Definitions:

A function $f: \mathcal{R} \to \mathcal{R}$ or $f: \mathcal{N} \to \mathcal{R}$ is ***bounded*** if there is a constant k such that $|f(x)| \leq k$ for all x in the domain of f.

For functions $f, g: \mathcal{R} \to \mathcal{R}$ or $f, g: \mathcal{N} \to \mathcal{R}$ (sequences of real numbers) the following are used to compare their growth rates:

- f is ***big-oh*** of g (g ***dominates*** f) if there exist constants C and k such that $|f(x)| \leq C|g(x)|$ for all $x > k$.

  Notation: f is $O(g)$, $f(x) \in O(g(x))$, $f \in O(g)$, $f = O(g)$.

- f is ***little-oh*** of g if $\lim_{x\to\infty} \left|\frac{f(x)}{g(x)}\right| = 0$; i.e., for every $C > 0$ there is a constant k such that $|f(x)| \leq C|g(x)|$ for all $x > k$.

  Notation: f is $o(g)$, $f(x) \in o(g(x))$, $f \in o(g)$, $f = o(g)$.

- f is ***big omega of*** g if there are constants C and k such that $|g(x)| \leq C|f(x)|$ for all $x > k$.

  Notation: f is $\Omega(g)$, $f(x) \in \Omega(g(x))$, $f \in \Omega(g)$, $f = \Omega(g)$.

- f is ***little omega of*** g if $\lim_{x\to\infty} \left|\frac{g(x)}{f(x)}\right| = 0$.

  Notation: f is $\omega(g)$, $f(x) \in \omega(g(x))$, $f \in \omega(g)$, $f = \omega(g)$.

- f is ***theta of*** g if there are positive constants C_1, C_2, and k such that $C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$ for all $x > k$.

  Notation: f is $\Theta(g)$, $f(x) \in \Theta(g(x))$, $f \in \Theta(g)$, $f = \Theta(g)$, $f \approx g$.

- f is ***asymptotic*** to g if $\lim_{x\to\infty} \frac{g(x)}{f(x)} = 1$. This relation is sometimes called ***asymptotic equality***.

  Notation: $f \sim g$, $f(x) \sim g(x)$.

Facts:

1. The notations $O(\ )$, $o(\ )$, $\Omega(\ )$, $\omega(\ )$, and $\Theta(\ )$ all stand for *collections* of functions. Hence the equality sign, as in $f = O(g)$, does not mean equality of functions.

2. The symbols $O(g)$, $o(g)$, $\Omega(g)$, $\omega(g)$, and $\Theta(g)$ are frequently used to represent a typical element of the class of functions it represents, as in an expression such as $f(n) = n \log n + o(n)$.

3. *Growth rates*:

- $O(g)$: the set of functions that grow no more rapidly than a positive multiple of g;
- $o(g)$: the set of functions that grow less rapidly than a positive multiple of g;
- $\Omega(g)$: the set of functions that grow at least as rapidly as a positive multiple of g;
- $\omega(g)$: the set of functions that grow more rapidly than a positive multiple of g;
- $\Theta(g)$: the set of functions that grow at the same rate as a positive multiple of g.

4. Asymptotic notation can be used to describe the growth of infinite sequences, since infinite sequences are functions from $\{0, 1, 2, \ldots\}$ or $\{1, 2, 3, \ldots\}$ to $\mathcal{R}$ (by considering the term a_n as $a(n)$, the value of the function $a(n)$ at the integer n).

5. The big-oh notation was introduced in 1892 by Paul Bachmann (1837–1920) in the study of the rates of growth of various functions in number theory.

6. The big-oh symbol is often called a *Landau symbol*, after Edmund Landau (1877–1938), who popularized this notation.

15. Write an algorithm whose input is a sequence $s_1, \ldots, s_n$ sorted in nondecreasing order and a value x. (Assume that all the values are real numbers.) The algorithm inserts x into the sequence so that the resulting sequence is sorted in nondecreasing order. *Example:* If the input sequence is

    2 6 12 14

    and $x = 5$, the resulting sequence is

    2 5 6 12 14.

16. Modify Algorithm 4.2.1 so that it finds *all* occurrences of p in t.
17. Describe best-case input for Algorithm 4.2.1.
18. Describe worst-case input for Algorithm 4.2.1.
19. Modify Algorithm 4.2.3 so that it sorts the sequence $s_1, \ldots, s_n$ in *nonincreasing* order.
20. The *selection sort* algorithm sorts the sequence $s_1, \ldots, s_n$ in nondecreasing order by first finding the smallest item, say s_i, and placing it first by swapping s_1 and s_i. It then finds the smallest item in $s_2, \ldots, s_n$, again say s_i, and places it second by swapping s_2 and s_i. It continues until the sequence is sorted. Write selection sort in pseudocode.
21. Trace selection sort (see Exercise 20) for the input of Exercises 4–7.
22. Show that the time for selection sort (see Exercise 20) is the same for all inputs of size n.

4.3 → Analysis of Algorithms

From Johnsonbaugh Discrete Math. (book used in cs191 & cs291)

A computer program, even though derived from a correct algorithm, might be useless for certain types of input because the time needed to run the program or the space needed to hold the data, program variables, and so on, is too great. **Analysis of an algorithm** refers to the process of deriving estimates for the time and space needed to execute the algorithm. In this section we deal with the problem of estimating the time required to execute algorithms.

Suppose that we are given a set X of n elements, some labeled "red" and some labeled "black," and we want to find the number of subsets of X that contain at least one red item. Suppose we construct an algorithm that examines all subsets of X and counts those that contain at least one red item and then implement this algorithm as a computer program. Since a set that has n elements has 2^n subsets (see Theorem 2.1.6), the program would require at least 2^n units of time to execute. It does not matter what the units of time are—2^n grows so fast as n increases (see Table 4.3.1) that, except for small values of n, it would be impractical to run the program.

Determining the performance parameters of a computer program is a difficult task and depends on a number of factors such as the computer that is being used, the way

TABLE 4.3.1 ■ Time to execute an algorithm if one step takes 1 microsecond to execute. $\lg n$ denotes $\log_2 n$ (the logarithm of n to base 2).

Number of Steps to Termination for Input of Size n	*Time to Execute if n =*				
	3	6	9	12	
1	10^{-6} sec	10^{-6} sec	10^{-6} sec	10^{-6} sec	
$\lg \lg n$	10^{-6} sec	10^{-6} sec	2×10^{-6} sec	2×10^{-6} sec	
$\lg n$	2×10^{-6} sec	3×10^{-6} sec	3×10^{-6} sec	4×10^{-6} sec	
n	3×10^{-6} sec	6×10^{-6} sec	9×10^{-6} sec	10^{-5} sec	
$n \lg n$	5×10^{-6} sec	2×10^{-5} sec	3×10^{-5} sec	4×10^{-5} sec	
n^2	9×10^{-6} sec	4×10^{-5} sec	8×10^{-5} sec	10^{-4} sec	
n^3	3×10^{-5} sec	2×10^{-4} sec	7×10^{-4} sec	2×10^{-3} sec	
2^n	8×10^{-6} sec	6×10^{-5} sec	5×10^{-4} sec	4×10^{-3} sec	
	50	100	1000	10^5	10^6
1	10^{-6} sec	10^{-6} sec	10^{-6} sec	10^{-6} sec	10^{-6} sec
$\lg \lg n$	2×10^{-6} sec	3×10^{-6} sec	3×10^{-6} sec	4×10^{-6} sec	4×10^{-6} sec
$\lg n$	6×10^{-6} sec	7×10^{-6} sec	10^{-5} sec	2×10^{-5} sec	2×10^{-5} sec
n	5×10^{-5} sec	10^{-4} sec	10^{-3} sec	0.1 sec	1 sec
$n \lg n$	3×10^{-4} sec	7×10^{-4} sec	10^{-2} sec	2 sec	20 sec
n^2	3×10^{-3} sec	0.01 sec	1 sec	3 hr	12 days
n^3	0.13 sec	1 sec	16.7 min	32 yr	31,710 yr
2^n	36 yr	4×10^{16} yr	3×10^{287} yr	3×10^{30089} yr	3×10^{301016} yr

the data are represented, and how the program is translated into machine instructions. Although precise estimates of the execution time of a program must take such factors into account, useful information can be obtained by analyzing the time of the underlying algorithm.

