41. Sort these lists using the selection sort.
   a) 3, 5, 4, 1, 2   b) 5, 4, 3, 2, 1   c) 1, 2, 3, 4, 5
42. Write the selection sort algorithm in pseudocode.
43. Describe an algorithm based on the linear search for determining the correct position in which to insert a new element in an already sorted list.
44. Describe an algorithm based on the binary search for determining the correct position in which to insert a new element in an already sorted list.
45. How many comparisons does the insertion sort use to sort the list 1, 2, . . . , n?
46. How many comparisons does the insertion sort use to sort the list n, n - 1, . . . , 2, 1?

The binary insertion sort is a variation of the insertion sort that uses a binary search technique (see Exercise 44) rather than a linear search technique to insert the i-th element in the correct place among the previously sorted elements.

47. Show all the steps used by the binary insertion sort to sort the list 3, 2, 4, 5, 1, 6.
48. Compare the number of comparisons used by the insertion sort and the binary insertion sort to sort the list 7, 4, 3, 8, 1, 5, 4, 2.
49. Express the binary insertion sort in pseudocode.
50. a) Devise a variation of the insertion sort that uses a linear search technique that inserts the j-th element in the correct place by first comparing it with the (j - 1)-th element, then the (j - 2)-th element if necessary, and so on.
   b) Use your algorithm to sort 3, 2, 4, 5, 1, 6.
   c) Answer Exercise 45 using this algorithm.
   d) Answer Exercise 46 using this algorithm.
51. When a list of elements is in close to the correct order, would it be better to use an insertion sort or its variation described in Exercise 50?

52. Use the greedy algorithm to make change using quarters, dimes, nickels, and pennies for
   a) 87 cents.   b) 49 cents.   c) 99 cents.   d) 33 cents.
53. Use the greedy algorithm to make change using quarters, dimes, nickels, and pennies for
   a) 51 cents.   b) 69 cents.
   c) 76 cents.   d) 60 cents.
54. Use the greedy algorithm to make change using quarters, dimes, and pennies (but no nickels) for each of the amounts given in Exercise 52. For which of these amounts does the greedy algorithm use the fewest coins of these denominations possible?
55. Use the greedy algorithm to make change using quarters, dimes, and pennies (but no nickels) for each of the amounts given in Exercise 53. For which of these amounts does the greedy algorithm use the fewest coins of these denominations possible?
56. Show that if there were a coin worth 12 cents, the greedy algorithm using quarters, 12-cent coins, dimes, nickels, and pennies would not always produce change using the fewest coins possible.

57. a) Describe a greedy algorithm that solves the problem of scheduling a subset of a collection of proposed talks in a lecture hall by selecting at each step the proposed talk that has the earliest finish time among all those that do not conflict with talks already scheduled. (In Section 3.3 we will show that this greedy algorithm always produces a schedule that includes the largest number of talks possible.)

2.2 The Growth of Functions

INTRODUCTION

In Section 2.1 we discussed the concept of an algorithm. We introduced algorithms that solve a variety of problems, including searching for an element in a list and sorting a list. In Section 2.3 we will study the number of operations used by these algorithms. In particular, we will estimate the number of comparisons used by the linear and binary search algorithms to find an element in a sequence of n elements. We will also estimate the number of comparisons used by the bubble sort and by the insertion sort to sort
a list of $n$ elements. The time required to solve a problem depends on more than only
the number of operations it uses. The time also depends on the hardware and software
used to run the program that implements the algorithm. However, when we change the
hardware and software used to implement an algorithm, we can closely 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.

**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-$O$ of $g(x)$.”)

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 $x > k$.

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,
so $|f(x)| \leq C'|g(x)| \leq C|g(x)|$ whenever $x > k' > k$.

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 $|f(x)| < C|g(x)|$ for $x > k$. This approach is illustrated in Example 1.

**EXAMPLE 1**

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

*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

$$0 < x^2 + 2x + 1 < x^2 + 2x^2 + x^2 = 3x^2$$

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

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

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 with larger values than $x^2$. 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 $f(x)$ 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.

