binary expansions of length $2n$ (add initial bits of zero in these expansions if necessary to make them the same length). Let

$$a = (a_{2n-1}a_{2n-2} \cdots a_1a_0)_2 \quad \text{and} \quad b = (b_{2n-1}b_{2n-2} \cdots b_1b_0)_2.$$

Let

$$a = 2^nA_1 + A_0, \quad b = 2^nB_1 + B_0,$$

where

$$A_1 = (a_{2n-1} \cdots a_{n+1}a_n)_2, \qquad A_0 = (a_{n-1} \cdots a_1a_0)_2,$$
$$B_1 = (b_{2n-1} \cdots b_{n+1}b_n)_2, \qquad B_0 = (b_{n-1} \cdots b_1b_0)_2.$$

The algorithm for fast multiplication of integers is based on the fact that ab can be rewritten as

$$ab = (2^{2n} + 2^n)A_1B_1 + 2^n(A_1 - A_0)(B_0 - B_1) + (2^n + 1)A_0B_0.$$

The important fact about this identity is that it shows that the multiplication of two $2n$-bit integers can be carried out using three multiplications of n-bit integers, together with additions, subtractions, and shifts. This shows that if $f(n)$ is the total number of bit operations needed to multiply two n-bit integers, then

$$f(2n) = 3f(n) + Cn.$$

The reasoning behind this equation is as follows. The three multiplications of n-bit integers are carried out using $3f(n)$-bit operations. Each of the additions, subtractions, and shifts uses a constant multiple of n-bit operations, and Cn represents the total number of bit operations used by these operations. ◀

EXAMPLE 5 **Fast Matrix Multiplication** In Example 7 of Section 3.3 we showed that multiplying two $n \times n$ matrices using the definition of matrix multiplication required n^3 multiplications and $n^2(n-1)$ additions. Consequently, computing the product of two $n \times n$ matrices in this way requires $O(n^3)$ operations (multiplications and additions). Surprisingly, there are more efficient divide-and-conquer algorithms for multiplying two $n \times n$ matrices. Such an algorithm, invented by Volker Strassen in 1969, reduces the multiplication of two $n \times n$ matrices, when n is even, to seven multiplications of two $(n/2) \times (n/2)$ matrices and 15 additions of $(n/2) \times (n/2)$ matrices.

Links

(See [CoLeRiSt09] for the details of this algorithm.) Hence, if $f(n)$ is the number of operations (multiplications and additions) used, it follows that

$$f(n) = 7f(n/2) + 15n^2/4$$

when n is even. ◀

As Examples 1–5 show, recurrence relations of the form $f(n) = af(n/b) + g(n)$ arise in many different situations. It is possible to derive estimates of the size of functions that satisfy such recurrence relations. Suppose that f satisfies this recurrence relation whenever n is divisible by b. Let $n = b^k$, where k is a positive integer. Then

$$\begin{aligned} f(n) &= af(n/b) + g(n) \\ &= a^2 f(n/b^2) + ag(n/b) + g(n) \\ &= a^3 f(n/b^3) + a^2 g(n/b^2) + ag(n/b) + g(n) \\ &\vdots \\ &= a^k f(n/b^k) + \sum_{j=0}^{k-1} a^j g(n/b^j). \end{aligned}$$

Because $n/b^k = 1$, it follows that

$$f(n) = a^k f(1) + \sum_{j=0}^{k-1} a^j g(n/b^j).$$

We can use this equation for $f(n)$ to estimate the size of functions that satisfy divide-and-conquer relations.

THEOREM 1

Let f be an increasing function that satisfies the recurrence relation

$$f(n) = af(n/b) + c$$

whenever n is divisible by b, where $a \geq 1$, b is an integer greater than 1, and c is a positive real number. Then

$$f(n) \text{ is } \begin{cases} O(n^{\log_b a}) \text{ if } a > 1, \\ O(\log n) \text{ if } a = 1. \end{cases}$$

Furthermore, when $n = b^k$ and $a \neq 1$, where k is a positive integer,

$$f(n) = C_1 n^{\log_b a} + C_2,$$

where $C_1 = f(1) + c/(a-1)$ and $C_2 = -c/(a-1)$.