The time needed to execute an algorithm is a function of the input. Usually, it is difficult to obtain an explicit formula for this function, and we settle for less. Instead of dealing directly with the input, we use parameters that characterize the *size* of the input. For example, if the input is a set containing n elements, we would say that the size of the input is n. We can ask for the minimum time needed to execute the algorithm among all inputs of size n. This time is called the **best-case time** for inputs of size n. We can also ask for the maximum time needed to execute the algorithm among all inputs of size n. This time is called the **worst-case time** for inputs of size n. Another important case is **average-case time**—the average time needed to execute the algorithm over some finite set of inputs all of size n.

Since we are primarily concerned with *estimating* the time of an algorithm rather than computing its exact time, as long as we count some fundamental, dominating steps of the algorithm, we will obtain a useful measure of the time. For example, if the principal activity of an algorithm is making comparisons, as might happen in a sorting routine, we might count the number of comparisons. As another example, if an algorithm consists of a single loop whose body executes in at most C steps, for some constant C, we might count the number of iterations of the loop.

Example 4.3.1 ▶

A reasonable definition of the size of input for Algorithm 4.1.2 that finds the largest value in a finite sequence is the number of elements in the input sequence. A reasonable definition of the execution time is the number of iterations of the while loop. With these definitions, the worst-case, best-case, and average-case times for Algorithm 4.1.2 for input of size n are each $n - 1$ since the loop is always executed $n - 1$ times. ◀

Usually we are less interested in the exact best-case or worst-case time required for an algorithm to execute than we are in how the best-case or worst-case time grows as the size of the input increases. For example, suppose that the worst-case time of an algorithm is

$$t(n) = 60n^2 + 5n + 1$$

for input of size n. For large n, the term $60n^2$ is approximately equal to $t(n)$ (see Table 4.3.2). In this sense, $t(n)$ grows like $60n^2$.

If $t(n)$ measures the worst-case time for input of size n in seconds, then

$$T(n) = n^2 + \frac{5}{60}n + \frac{1}{60}$$

measures the worst-case time for input of size n in minutes. Now this change of units does not affect how the worst-case time grows as the size of the input increases but only the units in which we measure the worst-case time for input of size n. Thus when we describe how the best-case or worst-case time grows as the size of the input increases, we not only seek the dominant term [e.g., $60n^2$ in the formula for $t(n)$], but we also may ignore constant coefficients. Under these assumptions, $t(n)$ grows like n^2 as n increases. We say that $t(n)$ is of **order** n^2 and write

$$t(n) = \Theta(n^2),$$

TABLE 4.3.2 ■ Comparing growth of $t(n)$ with $60n^2$.

n	$t(n) = 60n^2 + 5n + 1$	$60n^2$
10	6051	6000
100	600,501	600,000
1000	60,005,001	60,000,000
10,000	6,000,050,001	6,000,000,000

which is read "$t(n)$ is theta of n^2." The basic idea is to replace an expression, such as $t(n) = 60n^2 + 5n + 1$, with a simpler expression, such as n^2, that grows at the same rate as $t(n)$. The formal definitions follow.

Definition 4.3.2 ▶

Let f and g be functions with domain $\{1, 2, 3, \ldots\}$.

We write

$$f(n) = O(g(n))$$

and say that $f(n)$ *is of order at most* $g(n)$ or $f(n)$ *is big oh of* $g(n)$ if there exists a positive constant C_1 such that

$$|f(n)| \leq C_1|g(n)|$$

WWW for all but finitely many positive integers n.

We write

$$f(n) = \Omega(g(n))$$

and say that $f(n)$ *is of order at least* $g(n)$ or $f(n)$ *is omega of* $g(n)$ if there exists a positive constant C_2 such that

$$|f(n)| \geq C_2\,|g(n)|$$

for all but finitely many positive integers n.

We write

$$f(n) = \Theta(g(n))$$

and say that $f(n)$ *is of order* $g(n)$ or $f(n)$ *is theta of* $g(n)$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ◀

Definition 4.3.2 can be loosely paraphrased as follows: $f(n) = O(g(n))$ if, except for a constant factor and a finite number of exceptions, f is bounded above by g. We also say that g is an **asymptotic upper bound** for f. Similarly, $f(n) = \Omega(g(n))$ if, except for a constant factor and a finite number of exceptions, f is bounded below by g. We also say that g is an **asymptotic lower bound** for f. Also, $f(n) = \Theta(g(n))$ if, except for constant factors and a finite number of exceptions, f is bounded above and below by g. We also say that g is an **asymptotic tight bound** for f.

According to Definition 4.3.2, if $f(n) = O(g(n))$, all that we can conclude is that, except for a constant factor and a finite number of exceptions, f is bounded *above* by g, so g grows at least as fast as f. For example, if $f(n) = n$ and $g(n) = 2^n$, then $f(n) = O(g(n))$, but g grows considerably faster than f. The statement $f(n) = O(g(n))$ says nothing about a *lower* bound for f. On the other hand, if $f(n) = \Theta(g(n))$, we can draw the conclusion that, except for constant factors and a finite number of exceptions, f is bounded *above* and *below* by g, so f and g grow at the same rate. Notice that $n = O(2^n)$, but $n \neq \Theta(2^n)$.

Example 4.3.3 ▶

Since

$$60n^2 + 5n + 1 \leq 60n^2 + 5n^2 + n^2 = 66n^2 \qquad \text{for all } n \geq 1,$$

we may take $C_1 = 66$ in Definition 4.3.2 to obtain

$$60n^2 + 5n + 1 = O(n^2).$$

Since

$$60n^2 + 5n + 1 \geq 60n^2 \qquad \text{for all } n \geq 1,$$

we may take $C_2 = 60$ in Definition 4.3.2 to obtain

$$60n^2 + 5n + 1 = \Omega(n^2).$$

Since $60n^2 + 5n + 1 = O(n^2)$ and $60n^2 + 5n + 1 = \Omega(n^2)$,

$$60n^2 + 5n + 1 = \Theta(n^2).$$

◀

The method of Example 4.3.3 can be used to show that a polynomial in n of degree k with nonnegative coefficients is $\Theta(n^k)$. [In fact, *any* polynomial in n of degree k is $\Theta(n^k)$, even if some of its coefficients are negative. To prove this more general result, the method of Example 4.3.3 has to be modified.]