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 2.3. The German mathematician Paul Bachmann first introduced big-$O$ notation in 1892 in an

---

**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.*
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-Omega and big-Theta notations defined later in this section.

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

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

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

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

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 chosen to be as small as possible (sometimes from a set of reference functions, such as functions of the form $x^n$, where $n$ is a positive integer).

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 $O(g(x))$.

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

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.
EXEMPLARY 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 $k = 7$.

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 4,500 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 Petruchie System 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, since 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, since 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.)
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 $O(x^3)$.

**EXAMPLE 3** 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 there are constants $C$ and $k$ such that $x^3 \leq C(7x^2)$ whenever $x > k$. The inequality $x^3 \leq C(7x^2)$ is equivalent to the inequality $x \leq 7C$, which follows by dividing the original inequality by the positive quantity $x^2$. Note that no $C$ exists for which $x \leq 7C$ 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 $O(7x^2)$.

**SOME IMPORTANT BIG-O RESULTS**

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. The following theorem 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 $f(x) = a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0$, 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 35 in Section 1.5), if $x > 1$ we have

$$|f(x)| = |a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0|$$

$$\leq |a_n|x^n + |a_{n-1}|x^{n-1} + \cdots + |a_1|x + |a_0|$$

$$= x^n\left(|a_n| + |a_{n-1}|/x + \cdots + |a_1|/x^{n-1} + |a_0|/x^n\right)$$

$$\leq x^n(|a_n| + |a_{n-1}| + \cdots + |a_1| + |a_0|).$$

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 4** How can big-$O$ notation be used to estimate the sum of the first $n$ positive integers?

**Solution:** Since 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 \(1 + 2 + 3 + \ldots + n\) 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 5 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 5**

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

\[ n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \]

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,

\[ n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \leq n \cdot n \cdot n \cdot \ldots \cdot n = n^n. \]

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 6**

In Section 3.3 we will show that

\[ n < 2^n \]

whenever \(n\) is a positive integer. Using this inequality we can conclude that \(n\) is \(O(2^n)\). (Take \(k = C = 1\) as witnesses.) Since 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 \(O(n)\) since

\[ \log_b n = \frac{\log n}{\log b} < \frac{n}{\log b} \]
whenever \( n \) is a positive integer. We take \( C = 1/\log b \) and \( k = 1 \) as witnesses. (We have used Theorem 3 in Appendix 1 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.

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

\[ |f_1(x)| \leq C_1 |g_1(x)| \]

and

\[ |f_2(x)| \leq C_2 |g_2(x)| \]
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
\[
|f_1 + f_2)(x)| = |f_1(x) + f_2(x)|
\leq |f_1(x)| + |f_2(x)| \quad \text{using triangle inequality } |a + b| \leq |a| + |b|.
\]
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
\[
|f_1(x)| + |f_2(x)| \leq C_1|g_1(x)| + C_2|g_2(x)|
\leq C_1|g(x)| + C_2|g(x)|
= (C_1 + C_2)|g(x)|
= C|g(x)|,
\]
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 \( f_2(x) \) is \( O(g_2(x)) \). Then \( (f_1 + f_2)(x) \) is \( O(\max(|g_1(x)|, |g_2(x)|)) \).

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 \( O(g(x)) \), since \( \max(g(x), g(x)) = g(x) \). 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
\[
|(f_1 f_2)(x)| = |f_1(x)||f_2(x)|
\leq C_1|g_1(x)|C_2|g_2(x)|
\leq C_1C_2(|g_1g_2)(x)|
\leq C|g_1g_2)(x)|,
\]
where \( C = C_1C_2 \). From this inequality, it follows that \( f_1(x)f_2(x) \) is \( O(g_1g_2) \), since there are constants \( C \) and \( k \), namely, \( C = C_1C_2 \) and \( k = \max(k_1, k_2) \), such that \( |(f_1 f_2)(x)| \leq C|g_1(x)g_2(x)| \) 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 \( (f_1 f_2)(x) \) is \( O(g_1(x)g_2(x)) \).
The goal in using big-$O$ notation to estimate functions is to choose a function $g(x)$ that grows relatively slowly so that $f(x)$ is $O(g(x))$. The following examples 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 7**

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 5 we know that $\log(n!) = O(n \log n)$. Using this estimate and the fact that $3n = O(n)$, Theorem 3 gives the estimate that $3n \log(n!) = O(n^2 \log n)$.