Proof: First let $n = b^k$. From the expression for $f(n)$ obtained in the discussion preceding the theorem, with $g(n) = c$, we have

$$f(n) = a^k f(1) + \sum_{j=0}^{k-1} a^j c = a^k f(1) + c \sum_{j=0}^{k-1} a^j.$$

When $a = 1$ we have

$$f(n) = f(1) + ck.$$

Because $n = b^k$, we have $k = \log_b n$. Hence,

$$f(n) = f(1) + c \log_b n.$$

When n is not a power of b, we have $b^k < n < b^{k+1}$, for a positive integer k. Because f is increasing, it follows that $f(n) \leq f(b^{k+1}) = f(1) + c(k+1) = (f(1) + c) + ck \leq (f(1) + c) + c \log_b n$. Therefore, in both cases, $f(n)$ is $O(\log n)$ when $a = 1$.

Now suppose that $a > 1$. First assume that $n = b^k$, where k is a positive integer. From the formula for the sum of terms of a geometric progression (Theorem 1 in Section 2.4), it follows that

$$\begin{aligned} f(n) &= a^k f(1) + c(a^k - 1)/(a - 1) \\ &= a^k[f(1) + c/(a-1)] - c/(a-1) \\ &= C_1 n^{\log_b a} + C_2, \end{aligned}$$

because $a^k = a^{\log_b n} = n^{\log_b a}$ (see Exercise 4 in Appendix 2), where $C_1 = f(1) + c/(a-1)$ and $C_2 = -c/(a-1)$.

Now suppose that n is not a power of b. Then $b^k < n < b^{k+1}$, where k is a nonnegative integer. Because f is increasing,

$$\begin{aligned} f(n) &\leq f(b^{k+1}) = C_1 a^{k+1} + C_2 \\ &\leq (C_1 a) a^{\log_b n} + C_2 \\ &= (C_1 a) n^{\log_b a} + C_2, \end{aligned}$$

because $k \leq \log_b n < k + 1$.

Hence, we have $f(n)$ is $O(n^{\log_b a})$. ◁

Examples 6–9 illustrate how Theorem 1 is used.

EXAMPLE 6 *Extra Examples*

Let $f(n) = 5f(n/2) + 3$ and $f(1) = 7$. Find $f(2^k)$, where k is a positive integer. Also, estimate $f(n)$ if f is an increasing function.

Solution: From the proof of Theorem 1, with $a = 5$, $b = 2$, and $c = 3$, we see that if $n = 2^k$, then

$$\begin{aligned} f(n) &= a^k[f(1) + c/(a-1)] + [-c/(a-1)] \\ &= 5^k[7 + (3/4)] - 3/4 \\ &= 5^k(31/4) - 3/4. \end{aligned}$$

Also, if $f(n)$ is increasing, Theorem 1 shows that $f(n)$ is $O(n^{\log_b a}) = O(n^{\log 5})$. ◀

We can use Theorem 1 to estimate the computational complexity of the binary search algorithm and the algorithm given in Example 2 for locating the minimum and maximum of a sequence.

EXAMPLE 7 Give a big-O estimate for the number of comparisons used by a binary search.

Solution: In Example 1 it was shown that $f(n) = f(n/2) + 2$ when n is even, where f is the number of comparisons required to perform a binary search on a sequence of size n. Hence, from Theorem 1, it follows that $f(n)$ is $O(\log n)$. ◀

EXAMPLE 8 Give a big-O estimate for the number of comparisons used to locate the maximum and minimum elements in a sequence using the algorithm given in Example 2.

Solution: In Example 2 we showed that $f(n) = 2f(n/2) + 2$, when n is even, where f is the number of comparisons needed by this algorithm. Hence, from Theorem 1, it follows that $f(n)$ is $O(n^{\log 2}) = O(n)$. ◀

We now state a more general, and more complicated, theorem, which has Theorem 1 as a special case. This theorem (or more powerful versions, including big-Theta estimates) is sometimes known as the master theorem because it is useful in analyzing the complexity of many important divide-and-conquer algorithms.