Theorem 4.3.4 *Let*

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

be a polynomial in n of degree k, where each a_i is nonnegative. Then

$$p(n) = \Theta(n^k).$$

Proof We first show that $p(n) = O(n^k)$. Let

$$C_1 = a_k + a_{k-1} + \cdots + a_1 + a_0.$$

Then, for all n,

$$\begin{aligned} p(n) &= a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \\ &\le a_k n^k + a_{k-1} n^k + \cdots + a_1 n^k + a_0 n^k \\ &= (a_k + a_{k-1} + \cdots + a_1 + a_0) n^k = C_1 n^k. \end{aligned}$$

Therefore, $p(n) = O(n^k)$.

Next, we show that $p(n) = \Omega(n^k)$. For all n,

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \ge a_k n^k = C_2 n^k,$$

where $C_2 = a_k$. Therefore, $p(n) = \Omega(n^k)$.

Since $p(n) = O(n^k)$ and $p(n) = \Omega(n^k)$, $p(n) = \Theta(n^k)$.

Example 4.3.5 ▶ In this book, we let $\lg n$ denote $\log_2 n$ (the logarithm of n to the base 2). Since $\lg n < n$ for all $n \ge 1$ (see Figure 4.3.1),

$$2n + 3 \lg n < 2n + 3n = 5n \qquad \text{for all } n \ge 1.$$

Thus,

$$2n + 3 \lg n = O(n).$$

Also,

$$2n + 3 \lg n \ge 2n \qquad \text{for all } n \ge 1.$$

Thus,

$$2n + 3 \lg n = \Omega(n).$$

Therefore,

$$2n + 3 \lg n = \Theta(n).$$

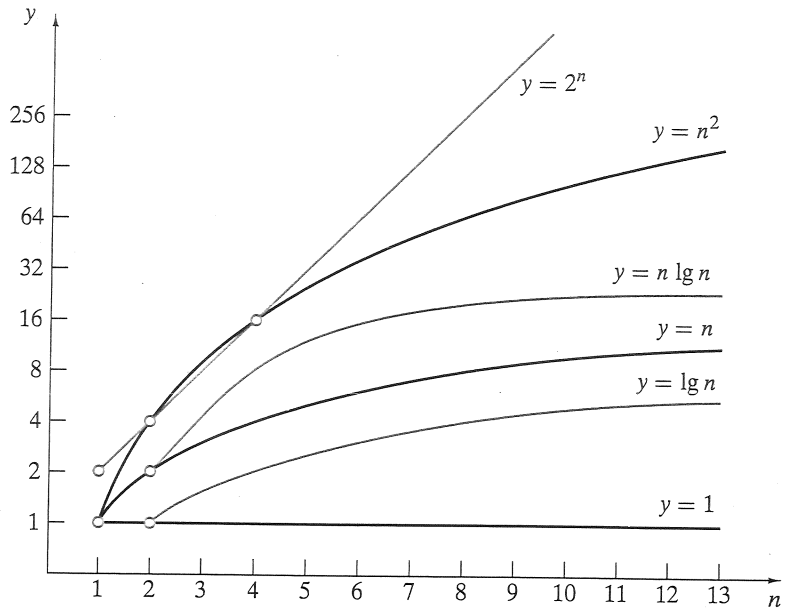


Figure 4.3.1 Growth of some common functions.

Example 4.3.6 ▶ If $a > 1$ and $b > 1$ (to ensure that $\log_b a > 0$), by the change-of-base formula for logarithms [Theorem B.37(e)],

$$\log_b n = \log_b a \log_a n \qquad \text{for all } n \geq 1.$$

Therefore,

$$\log_b n \leq C \log_a n \qquad \text{for all } n \geq 1,$$

where $C = \log_b a$. Thus, $\log_b n = O(\log_a n)$.

Also,

$$\log_b n \geq C \log_a n \qquad \text{for all } n \geq 1;$$

so $\log_b n = \Omega(\log_a n)$. Since $\log_b n = O(\log_a n)$ and $\log_b n = \Omega(\log_a n)$, we conclude that $\log_b n = \Theta(\log_a n)$.

Because $\log_b n = \Theta(\log_a n)$, when using asymptotic notation we need not worry about which number is used as the base for the logarithm function (as long as the base is greater than 1). For this reason, we sometimes simply write log without specifying the base. ◀

Example 4.3.7 ▶ If we replace each integer $1, 2, \ldots, n$ by n in the sum $1 + 2 + \cdots + n$, the sum does not decrease and we have

$$1 + 2 + \cdots + n \leq n + n + \cdots + n = n \cdot n = n^2 \qquad \text{for all } n \geq 1. \qquad \textbf{(4.3.1)}$$

It follows that

$$1 + 2 + \cdots + n = O(n^2).$$

To obtain a lower bound, we might imitate the preceding argument and replace each integer $1, 2, \ldots, n$ by 1 in the sum $1 + 2 + \cdots + n$ to obtain

$$1 + 2 + \cdots + n \geq 1 + 1 + \cdots + 1 = n \qquad \text{for all } n \geq 1.$$

In this case we conclude that

$$1 + 2 + \cdots + n = \Omega(n),$$

and while the preceding expression is true, we cannot deduce a Θ-estimate for $1+2+\cdots+n$, since the upper bound n^2 and lower bound n are not equal. We must be craftier in deriving a lower bound.

One way to get a sharper lower bound is to argue as in the previous paragraph, but first throw away the first half of the terms, to obtain

$$\begin{aligned} 1+2+\cdots+n &\geq \lceil n/2\rceil+\cdots+(n-1)+n \\ &\geq \lceil n/2\rceil+\cdots+\lceil n/2\rceil+\lceil n/2\rceil \\ &= \lceil (n+1)/2\rceil\lceil n/2\rceil \geq (n/2)(n/2) = \frac{n^2}{4} \end{aligned} \qquad \textbf{(4.3.2)}$$

for all $n \geq 1$. We can now conclude that

$$1+2+\cdots+n = \Omega(n^2).$$

Therefore,

$$1+2+\cdots+n = \Theta(n^2). \quad \blacktriangleleft$$

Example 4.3.8 ▶ If k is a positive integer and, as in Example 4.3.7, we replace each integer $1, 2, \ldots, n$ by n, we have

$$1^k+2^k+\cdots+n^k \leq n^k+n^k+\cdots+n^k = n\cdot n^k = n^{k+1}$$

for all $n \geq 1$; hence

$$1^k+2^k+\cdots+n^k = O(n^{k+1}).$$

We can also obtain a lower bound as in Example 4.3.7:

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

for all $n \geq 1$. We conclude that

$$1^k+2^k+\cdots+n^k = \Omega(n^{k+1}),$$

and hence

$$1^k+2^k+\cdots+n^k = \Theta(n^{k+1}). \quad \blacktriangleleft$$

Notice the difference between the polynomial

$$a_k n^k + a_{k-1}n^{k-1}+\cdots+a_1 n + a_0$$

in Theorem 4.3.4 and the expression

$$1^k+2^k+\cdots+n^k$$

in Example 4.3.8. A polynomial has a fixed number of terms, whereas the number of terms in the expression in Example 4.3.8 is dependent on the value of n. Furthermore, the polynomial in Theorem 4.3.4 is $\Theta(n^k)$, but the expression in Example 4.3.8 is $\Theta(n^{k+1})$.

Our next example gives a theta notation for $\lg n!$.