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

**Example 8**

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

**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-$\Omega$ 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-$\Theta$ notation. Both big-$\Omega$ and big-$\Theta$ notation were introduced 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.

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.

**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 positive constants $C$ and $k$ such that

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

whenever $x > k$. (This is read as "$f(x)$ is big-$\Omega$ of $g(x)$".)
EXAMPLE 9  

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 since \( f(x) = 8x^3 + 5x^2 + 7 \geq 8x^3 \) 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 \( f(x) = O(g(x)) \) and \( f(x) = \Omega(g(x)) \). 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 \( f(x) = \Theta(g(x)) \) if \( f(x) = O(g(x)) \) and \( f(x) = \Omega(g(x)) \). When \( f(x) = \Theta(g(x)) \), we say that “\( f \) is big-Theta of \( g(x) \)” and we also say that \( f(x) \) is of order \( g(x) \).

When \( f(x) = \Theta(g(x)) \), it is also the case that \( g(x) = \Theta(f(x)) \). Also note that \( f(x) = \Theta(g(x)) \) if and only if \( f(x) = O(g(x)) \) and \( g(x) = O(f(x)) \) (see Exercise 25).

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 10  

We showed (in Example 4) that the sum of the first \( n \) positive integers is \( O(n^2) \). Is this sum of order \( n^2 \)?

**Solution:** Let \( f(n) = 1 + 2 + 3 + \cdots + n \). Since we already know that \( f(n) = O(n^2) \), to show that \( f(n) \) is of order \( n^2 \) we need to find a positive constant \( C \) such that \( f(n) \geq Cn^2 \) 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 \( \lfloor n/2 \rfloor \), we find that

\[
1 + 2 + \cdots + n \geq \lfloor n/2 \rfloor + (\lfloor n/2 \rfloor + 1) + \cdots + n
\]

\[
\geq \lfloor n/2 \rfloor + \lfloor n/2 \rfloor + \cdots + \lfloor n/2 \rfloor
\]

\[
= (n - \lfloor n/2 \rfloor + 1) \lfloor n/2 \rfloor
\]

\[
\geq (n/2)(n/2)
\]

\[
= n^2/4.
\]

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

We can show that \( f(x) = \Theta(g(x)) \) if we can find 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 \( x > k \). This shows that \( f(x) = O(g(x)) \) and \( f(x) = \Omega(g(x)) \).

EXAMPLE 11  

Show that \( 3x^2 + 8x \log x \) is \( \Theta(x^2) \).
Solution: Since $0 \leq 8x \log x \leq 8x^2$, 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 the following theorem, whose proof is left as an exercise at the end of this section.

**Theorem 4** Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, 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 12**

The polynomials $3x^8 + 10x^7 + 221x^2 + 1444$, $x^{19} - 18x^4 - 10112$, and $-x^{99} + 40001x^{96} + 100003x$ 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) $f(x) = [x]$
   f) $f(x) = [x/2]$

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) $f(x) = x^2/2$
   e) $f(x) = 2^x$
   f) $f(x) = [x] \cdot [x]$

3. Use the definition of the fact that $f(x)$ is $O(g(x))$ to show that $x^2 + 9x^3 + 4x + 7$ is $O(x^4)$.

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

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

6. Show that $(x^2 + 2x)/(2x + 1)$ 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) $f(x) = (x^4 + x^3 + 1)/(x^3 + 1)$
   d) $f(x) = x^3 + (\log x)/(x^3 + 1)$

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) $f(x) = x^2 + x^3$
   c) $g(x) = x^3 + x^4$
   d) $g(x) = 3^x$
   e) $g(x) = 2^x$

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 $O(x^5)$ 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 $2^x$ is $O(3^x)$ but that $3^x$ is not $O(2^x)$.