THEOREM 2

MASTER THEOREM Let f be an increasing function that satisfies the recurrence relation

$$f(n) = af(n/b) + cn^d$$

whenever $n = b^k$, where k is a positive integer, $a \geq 1$, b is an integer greater than 1, and c and d are real numbers with c positive and d nonnegative. Then

$$f(n) \text{ is } \begin{cases} O(n^d) & \text{if } a < b^d, \\ O(n^d \log n) & \text{if } a = b^d, \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

The proof of Theorem 2 is left for the reader as Exercises 29–33.

EXAMPLE 9 **Complexity of Merge Sort** In Example 3 we explained that the number of comparisons used by the merge sort to sort a list of n elements is less than $M(n)$, where $M(n) = 2M(n/2) + n$. By the master theorem (Theorem 2) we find that $M(n)$ is $O(n \log n)$, which agrees with the estimate found in Section 5.4. ◀

EXAMPLE 10 Give a big-O estimate for the number of bit operations needed to multiply two n-bit integers using the fast multiplication algorithm described in Example 4.

Solution: Example 4 shows that $f(n) = 3f(n/2) + Cn$, when n is even, where $f(n)$ is the number of bit operations required to multiply two n-bit integers using the fast multiplication algorithm. Hence, from the master theorem (Theorem 2), it follows that $f(n)$ is $O(n^{\log 3})$. Note that $\log 3 \sim 1.6$. Because the conventional algorithm for multiplication uses $O(n^2)$ bit operations, the fast multiplication algorithm is a substantial improvement over the conventional algorithm in

terms of time complexity for sufficiently large integers, including large integers that occur in practical applications. ◀

EXAMPLE 11 Give a big-O estimate for the number of multiplications and additions required to multiply two $n \times n$ matrices using the matrix multiplication algorithm referred to in Example 5.

Solution: Let $f(n)$ denote the number of additions and multiplications used by the algorithm mentioned in Example 5 to multiply two $n \times n$ matrices. We have $f(n) = 7f(n/2) + 15n^2/4$, when n is even. Hence, from the master theorem (Theorem 2), it follows that $f(n)$ is $O(n^{\log 7})$. Note that $\log 7 \sim 2.8$. Because the conventional algorithm for multiplying two $n \times n$ matrices uses $O(n^3)$ additions and multiplications, it follows that for sufficiently large integers n, including those that occur in many practical applications, this algorithm is substantially more efficient in time complexity than the conventional algorithm. ◀

THE CLOSEST-PAIR PROBLEM We conclude this section by introducing a divide-and-conquer algorithm from computational geometry, the part of discrete mathematics devoted to algorithms that solve geometric problems.

EXAMPLE 12 **The Closest-Pair Problem** Consider the problem of determining the closest pair of points in a set of n points $(x_1, y_1), \ldots, (x_n, y_n)$ in the plane, where the distance between two points (x_i, y_i) and (x_j, y_j) is the usual Euclidean distance $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. This problem arises in many applications such as determining the closest pair of airplanes in the air space at a particular altitude being managed by an air traffic controller. How can this closest pair of points be found in an efficient way?

Links

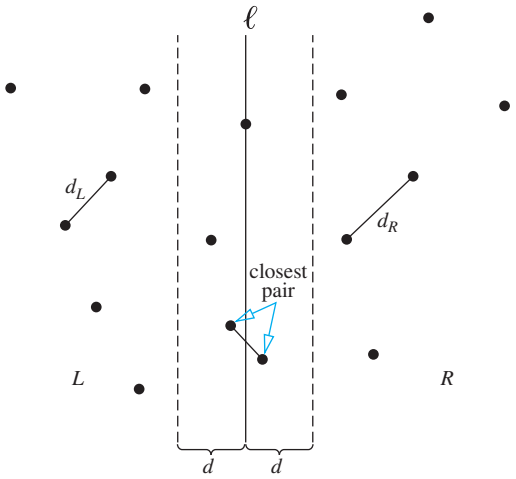
It took researchers more than 10 years to find an algorithm with $O(n \log n)$ complexity that locates the closest pair of points among n points.

Solution: To solve this problem we can first determine the distance between every pair of points and then find the smallest of these distances. However, this approach requires $O(n^2)$ computations of distances and comparisons because there are $C(n, 2) = n(n-1)/2$ pairs of points. Surprisingly, there is an elegant divide-and-conquer algorithm that can solve the closest-pair problem for n points using $O(n \log n)$ computations of distances and comparisons. The algorithm we describe here is due to Michael Samos (see [PrSa85]).