Example 4.3.9 ▶ Using an argument similar to that in Example 4.3.7, we show that

$$\lg n! = \Theta(n \lg n).$$

By properties of logarithms, we have

$$\lg n! = \lg n + \lg(n-1)+\cdots+\lg 2+\lg 1$$

for all $n \geq 1$. Since lg is an increasing function,

$$\lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1 \leq \lg n + \lg n + \cdots + \lg n + \lg n = n \lg n$$

for all $n \geq 1$. We conclude that

$$\lg n! = O(n \lg n).$$

For all $n \geq 4$, we have

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

[since $(\lg n)/2 \geq 1$ for all $n \geq 4$]. Therefore,

$$\lg n! = \Omega(n \lg n).$$

It follows that

$$\lg n! = \Theta(n \lg n).$$ ◀

Example 4.3.10 ▶ Show that if $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.

Because $f(n) = \Theta(g(n))$, there are constants C_1 and C_2 such that

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

for all but finitely many positive integers n. Because $g(n) = \Theta(h(n))$, there are constants C_3 and C_4 such that

$$C_3|h(n)| \leq |g(n)| \leq C_4|h(n)|$$

for all but finitely many positive integers n. Therefore,

$$C_1C_3|h(n)| \leq C_1|g(n)| \leq |f(n)| \leq C_2|g(n)| \leq C_2C_4|h(n)|$$

for all but finitely many positive integers n. It follows that $f(n) = \Theta(h(n))$. ◀

We next define what it means for the best-case, worst-case, or average-case time of an algorithm to be of order at most $g(n)$.

Definition 4.3.11 ▶ If an algorithm requires $t(n)$ units of time to terminate in the best case for an input of size n and

$$t(n) = O(g(n)),$$

we say that the *best-case time required by the algorithm is of order at most* $g(n)$ or that the *best-case time required by the algorithm* is $O(g(n))$.

If an algorithm requires $t(n)$ units of time to terminate in the worst case for an input of size n and

$$t(n) = O(g(n)),$$

we say that the *worst-case time required by the algorithm is of order at most* $g(n)$ or that the *worst-case time required by the algorithm* is $O(g(n))$.

If an algorithm requires $t(n)$ units of time to terminate in the average case for an input of size n and

$$t(n) = O(g(n)),$$

we say that the *average-case time required by the algorithm is of order at most* $g(n)$ or that the *average-case time required by the algorithm* is $O(g(n))$. ◀

By replacing O by Ω and "at most" by "at least" in Definition 4.3.11, we obtain the definition of what it means for the best-case, worst-case, or average-case time of an algorithm to be of order at least $g(n)$. If the best-case time required by an algorithm is $O(g(n))$ and $\Omega(g(n))$, we say that the best-case time required by the algorithm is $\Theta(g(n))$. An analogous definition applies to the worst-case and average-case times of an algorithm.

Example 4.3.12 ▶ Suppose that an algorithm is known to take

$$60n^2 + 5n + 1$$

units of time to terminate in the worst case for inputs of size n. We showed in Example 4.3.3 that

$$60n^2 + 5n + 1 = \Theta(n^2).$$

Thus the worst-case time required by this algorithm is $\Theta(n^2)$. ◀

Example 4.3.13 ▶ Find a theta notation in terms of n for the number of times the statement $x = x + 1$ is executed.

```
1.  for i = 1 to n
2.      for j = 1 to i
3.          x = x + 1
```

First, i is set to 1 and, as j runs from 1 to 1, line 3 is executed one time. Next, i is set to 2 and, as j runs from 1 to 2, line 3 is executed two times, and so on. Thus the total number of times line 3 is executed is (see Example 4.3.7)

$$1 + 2 + \cdots + n = \Theta(n^2).$$

Thus a theta notation for the number of times the statement $x = x + 1$ is executed is $\Theta(n^2)$. ◀

Example 4.3.14 ▶ Find a theta notation in terms of n for the number of times the statement $x = x + 1$ is executed:

```
1.  i = n
2.  while (i ≥ 1) {
3.      x = x + 1
4.      i = ⌊i/2⌋
5.  }
```

First, we examine some specific cases. Because of the floor function, the computations are simplified if n is a power of 2. Consider, for example, the case $n = 8$. At line 1, i is set to 8. At line 2, the condition $i \geq 1$ is true. At line 3, we execute the statement $x = x + 1$ the first time. At line 4, i is reset to 4 and we return to line 2.

At line 2, the condition $i \geq 1$ is again true. At line 3, we execute the statement $x = x + 1$ the second time. At line 4, i is reset to 2 and we return to line 2.

At line 2, the condition $i \geq 1$ is again true. At line 3, we execute the statement $x = x + 1$ the third time. At line 4, i is reset to 1 and we return to line 2.

At line 2, the condition $i \geq 1$ is again true. At line 3, we execute the statement $x = x + 1$ the fourth time. At line 4, i is reset to 0 and we return to line 2.

This time at line 2, the condition $i \geq 1$ is false. The statement $x = x + 1$ was executed four times.

Now suppose that n is 16. At line 1, i is set to 16. At line 2, the condition $i \geq 1$ is true. At line 3, we execute the statement $x = x + 1$ the first time. At line 4, i is reset to 8 and we return to line 2. Now execution proceeds as before; the statement $x = x + 1$ is executed four more times, for a total of five times.

Similarly, if n is 32, the statement $x = x + 1$ is executed a total of six times.

A pattern is emerging. Each time the initial value of n is doubled, the statement $x = x + 1$ is executed one more time. More precisely, if $n = 2^k$, the statement $x = x + 1$ is executed $k + 1$ times. Since k is the exponent for 2, $k = \lg n$. Thus if $n = 2^k$, the statement $x = x + 1$ is executed $1 + \lg n$ times.

If n is an arbitrary positive integer (not necessarily a power of 2), it lies between two powers of 2; that is, for some $k \geq 1$,

$$2^{k-1} \leq n < 2^k.$$

We use induction on k to show that in this case the statement $x = x + 1$ is executed k times.

If $k = 1$, we have

$$1 = 2^{1-1} \leq n < 2^1 = 2.$$

Therefore, n is 1. In this case, the statement $x = x + 1$ is executed once. Thus the Basis Step is proved.

Now suppose that if n satisfies

$$2^{k-1} \leq n < 2^k,$$

the statement $x = x + 1$ is executed k times. We must show that if n satisfies

$$2^k \leq n < 2^{k+1},$$

the statement $x = x + 1$ is executed $k + 1$ times.

Suppose that n satisfies

$$2^k \leq n < 2^{k+1}.$$

At line 1, i is set to n. At line 2, the condition $i \geq 1$ is true. At line 3, we execute the statement $x = x + 1$ the first time. At line 4, i is reset to $\lfloor n/2 \rfloor$ and we return to line 2. Notice that

$$2^{k-1} \leq n/2 < 2^k.$$

Because 2^{k-1} is an integer, we must also have

$$2^{k-1} \leq \lfloor n/2 \rfloor < 2^k.$$

By the inductive assumption, the statement $x = x + 1$ is executed k more times, for a total of $k + 1$ times. The Inductive Step is complete. Therefore, if n satisfies

$$2^{k-1} \leq n < 2^k,$$

the statement $x = x + 1$ is executed k times.