14. Is it true that $x^2$ is $O(g(x))$, if $g$ is the given function? [For example, if $g(x) = x + 1$, this question asks whether $x^3$ is $O(x+1)$.] 
   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) $g(x) = 2^x$

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 $1^k + 2^k + \ldots + n^k$ is $O(n^{k+1})$.

19. Give as good a big-$O$ estimate as possible for each of these functions.
   a) $(n^2 + 8)(n + 1)$
   b) $(n \log n + n^2)(n^2 + 2)$
   c) $(n! + 2^n)(n^3 + \log(n^7 + 1))$
20. 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) $(n^2 + n^2 \log n)(n + 1) + (17 \log n + 19)(n^3 + 2)$
   b) $(2^n + n^2)(n^3 + 3^n)$
   c) $(n^2 + n + 1)(n! + 5^n)$

21. 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) $n^2 + n^2$

22. For each function in Exercise 1, determine whether that function is $\Omega(x^2)$ and whether it is $\Theta(x)$.

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

24. a) Show that $3x + 7$ is $\Theta(x)$.
   b) Show that $2x^2 + x - 7$ is $\Theta(x^2)$.
   c) Show that $|x + 1/2|$ is $\Theta(x)$.
   d) Show that $\log(x^2 + 1)$ is $\Theta(\log x)$.
   e) Show that $\log \log x$ is $\Theta(\log x)$.

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

26. 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 $\Omega(f(x))$.

27. 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 $k, C_1, C_2$ such that $C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$ whenever $x > k$.

28. a) Show that $3x^2 + x + 1$ is $\Theta(3x^2)$ by directly finding the constants $k, C_1, C_2$ in Exercise 27.
   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 $C_1, C_2, k$ are the constants you found in part (a) to show that $3x^2 + x + 1$ is $\Theta(3x^2)$.

29. 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.

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

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

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

33. Show that if $f$ and $g$ are real-valued functions such that $f(x)$ is $O(g(x))$, then $f^b(x)$ is $O(g^b(x))$. [Note that $f^b(x) = (f(x))^b$.]

34. Show that if $f(x)$ is $O(\log_a x)$ where $a > 1$, then $f(x)$ is $O(\log_a x)$ where $a > 1$.

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

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

37. 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 $f_1(x) + f_2(x) = \Theta(g(x))$. Is this still true if $f_1(x)$ and $f_2(x)$ can take negative values?

38. 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 $\Theta(h(x))$.

39. 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 counter-example.

40. 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 $\Theta(g_1 g_2(x))$.

41. 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 $O(f(n))$ simultaneously.

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

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

44. Show that if $f(x) = a_0 x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$, where $a_0, a_1, \ldots, a_{n-1}, a_n$ are real numbers 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 $C, k_1, k_2$ such that $|f(x, y)| \leq C|g(x, y)|$ whenever $x > k_1$ and $y > k_2$.

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

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

47. Show that $x^2 + x y + x \log y$ is $O(x^2 y^2)$.

48. Show that $x^2 y^3 + x^3 y^4 + x^4 y^3$ is $\Omega(x^5 y^5)$.

49. Show that $|xy|$ is $O(xy)$.

50. Show that $|xy|$ is $\Omega(xy)$.

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

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0.$$
52. (Calculus required) Show that if \( f(x) \) and \( g(x) \) are functions such that \( f(x) = o(g(x)) \) and \( c \) is a constant, then \( c f(x) = o(g(x)) \) where \( (c f)(x) = cf(x) \).

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

53. (Calculus required) 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 \( o(x^2) \).

54. (Calculus required) Express the relationship \( f(x) \) is \( o(g(x)) \) using a picture. Show the graphs of \( f(x) \), \( g(x) \), and \( f(x)/g(x) \).

*55. (Calculus required) Suppose that \( f(x) = o(g(x)) \). Does it follow that \( 2^{o(1)} = o(2^{o(1)}) \)?

*56. (Calculus required) Suppose that \( f(x) = o(g(x)) \). Does it follow that \( \log f(x) = o(\log g(x)) \)?

57. (Calculus required) 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 \( f(x) = o(g(x)) \), then \( f(x) = O(g(x)) \).