For simplicity, we assume that $n = 2^k$, where k is a positive integer. (We avoid some technical considerations that are needed when n is not a power of 2.) When $n = 2$, we have only one pair of points; the distance between these two points is the minimum distance. At the start of the algorithm we use the merge sort twice, once to sort the points in order of increasing x coordinates, and once to sort the points in order of increasing y coordinates. Each of these sorts requires $O(n \log n)$ operations. We will use these sorted lists in each recursive step.

The recursive part of the algorithm divides the problem into two subproblems, each involving half as many points. Using the sorted list of the points by their x coordinates, we construct a vertical line ℓ dividing the n points into two parts, a left part and a right part of equal size, each containing $n/2$ points, as shown in Figure 1. (If any points fall on the dividing line ℓ, we divide them among the two parts if necessary.) At subsequent steps of the recursion we need not sort on x coordinates again, because we can select the corresponding sorted subset of all the points. This selection is a task that can be done with $O(n)$ comparisons.

There are three possibilities concerning the positions of the closest points: (1) they are both in the left region L, (2) they are both in the right region R, or (3) one point is in the left region and the other is in the right region. Apply the algorithm recursively to compute d_L and d_R, where d_L is the minimum distance between points in the left region and d_R is the minimum distance between points in the right region. Let $d = \min(d_L, d_R)$. To successfully divide the problem of finding the closest two points in the original set into the two problems of finding the



In this illustration the problem of finding the closest pair in a set of 16 points is reduced to two problems of finding the closest pair in a set of eight points *and* the problem of determining whether there are points closer than $d = \min(d_L, d_R)$ within the strip of width $2d$ centered at ℓ.

FIGURE 1 **The recursive step of the algorithm for solving the closest-pair problem.**

shortest distances between points in the two regions separately, we have to handle the conquer part of the algorithm, which requires that we consider the case where the closest points lie in different regions, that is, one point is in L and the other in R. Because there is a pair of points at distance d where both points lie in R or both points lie in L, for the closest points to lie in different regions requires that they must be a distance less than d apart.

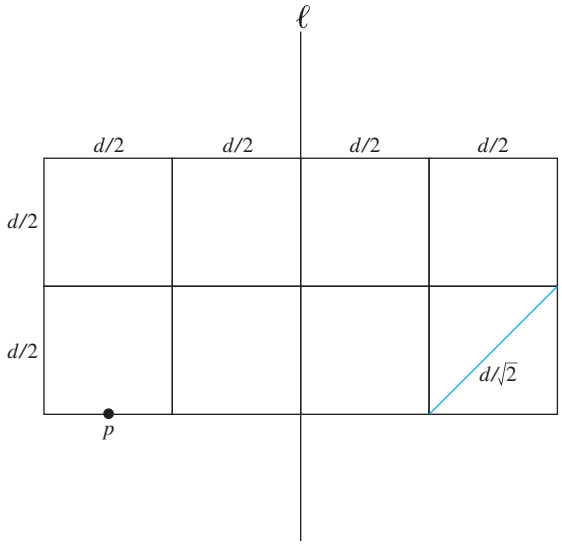
For a point in the left region and a point in the right region to lie at a distance less than d apart, these points must lie in the vertical strip of width $2d$ that has the line ℓ as its center. (Otherwise, the distance between these points is greater than the difference in their x coordinates, which exceeds d.) To examine the points within this strip, we sort the points so that they are listed in order of increasing y coordinates, using the sorted list of the points by their y coordinates. At each recursive step, we form a subset of the points in the region sorted by their y coordinates from the already sorted set of all points sorted by their y coordinates, which can be done with $O(n)$ comparisons.

Beginning with a point in the strip with the smallest y coordinate, we successively examine each point in the strip, computing the distance between this point and all other points in the strip that have larger y coordinates that could lie at a distance less than d from this point. Note that to examine a point p, we need only consider the distances between p and points in the set that lie within the rectangle of height d and width $2d$ with p on its base and with vertical sides at distance d from ℓ.