Suppose that n satisfies

$$2^{k-1} \leq n < 2^k.$$

Taking logarithms to the base 2, we have

$$k - 1 \leq \lg n < k.$$

Therefore, k, the number of times the statement $x = x + 1$ is executed, satisfies

$$\lg n < k \leq 1 + \lg n.$$

Because k is an integer, we must have

$$k \leq 1 + \lfloor \lg n \rfloor.$$

Furthermore,

$$\lfloor \lg n \rfloor < k.$$

It follows from the last two inequalities that

$$k = 1 + \lfloor \lg n \rfloor.$$

Since

$$1 + \lfloor \lg n \rfloor = \Theta(\lg n),$$

a theta notation for the number of times the statement $x = x + 1$ is executed is $\Theta(\lg n)$. ◀

Many algorithms are based on the idea of repeated halving. Example 4.3.14 shows that for size n, repeated halving takes time $\Theta(\lg n)$. Of course, the algorithm may do work in addition to the halving that will increase the overall time.

Example 4.3.15 ▶ Find a theta notation in terms of n for the number of times the statement $x = x + 1$ is executed.

```
1.  j = n
2.  while (j ≥ 1) {
3.      for i = 1 to j
4.          x = x + 1
5.      j = ⌊j/2⌋
6.  }
```

Let $t(n)$ denote the number of times we execute the statement $x = x + 1$. The first time we arrive at the body of the while loop, the statement $x = x + 1$ is executed n times. Therefore $t(n) \geq n$ for all $n \geq 1$ and $t(n) = \Omega(n)$.

Next we derive a big oh notation for $t(n)$. After j is set to n, we arrive at the while loop for the first time. The statement $x = x + 1$ is executed n times. At line 5, j is replaced by $\lfloor n/2 \rfloor$; hence $j \leq n/2$. If $j \geq 1$, we will execute $x = x + 1$ at most $n/2$ additional times in the next iteration of the while loop, and so on. If we let k denote the number of times we execute the body of the while loop, the number of times we execute $x = x + 1$ is at most

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{k-1}}.$$

This geometric sum (see Example 1.7.4) is equal to

$$\frac{n\left(1 - \frac{1}{2^k}\right)}{1 - \frac{1}{2}}.$$

Now

$$t(n) \leq \frac{n\left(1 - \frac{1}{2^k}\right)}{1 - \frac{1}{2}} = 2n\left(1 - \frac{1}{2^k}\right) \leq 2n \qquad \text{for all } n \geq 1,$$

so $t(n) = O(n)$. Thus a theta notation for the number of times we execute $x = x + 1$ is $\Theta(n)$. ◀

Example 4.3.16 ▶ Determine, in theta notation, the best-case, worst-case, and average-case times required to execute Algorithm 4.3.17, which follows. Assume that the input size is n and that the run time of the algorithm is the number of comparisons made at line 3. Also, assume that the $n + 1$ possibilities of *key* being at any particular position in the sequence or not being in the sequence are equally likely.

The best-case time can be analyzed as follows. If $s_1 = key$, line 3 is executed once. Thus the best-case time of Algorithm 4.3.17 is

$$\Theta(1).$$

The worst-case time of Algorithm 4.3.17 is analyzed as follows. If *key* is not in the sequence, line 3 will be executed n times, so the worst-case time of Algorithm 4.3.17 is

$$\Theta(n).$$

Finally, consider the average-case time of Algorithm 4.3.17. If *key* is found at the ith position, line 3 is executed i times; if *key* is not in the sequence, line 3 is executed n times. Thus the average number of times line 3 is executed is

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

Now

$$\begin{aligned} \frac{(1 + 2 + \cdots + n) + n}{n + 1} &\leq \frac{n^2 + n}{n + 1} && \text{by (4.3.1)} \\ &= \frac{n(n + 1)}{n + 1} = n. \end{aligned}$$

Therefore, the average-case time of Algorithm 4.3.17 is

$$O(n).$$

Also,

$$\begin{aligned} \frac{(1 + 2 + \cdots + n) + n}{n + 1} &\geq \frac{n^2/4 + n}{n + 1} && \text{by (4.3.2)} \\ &\geq \frac{n^2/4 + n/4}{n + 1} = \frac{n}{4}. \end{aligned}$$

Therefore the average-case time of Algorithm 4.3.17 is

$$\Omega(n).$$

Thus the average-case time of Algorithm 4.3.17 is

$$\Theta(n).$$

For this algorithm, the worst-case and average-case times are both $\Theta(n)$. ◀

Algorithm 4.3.17 **Searching an Unordered Sequence**

Given the sequence $s_1, \ldots, s_n$ and a value *key*, this algorithm returns the index of *key*. If *key* is not found, the algorithm returns 0.

Input: $s_1, s_2, \ldots, s_n, n$, and *key* (the value to search for)
Output: The index of *key*, or if *key* is not found, 0

```
1.  linear_search(s, n, key) {
2.      for i = 1 to n
3.          if (key == s_i)
4.              return i // successful search
5.      return 0 // unsuccessful search
6.  }
```

The constants that are suppressed in the theta notation may be important. Even if for any input of size n, algorithm A requires exactly $C_1 n$ time units and algorithm B requires exactly $C_2 n^2$ time units, for certain sizes of inputs algorithm B may be superior. For example, suppose that for any input of size n, algorithm A requires $300n$ units of time and algorithm B requires $5n^2$ units of time. For an input size of $n = 5$, algorithm A requires 1500 units of time and algorithm B requires 125 units of time, and thus algorithm B is faster. Of course, for sufficiently large inputs, algorithm A is considerably faster than algorithm B.

TABLE 4.3.3 ■ Common growth functions.

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n \lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k), k \geq 1$	Polynomial
$\Theta(c^n), c > 1$	Exponential
$\Theta(n!)$	Factorial

Certain growth functions occur so often that they are given special names, as shown in Table 4.3.3. The functions in Table 4.3.3, with the exception of $\Theta(n^k)$, are arranged so that if $\Theta(f(n))$ is above $\Theta(g(n))$, then $f(n) \leq g(n)$ for all but finitely many positive integers n. Thus, if algorithms A and B have run times that are $\Theta(f(n))$ and $\Theta(g(n))$, respectively, and $\Theta(f(n))$ is above $\Theta(g(n))$ in Table 4.3.3, then algorithm A is more time-efficient than algorithm B for sufficiently large inputs.

It is important to develop some feeling for the relative sizes of the functions in Table 4.3.3. In Figure 4.3.1 we have graphed some of these functions. Another way to develop some appreciation for the relative sizes of the functions $f(n)$ in Table 4.3.3 is to determine how long it would take an algorithm to terminate whose run time is exactly $f(n)$. For this purpose, let us assume that we have a computer that can execute one step in 1 microsecond (10^{-6} sec). Table 4.3.1 shows the execution times, under this assumption, for various input sizes. Notice that it is practical to implement an algorithm that requires 2^n steps for an input of size n only for very small input sizes. Algorithms requiring n^2 or n^3 steps also become impractical to implement, but for relatively larger input sizes. Also, notice the dramatic improvement that results when we move from n^2 steps to $n \lg n$ steps.

A problem that has a worst-case polynomial-time algorithm is considered to have a "good" algorithm; the interpretation is that such a problem has an efficient solution. Such problems are called **feasible** or **tractable.** Of course, if the worst-case time to solve the problem is proportional to a high-degree polynomial, the problem can still take a long time to solve. Fortunately, in many important cases, the polynomial bound has small degree.

A problem that does not have a worst-case polynomial-time algorithm is said to be **intractable.** Any algorithm, if there is one, that solves an intractable problem is guaranteed to take a long time to execute in the worst case, even for modest sizes of the input.