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

58. (Calculus required) Show that if \( f(x) \) is a polynomial of degree \( n \) and \( g(x) \) is a polynomial of degree \( m \) where \( m > n \), then \( f(x) = o(g(x)) \).

59. (Calculus required) Show that if \( f_1(x) = O(g(x)) \) and \( f_2(x) = O(g(x)) \), then \( f_1(x) + f_2(x) = O(g(x)) \).

60. (Calculus required) Let \( H_n \) be the \( n \)th harmonic number

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

Show that \( H_n = O(\log n) \). (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 \( n \).

61. Show that \( n \log n = O(\log n!) \).

62. Determine whether \( \log(n!) \) is \( \Theta(n \log n) \). Justify your answer.

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

\[
\lim_{x \to \infty} \frac{f(x)}{g(x)} = 1.
\]

63. (Calculus required) 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) \( f(x) = x^3 \log x \), \( g(x) = x^3 \)

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

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

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

f) \( f(x) = 2^{x+3} \), \( g(x) = 2^x \)

g) \( f(x) = 2^x \), \( g(x) = 2^x \)

### 2.3 Complexity of Algorithms

**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 3. 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 obviously 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.

**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 $10^{-9}$ second (1 nanosecond), but personal computers may require $10^{-6}$ second (1 microsecond), 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 2.1, which finds the maximum of a finite set of integers.

**EXAMPLE 1** Describe the time complexity of Algorithm 1 of Section 2.1 for finding the maximum element in a set.

*Solution:* The number of comparisons will be used as the measure of the time complexity of the algorithm, since 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 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 to determine that the end of the list has not been reached and another to determine whether to update the temporary maximum. Since two comparisons are used for each of the second through the $n$th 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.

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

**EXAMPLE 2** Describe the time complexity of the linear search algorithm.

*Solution:* The number of comparisons used by the algorithm will be taken as the measure of the time complexity. At each step of the loop in the algorithm, two comparisons are performed—one to see whether the end of the list has been reached and one to compare the element $x$ with a term of the list. Finally, one more comparison 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 at most \( O(n) \) comparisons.

**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.

*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 \( 2 \lfloor \log n \rfloor + 2 \) comparisons.) Consequently, a binary search requires at most \( \Theta(\log n) \) comparisons. From this analysis it follows that the binary search algorithm is more efficient, in the worst case, than a linear search.

**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 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, assuming that the element \( x \) is in the list.

*Solution:* There are \( n \) types of possible inputs when \( x \) is known to be in the list. If \( x \) is the first term of the list, three comparisons are needed, one to determine whether the end of the list has been reached, one to compare \( x \) and the first term, and one outside the loop.
If $x$ is the second term of the list, two more comparisons are needed, so that a total of five comparisons are used. In general, if $x$ is the $i$th term of the list, 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

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

In Section 3.3 we will show that

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

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 Chapter 4 it has been assumed that $x$ is in the list being searched and it is equally likely that $x$ is in any position. It is also possible to do an average-case analysis of this algorithm when $x$ may not be in the list (see Exercise 13 at the end of this section).

**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 in Example 4 in Section 2.1) 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 $i$th 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 that we will prove in Section 3.3. 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 Example 5 in Section 2.1) inserts the $j$th 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 $j$th 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 $j$th 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 sum of consecutive integers that we will prove in Section 3.3 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 $\Theta(n^2)$.

**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** since 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 $n \log n$ complexity, such as the merge sort, which we will introduce in Chapter 3.

An algorithm has **polynomial complexity** if it has complexity $O(n^b)$, where $b$ is an integer with $b \geq 1$. For example, the bubble sort algorithm is a polynomial-time algorithm because it uses $O(n^2)$ comparisons in the worst case. An algorithm has **exponential complexity** if it has time complexity $O(b^n)$, 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 $O(2^n)$ operations. Finally, an algorithm has **factorial complexity** if it has $O(n!)$ time com-

<table>
<thead>
<tr>
<th><strong>Complexity</strong></th>
<th><strong>Terminology</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant complexity</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>Logarithmic complexity</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear complexity</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>$n \log n$ complexity</td>
</tr>
<tr>
<td>$O(n^b)$</td>
<td>Polynomial complexity</td>
</tr>
<tr>
<td>$O(b^n)$, where $b &gt; 1$</td>
<td>Exponential complexity</td>
</tr>
<tr>
<td>$O(n!)$</td>
<td>Factorial complexity</td>
</tr>
</tbody>
</table>
plexity. The algorithm that finds all orders that a travelling salesman could use to visit \( n \) cities has factorial complexity; we will discuss this algorithm in Chapter 8.