We can show that there are at most eight points from the set, including p, in or on this $2d \times d$ rectangle. To see this, note that there can be at most one point in each of the eight $d/2 \times d/2$ squares shown in Figure 2. This follows because the farthest apart points can be on or within one of these squares is the diagonal length $d/\sqrt{2}$ (which can be found using the Pythagorean theorem), which is less than d, and each of these $d/2 \times d/2$ squares lies entirely within the left region or the right region. This means that at this stage we need only compare at most seven distances, the distances between p and the seven or fewer other points in or on the rectangle, with d.

Because the total number of points in the strip of width $2d$ does not exceed n (the total number of points in the set), at most $7n$ distances need to be compared with d to find the minimum distance between points. That is, there are only $7n$ possible distances that could be less than d. Consequently, once the merge sort has been used to sort the pairs according to their x coordinates and according to their y coordinates, we find that the increasing function $f(n)$ satisfying the recurrence relation

$$f(n) = 2f(n/2) + 7n,$$



At most eight points, including p, can lie in or on the $2d \times d$ rectangle centered at ℓ because at most one point can lie in or on each of the eight $(d/2) \times (d/2)$ squares.

FIGURE 2 **Showing that there are at most seven other points to consider for each point in the strip.**

where $f(2) = 1$, exceeds the number of comparisons needed to solve the closest-pair problem for n points. By the master theorem (Theorem 2), it follows that $f(n)$ is $O(n \log n)$. The two sorts of points by their x coordinates and by their y coordinates each can be done using $O(n \log n)$ comparisons, by using the merge sort, and the sorted subsets of these coordinates at each of the $O(\log n)$ steps of the algorithm can be done using $O(n)$ comparisons each. Thus, we find that the closest-pair problem can be solved using $O(n \log n)$ comparisons. ◀

Exercises

1. How many comparisons are needed for a binary search in a set of 64 elements?
2. How many comparisons are needed to locate the maximum and minimum elements in a sequence with 128 elements using the algorithm in Example 2?
3. Multiply $(1110)_2$ and $(1010)_2$ using the fast multiplication algorithm.
4. Express the fast multiplication algorithm in pseudocode.
5. Determine a value for the constant C in Example 4 and use it to estimate the number of bit operations needed to multiply two 64-bit integers using the fast multiplication algorithm.
6. How many operations are needed to multiply two 32×32 matrices using the algorithm referred to in Example 5?
7. Suppose that $f(n) = f(n/3) + 1$ when n is a positive integer divisible by 3, and $f(1) = 1$. Find
   a) $f(3)$. **b)** $f(27)$. **c)** $f(729)$.
8. Suppose that $f(n) = 2f(n/2) + 3$ when n is an even positive integer, and $f(1) = 5$. Find
   a) $f(2)$. **b)** $f(8)$. **c)** $f(64)$. **d)** $f(1024)$.
9. Suppose that $f(n) = f(n/5) + 3n^2$ when n is a positive integer divisible by 5, and $f(1) = 4$. Find
   a) $f(5)$. **b)** $f(125)$. **c)** $f(3125)$.
10. Find $f(n)$ when $n = 2^k$, where f satisfies the recurrence relation $f(n) = f(n/2) + 1$ with $f(1) = 1$.
11. Give a big-O estimate for the function f in Exercise 10 if f is an increasing function.
12. Find $f(n)$ when $n = 3^k$, where f satisfies the recurrence relation $f(n) = 2f(n/3) + 4$ with $f(1) = 1$.
13. Give a big-O estimate for the function f in Exercise 12 if f is an increasing function.
14. Suppose that there are $n = 2^k$ teams in an elimination tournament, where there are $n/2$ games in the first round, with the $n/2 = 2^{k-1}$ winners playing in the second round, and so on. Develop a recurrence relation for the number of rounds in the tournament.
15. How many rounds are in the elimination tournament described in Exercise 14 when there are 32 teams?
16. Solve the recurrence relation for the number of rounds in the tournament described in Exercise 14.
17. Suppose that the votes of n people for different candidates (where there can be more than two candidates) for a particular office are the elements of a sequence. A person wins the election if this person receives a majority of the votes.