Certain problems are so hard that they have no algorithms at all. A problem for which there is no algorithm is said to be **unsolvable.** A large number of problems are known to be unsolvable, some of considerable practical importance. One of the earliest problems to be proved unsolvable is the **halting problem:** Given an arbitrary program and a set of inputs, will the program eventually halt?

A large number of solvable problems have an as yet undetermined status; they are thought to be intractable, but none of them has been proved to be intractable. (Most of these problems belong to the class of NP-complete problems; see [Johnsonbaugh] for details.) An example of an NP-complete problem is:

> Given a collection $\mathcal{C}$ of finite sets and a positive integer $k \leq |\mathcal{C}|$, does $\mathcal{C}$ contain at least k mutually disjoint sets?

Other NP-complete problems include the traveling-salesperson problem and the Hamiltonian-cycle problem (see Section 8.3).

Problem-Solving Tips

To derive a big oh notation for an expression $f(n)$ directly, you must find a constant C_1 and a simple expression $g(n)$ (e.g., n, $n \lg n$, n^2) such that $|f(n)| \leq C_1|g(n)|$ for all but finitely many n. Remember you're trying to derive an *inequality, not* an equality, so you can replace terms in $f(n)$ with other terms if the result is *larger* (see, e.g., Example 4.3.3).

To derive an omega notation for an expression $f(n)$ directly, you must find a constant C_2 and a simple expression $g(n)$ such that $|f(n)| \geq C_2|g(n)|$ for all but finitely

many n. Again, you're trying to derive an *inequality* so you can replace terms in $f(n)$ with other terms if the result is *smaller* (again, see Example 4.3.3).

To derive a theta notation, you must derive both big oh and omega notations.

Another way to derive big oh, omega, and theta estimates is to use known results:

Expression	*Name*	*Estimate*	*Reference*
$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$	Polynomial	$\Theta(n^k)$	Theorem 4.3.4
$1 + 2 + \cdots + n$	Arithmetic Sum (Case $k = 1$ for Next Entry)	$\Theta(n^2)$	Example 4.3.7
$1^k + 2^k + \cdots + n^k$	Sum of Powers	$\Theta(n^{k+1})$	Example 4.3.8
$\lg n!$	log n Factorial	$\Theta(n \lg n)$	Example 4.3.9

To derive an asymptotic estimate for the time of an algorithm, count the number of steps $t(n)$ required by the algorithm, and then derive an estimate for $t(n)$ as described previously. Algorithms typically contain loops, in which case, deriving $t(n)$ requires counting the number of iterations of the loops.

Section Review Exercises

1. To what does "analysis of algorithms" refer?
2. What is the worst-case time of an algorithm?
3. What is the best-case time of an algorithm?
4. What is the average-case time of an algorithm?
5. Define $f(n) = O(g(n))$. What is this notation called?
6. Give an intuitive interpretation of how f and g are related if $f(n) = O(g(n))$.
7. Define $f(n) = \Omega(g(n))$. What is this notation called?
8. Give an intuitive interpretation of how f and g are related if $f(n) = \Omega(g(n))$.
9. Define $f(n) = \Theta(g(n))$. What is this notation called?
10. Give an intuitive interpretation of how f and g are related if $f(n) = \Theta(g(n))$.

Exercises

Select a theta notation from Table 4.3.3 for each expression in Exercises 1–12.

1. $6n + 1$
2. $2n^2 + 1$
3. $6n^3 + 12n^2 + 1$
4. $3n^2 + 2n \lg n$
5. $2 \lg n + 4n + 3n \lg n$
6. $6n^6 + n + 4$
7. $2 + 4 + 6 + \cdots + 2n$
8. $(6n + 1)^2$
9. $(6n + 4)(1 + \lg n)$
10. $\dfrac{(n+1)(n+3)}{n+2}$
11. $\dfrac{(n^2 + \lg n)(n+1)}{n+n^2}$
12. $2 + 4 + 8 + 16 + \cdots + 2^n$

In Exercises 13–15, select a theta notation for $f(n) + g(n)$.

13. $f(n) = \Theta(1), \quad g(n) = \Theta(n^2)$
14. $f(n) = 6n^3 + 2n^2 + 4, \quad g(n) = \Theta(n \lg n)$
15. $f(n) = \Theta(n^{3/2}), \quad g(n) = \Theta(n^{5/2})$

In Exercises 16–25, select a theta notation from among

$$\Theta(1), \quad \Theta(\lg n), \quad \Theta(n), \quad \Theta(n \lg n),$$
$$\Theta(n^2), \quad \Theta(n^3), \quad \Theta(2^n), \quad \textit{or} \quad \Theta(n!)$$

for the number of times the statement $x = x + 1$ is executed.

16.
```
for i = 1 to 2n
    x = x + 1
```

17.
```
i = 1
while (i ≤ 2n) {
    x = x + 1
    i = i + 2
}
```

18.
```
for i = 1 to n
    for j = 1 to n
        x = x + 1
```

19.
```
for i = 1 to 2n
    for j = 1 to n
        x = x + 1
```

20.
```
for i = 1 to n
    for j = 1 to ⌊i/2⌋
        x = x + 1
```

21.
```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            x = x + 1
```

22.
```
for i = 1 to n
    for j = 1 to n
        for k = 1 to i
            x = x + 1
```

23.
```
for i = 1 to n
    for j = 1 to i
        for k = 1 to j
            x = x + 1
```

24.
```
j = n
while (j ≥ 1) {
    for i = 1 to j
        x = x + 1
    j = ⌊j/3⌋
}
```

25.
```
i = n
while (i ≥ 1) {
    for j = 1 to n
        x = x + 1
    i = ⌊i/2⌋
}
```

26. Find a theta notation for the number of times the statement $x = x + 1$ is executed.

```
i = 2
while (i < n) {
    i = i^2
    x = x + 1
}
```

27. Let $t(n)$ be the total number of times that i is incremented and j is decremented in the following pseudocode, where $a_1, a_2, \ldots$ is a sequence of real numbers.

```
i = 1
j = n
while (i < j) {
    while (i < j ∧ a_i < 0)
        i = i + 1
    while (i < j ∧ a_j ≥ 0)
        j = j − 1
    if (i < j)
        swap(a_i, a_j)
}
```

Find a theta notation for $t(n)$.

28. Find a theta notation for the worst-case time required by the following algorithm:

```
iskey(s, n, key) {
    for i = 1 to n − 1
        for j = i + 1 to n
            if (s_i + s_j == key)
                return 1
            else
                return 0
}
```

29. In addition to finding a theta notation in Exercises 1–28, prove that it is correct.

30. Find the exact number of comparisons (lines 10, 15, 17, 24, and 26) required by the following algorithm when n is even and when n is odd. Find a theta notation for this algorithm.