A problem that is solvable using an algorithm with polynomial worst-case complexity is called \textit{tractable}, since 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-\( O \) 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 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 \textit{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 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 \textit{unsolvable} (as opposed to \textit{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. The problem he showed unsolvable is the \textit{halting problem}. This problem takes as its input a program together with input to this program. The problem asks whether the program will halt when executed with the input to the program. We will study the halting problem in Section 3.1. (A biography of Alan Turing and a description of some of his other work can be found in Chapter 11.)

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 \textit{class NP} (tractable problems are said to belong to \textit{class P}). There is also an important class of problems, called \textit{NP-complete problems}, with the property that if any of these problems can be solved by a polynomial worst-case time algorithm, then all can be solved by polynomial worst-case time algorithms.

The satisfiability problem is an example of an NP-complete 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

---

*NP stands for \textit{nondeterministic polynomial} time.*
values requires $\Theta(2^n)$ bit operations where $n$ is the number of variables in the compound proposition.] Furthermore, if a polynomial time algorithm for solving the satisfiability problem 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).

Despite extensive research, no polynomial worst-case time algorithm has been found for any problem in this class. It is generally accepted, although not proven, that no NP-complete problem can be solved in polynomial time. For more information about the complexity of algorithms, consult the references, including [CoLeRi 90], for this section listed at the end of this book.

Note that a big-$O$ estimate of the time complexity of an algorithm expresses how the time required to solve the problem changes 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-$O$ estimates of time complexity cannot be directly translated into the actual amount of computer time used. One reason is that a big-$O$ estimate $f(n)$ is $O(g(n))$, where $f(n)$ is the time complexity of an algorithm and $g(n)$ is a reference function, means that $f(n) \leq Cg(n)$ when $n > k$, where $C$ and $k$ are constants. So without knowing the constants $C$ and $k$ in the inequality, this estimate cannot be used to determine an upper bound on the number of operations used. Moreover, as remarked before, the time required for an operation depends on the type of operation and the computer being used. (Also 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. To provide a lower bound, a big-Theta estimate should be used. However, for simplicity, we will use big-$O$ estimates when describing the time complexity of algorithms, with the understanding that big-Theta estimates would provide more information.)

However, the time required for an algorithm to solve a problem of a specified size can be determined if all operations can be reduced to the bit operations used by the computer. Table 2 displays the time needed to solve problems of various sizes with an algorithm using the indicated number of bit operations. Times of more than $10^{100}$ years are indicated with an asterisk. (In Section 2.5 the number of bit operations used to add and multiply two integers will be discussed.) In the construction of this table, each bit operation is assumed to take $10^{-9}$ second, which is the time required for a bit operation using the fastest computers today. In the future, these times will decrease as faster computers are developed.

It is important to know how long a computer will need to solve a problem. For instance, if an algorithm requires 10 hours, it may be worthwhile to spend the computer time (and money) required to solve this problem. But, if an algorithm requires 10 billion

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Bit Operations Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>10</td>
<td>$3 \times 10^{-9}$</td>
</tr>
<tr>
<td>$10^2$</td>
<td>$7 \times 10^{-8}$</td>
</tr>
<tr>
<td>$10^3$</td>
<td>$1.0 \times 10^{-8}$</td>
</tr>
<tr>
<td>$10^4$</td>
<td>$1.3 \times 10^{-8}$</td>
</tr>
<tr>
<td>$10^5$</td>
<td>$1.7 \times 10^{-8}$</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$2 \times 10^{-8}$</td>
</tr>
</tbody>
</table>
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 5 years ago are now routinely solved, and certainly 5 years from now this statement will still be true.