Input: $s_1, s_2, \ldots, s_n, n$

Output: *large* (the largest item in $s_1, s_2, \ldots, s_n$), *small* (the smallest item in $s_1, s_2, \ldots, s_n$)

```
1.  large_small(s, n, large, small) {
2.      if (n == 1) {
3.          large = s_1
4.          small = s_1
5.          return
6.      }
7.      m = 2⌊n/2⌋
8.      i = 1
9.      while (i ≤ m − 1) {
10.         if (s_i > s_{i+1})
11.             swap(s_i, s_{i+1})
12.         i = i + 2
13.     }
14.     if (n > m) {
15.         if (s_{m−1} > s_n)
16.             swap(s_{m−1}, s_n)
17.         if (s_n > s_m)
18.             swap(s_m, s_n)
19.     }
20.     small = s_1
21.     large = s_2
22.     i = 3
23.     while (i ≤ m − 1) {
24.         if (s_i < small)
25.             small = s_i
26.         if (s_{i+1} > large)
27.             large = s_{i+1}
28.         i = i + 2
29.     }
30. }
```

31. This exercise shows another way to guess a formula for $1 + 2 + \cdots + n$.

Example 4.3.7 suggests that

$$1 + 2 + \cdots + n = An^2 + Bn + C \qquad \text{for all } n,$$

for some constants A, B, and C. Assuming that this is true, plug in $n = 1, 2, 3$ to obtain three equations in the three unknowns A, B, and C. Now solve for A, B, and C. The resulting formula can now be proved using mathematical induction (see Section 1.7).

32. Suppose that $a > 1$ and that $f(n) = \Theta(\log_a n)$. Show that $f(n) = \Theta(\lg n)$.

33. Show that $n! = O(n^n)$.

34. Show that $2^n = O(n!)$.

35. By using an argument like the one shown in Examples 4.3.7–4.3.9 or otherwise, prove that $\sum_{i=1}^{n} i \lg i = \Theta(n^2 \lg n)$.

★ 36. Show that $n^{n+1} = O(2^{n^2})$.

37. Show that $\lg(n^k + c) = \Theta(\lg n)$ for every fixed $k > 0$ and $c > 0$.

38. Show that if n is a power of 2, say $n = 2^k$, then

$$\sum_{i=0}^{k} \lg(n/2^i) = \Theta(\lg^2 n).$$

39. Suppose that $f(n) = O(g(n))$, and $f(n) \geq 0$ and $g(n) > 0$ for all $n \geq 1$. Show that for some constant C, $f(n) \leq Cg(n)$ for all $n \geq 1$.

40. State and prove a result for Ω similar to that for Exercise 39.

41. State and prove a result for Θ similar to that for Exercises 39 and 40.

Determine whether each statement in Exercises 42–52 is true or false. If the statement is false, give a counterexample. Assume that the functions f, g, and h take on only positive values.

42. $n^n = O(2^n)$

43. $2 + \sin n = O(2 + \cos n)$

44. If $f(n) = \Theta(h(n))$ and $g(n) = \Theta(h(n))$, then $f(n) + g(n) = \Theta(h(n))$.

45. If $f(n) = \Theta(g(n))$, then $cf(n) = \Theta(g(n))$ for any $c \neq 0$.

46. If $f(n) = \Theta(g(n))$, then $2^{f(n)} = \Theta(2^{g(n)})$.

47. If $f(n) = \Theta(g(n))$, then $\lg f(n) = \Theta(\lg g(n))$. Assume that $f(n) \geq 1$ and $g(n) \geq 1$ for all $n = 1, 2, \ldots$.

48. If $f(n) = O(g(n))$, then $g(n) = O(f(n))$.

49. If $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$.

50. If $f(n) = \Theta(g(n))$, then $g(n) = \Theta(f(n))$.

51. $f(n) + g(n) = \Theta(h(n))$, where $h(n) = \max\{f(n), g(n)\}$

52. $f(n) + g(n) = \Theta(h(n))$, where $h(n) = \min\{f(n), g(n)\}$

53. Write out exactly what $f(n) \neq O(g(n))$ means.

54. What is wrong with the following argument that purports to show that we cannot simultaneously have $f(n) \neq O(g(n))$ and $g(n) \neq O(f(n))$?

If $f(n) \neq O(g(n))$, then for every $C > 0$, $|f(n)| > C|g(n)|$. In particular, $|f(n)| > 2|g(n)|$. If $g(n) \neq O(f(n))$, then for every $C > 0$, $|g(n)| > C|f(n)|$. In particular, $|g(n)| > 2|f(n)|$. But now

$$|f(n)| > 2|g(n)| > 4|f(n)|.$$

Cancelling $|f(n)|$ gives $1 > 4$, which is a contradiction. Therefore, we cannot simultaneously have $f(n) \neq O(g(n))$ and $g(n) \neq O(f(n))$.

★ **55.** Find functions f and g satisfying

$$f(n) \neq O(g(n)) \qquad \text{and} \qquad g(n) \neq O(f(n)).$$

★ **56.** Give an example of increasing positive functions f and g defined on the positive integers for which

$$f(n) \neq O(g(n)) \qquad \text{and} \qquad g(n) \neq O(f(n)).$$

★ **57.** Prove that $n^k = O(c^n)$ for all $k = 1, 2, \ldots$ and $c > 1$.

58. Find functions f, g, h, and t satisfying

$$f(n) = \Theta(g(n)), \qquad h(n) = \Theta(t(n)),$$
$$f(n) - h(n) \neq \Theta(g(n) - t(n)).$$

59. Suppose that the worst-case time of an algorithm is $\Theta(n)$. What is the error in the following reasoning? Since $2n = \Theta(n)$, the worst-case time to run the algorithm with input of size $2n$ will be approximately the same as the worst-case time to run the algorithm with input of size n.

60. Does

$$f(n) = O(g(n))$$

define an equivalence relation on the set of real-valued functions on $\{1, 2, \ldots\}$?

61. Does

$$f(n) = \Theta(g(n))$$

define an equivalence relation on the set of real-valued functions on $\{1, 2, \ldots\}$?

62. [Requires the integral]

(a) Show, by consulting the figure, that

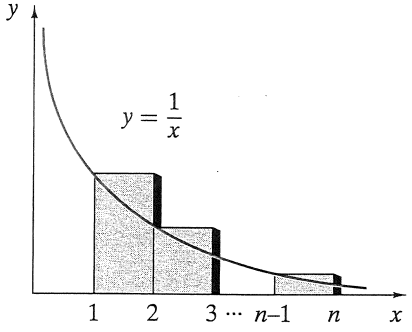
$$\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} < \log_e n.$$

(b) Show, by consulting the figure, that

$$\log_e n < 1 + \frac{1}{2} + \cdots + \frac{1}{n-1}.$$

(c) Use parts (a) and (b) to show that

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



63. [Requires the integral] Use an argument like the one shown in Exercise 62 to show that

$$\frac{n^{m+1}}{m+1} < 1^m + 2^m + \cdots + n^m < \frac{(n+1)^{m+1}}{m+1},$$

where m is a positive integer.

64. By using the formula

$$\frac{b^{n+1} - a^{n+1}}{b-a} = \sum_{i=0}^{n} a^i b^{n-i} \qquad 0 \leq a < b$$

or otherwise, prove that

$$\frac{b^{n+1} - a^{n+1}}{b-a} < (n+1)b^n \qquad 0 \leq a < b.$$

65. Take $a = 1 + 1/(n+1)$ and $b = 1 + 1/n$ in the inequality of Exercise 64 to prove that the sequence $\{(1+1/n)^n\}$ is increasing.

66. Take $a = 1$ and $b = 1 + 1/(2n)$ in the inequality of Exercise 64 to prove that

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

for all $n \geq 1$. Use the preceding exercise to conclude that

$$\left(1 + \frac{1}{n}\right)^n < 4$$

for all $n \geq 1$.

The method used to prove the results of this exercise and its predecessor is apparently due to Fort in 1862 (see [Chrystal, vol. II, page 77]).

67. By using the preceding two exercises or otherwise, prove that

$$\frac{1}{n} \leq \lg(n+1) - \lg n < \frac{2}{n}$$

for all $n \geq 1$.