### Exercises

1. How many comparisons are used by the algorithm given in Exercise 16 of Section 2.1 to find the smallest natural number in a sequence of n natural numbers?

2. Write the algorithm that puts the first four terms of a list of arbitrary length in increasing order. Show that this algorithm has time complexity $O(1)$ in terms of the number of comparisons used.

3. 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?

4. Determine the number of multiplications used to find $x^k$ starting with $x$ and successively squaring (to find $x^2, x^4, \text{and so on}$). Is this a more efficient way to find $x^k$ than by multiplying $x$ by itself the appropriate number of times?

5. 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 2.1).

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

   ```plaintext
   procedure bit count(S: bit string)
   count := 0
   while $S \neq 0$
   begin
     count := count + 1
     $S := S \land (S - 1)$
   end (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 $S \land (S - 1)$ 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 $S$?

7. 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

   ```plaintext
   procedure polynomial(c, a_0, a_1, \ldots, a_n: real numbers)
   power := 1
   y := a_0
   for i := 1 to n
   begin
     power := power * c
     y := y + a_i * power
   end (y = a_n c^n + a_{n-1} c^{n-1} + \cdots + 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.

   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.)

8. 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 $x = c$.

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

9. How large a problem can be solved in 1 second using an algorithm that requires \( f(n) \) bit operations, where each bit operation is carried out in \( 10^{-9} \) second, with these values for \( f(n) ? \)
   a) \( \log n \)  b) \( n \)  c) \( n \log n \)
   d) \( n^2 \)  e) \( 2^n \)  f) \( n! \)

10. How much time does an algorithm take to solve a problem of size \( n \) if this algorithm uses \( 2n^2 + 2^n \) bit operations, each requiring \( 10^{-9} \) second, with these values of \( n ? \)
   a) 10  b) 20  c) 50  d) 100

11. How much time does an algorithm using \( 2^n \) bit operations need if each bit operation takes these amounts of time?
   a) \( 10^{-6} \) second  b) \( 10^{-9} \) second  c) \( 10^{-12} \) second

12. Determine the least number of comparisons, or best-case performance, of an algorithm to find the maximum of a sequence of \( n \) integers, using Algorithm 1 of Section 2.1.
   a) used to locate an element in a list of \( n \) terms with a linear search.
   b) used to locate an element in a list of \( n \) terms using a binary search.

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

14. 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 2.1 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)?

15. Describe the worst-case time complexity, measured in terms of comparisons, of the ternary search algorithm described in Exercise 27 of Section 2.1.

16. Describe the worst-case time complexity, measured in terms of comparisons, of the search algorithm described in Exercise 28 of Section 2.1.

17. Analyze the worst-case time complexity of the algorithm you devised in Exercise 29 of Section 2.1 for locating a mode in a list of nondecreasing integers.

18. Analyze the worst-case time complexity of the algorithm you devised in Exercise 30 of Section 2.1 for locating all modes in a list of nondecreasing integers.

19. Analyze the worst-case time complexity of the algorithm you devised in Exercise 31 of Section 2.1 for finding the first term of a sequence of integers equal to some previous term.

20. Analyze the worst-case time complexity of the algorithm you devised in Exercise 32 of Section 2.1 for finding all terms of a sequence that are greater than the sum of all previous terms.

21. Analyze the worst-case time complexity of the algorithm you devised in Exercise 33 of Section 2.1 for finding the first term of a sequence less than the immediately preceding term.

22. Determine the worst-case complexity in terms of comparisons of the algorithm from Exercise 5 in Section 2.1 for determining all values that occur more than once in a sorted list of integers.

23. Determine the worst-case complexity in terms of comparisons of the algorithm from Exercise 9 in Section 2.1 for determining whether a string is a palindrome.

24. How many comparisons does the selection sort (see preamble to Exercise 41 in Section 2.1) 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.

25. Find 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 47 in Section 2.1.

26. 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.

27. 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

28. 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 41 in Section 2.1)
   d) binary insertion sort (described in the preamble to Exercise 47 in Section 2.1)