68. Use the preceding exercise to prove that

$$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\lg n).$$

(Compare with Exercise 62.)

69. What is wrong with the following "proof" that any algorithm has a run time that is $O(n)$?

We must show that the time required for an input of size n is at most a constant times n.

Basis Step

Suppose that $n = 1$. If the algorithm takes C units of time for an input of size 1, the algorithm takes at most $C \cdot 1$ units of time. Thus the assertion is true for $n = 1$.

Inductive Step

Assume that the time required for an input of size n is at most $C'n$ and that the time for processing an additional item is C''. Let C be the maximum of C' and C''. Then the total time required for an input of size $n+1$ is at most

$$C'n + C'' \leq Cn + C = C(n+1).$$

The Inductive Step has been verified.

By induction, for input of size n, the time required is at most a constant time n. Therefore, the run time is $O(n)$.

In Exercises 70–75, determine whether the statement is true or false. If the statement is true, prove it. If the statement is false, give a counterexample. Assume that f and g are real-valued functions defined on the set of positive integers and that $g(n) \neq 0$ for $n \geq 1$. These exercises require calculus.

70. If
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0,$$
then $f(n) = O(g(n))$.

71. If
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0,$$
then $f(n) = \Theta(g(n))$.

72. If
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \neq 0,$$
then $f(n) = O(g(n))$.

73. If
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \neq 0,$$
then $f(n) = \Theta(g(n))$.

74. If $f(n) = O(g(n))$, then
$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$
exists and is equal to some real number.

75. If $f(n) = \Theta(g(n))$, then
$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$
exists and is equal to some real number.

★ 76. Use induction to prove that
$$\lg n! \geq \frac{n}{2} \lg \frac{n}{2}.$$

77. [Requires calculus] Let $\ln x$ denote the natural logarithm ($\log_e x$) of x. Use the integral to obtain the estimate
$$n \ln n - n \leq \sum_{k=1}^{n} \ln k = \ln n!, \qquad n \geq 1.$$

78. Use the result of Exercise 77 and the change-of-base formula for logarithms to obtain the formula
$$n \lg n - n \lg e \leq \lg n!, \qquad n \geq 1.$$

79. Deduce
$$\lg n! \geq \frac{n}{2} \lg \frac{n}{2}$$
from the inequality of Exercise 78.

Problem-Solving Corner: Design and Analysis of an Algorithm

Problem

Develop and analyze an algorithm that returns the maximum sum of consecutive values in the numerical sequence
$$s_1, \ldots, s_n.$$
In mathematical notation, the problem is to find the maximum sum of the form $s_j + s_{j+1} + \cdots + s_i$. *Example:* If the sequence is

27 6 −50 21 −3 14 16 −8 42 33 −21 9,

the algorithm returns 115—the sum of

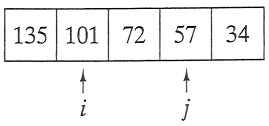
21 −3 14 16 −8 42 33.

If all the numbers in a sequence are negative, the maximum sum of consecutive values is defined to be 0. (The idea is that the maximum of 0 is achieved by taking an "empty" sum.)

Attacking the Problem

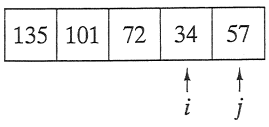
In developing an algorithm, a good way to start is to ask the question, "How would I solve this problem by hand?" At least initially, take a straightforward approach. Here we might just list the sums of *all* consecutive values and pick the largest. For the example sequence, the sums are as follows:

	j											
i	1	2	3	4	5	6	7	8	9	10	11	12
1	27											
2	33	6										
3	−17	−44	−50									
4	4	−23	−29	21								
5	1	−26	−32	18	−3							
6	15	−12	−18	32	11	14						
7	31	4	−2	48	27	30	16					
8	23	−4	−10	40	19	22	8	−8				
9	65	38	32	82	61	64	50	34	42			
10	98	71	65	115	94	97	83	67	75	33		
11	77	50	44	94	73	76	62	46	54	12	−21	
12	86	59	53	103	82	85	71	55	63	21	−12	9



We next swap a_i and a_j, where $i = 3$ and $j = rand(3, 5) = 3$. The sequence is unchanged.

We next swap a_i and a_j, where $i = 4$ and $j = rand(4, 5) = 5$. After the swap we have



11. The while loop tests whether p occurs at index i in t. If p does occur at index i in t, t_{i+j-1} will be equal to p_j for all $j = 1, \ldots, m$. Thus j becomes $m + 1$ and the algorithm returns i. If p does not occur at index i in t, t_{i+j-1} will not be equal to p_j for some j. In this case the while loop terminates (without executing return i).

Now suppose that p occurs in t and its first occurrence is at index i in t. As noted in the previous paragraph, the algorithm correctly returns i, the smallest index in t where p occurs.

If p does not occur in t, then the while loop terminates for every i and i increments in the for loop. Therefore, the for loop runs to completion, and the algorithm correctly returns 0 to indicate that p was not found in t.

14. Input: s (the sequence $s_1, \ldots, s_n$), n, and *key*

Output: i (the index of the last occurrence of *key* in s, or 0 if *key* is not in s

```
reverse_linear_search(s, n, key) {
  i = n
  while (i ≥ 1) {
    if (s_i == key)
      return i
    i = i − 1
  }
  return 0
}
```

17. We measure the time of the algorithm by counting the number of comparisons ($t_{i+j-1} == p_j$) in the while loop.

No comparisons will be made if $n - m + 1 \leq 0$. In the remainder of this solution, we assume that $n - m + 1 > 0$.

If p is in t, m comparisons must be performed to verify that p is, in fact, in t. We can guarantee that exactly m comparisons are performed if p is at index 1 in t.

If p is not in t, at least one comparison must be performed for each i. We can guarantee that exactly one comparison is performed for each i if the first character in p does not occur in t. In this case, $n - m + 1$ comparisons are made.

If $m < n - m + 1$, the best case is that p is at index 1 in t. If $n - m + 1 < m$, the best case is that the first character in p does not occur in t. If $m = n - m + 1$, either situation is the best case.

20. Input: s (the sequence $s_1, \ldots, s_n$) and n

Output: s (sorted in nondecreasing order)

```
selection_sort(s, n) {
  for i = 1 to n − 1 {
    // find smallest in s_i, ..., s_n
    small_index = i
    for j = i + 1 to n
      if (s_j < s_small_index)
        small_index = j
    swap(s_i, s_small_index)
  }
}
```

Section 4.3 Review

1. Analysis of algorithms refers to the process of deriving estimates for the time and space needed to execute algorithms.
2. The worst-case time for input of size n of an algorithm is the maximum time needed to execute the algorithm among all inputs of size n.
3. The best-case time for input of size n of an algorithm is the minimum time needed to execute the algorithm among all inputs of size n.
4. The average-case time for input of size n of an algorithm is the average time needed to execute the algorithm over some finite set of inputs all of size n.
5. $f(n) = O(g(n))$ if there exists a positive constant C_1 such that $|f(n)| \leq C_1|g(n)|$ for all but finitely many positive integers n. This notation is called the big oh notation.
6. Except for constants and a finite number of exceptions, f is bounded above by g.
7. $f(n) = \Omega(g(n))$ if there exists a positive constant C_2 such that $|f(n)| \geq C_2|g(n)|$ for all but finitely many positive integers n. This notation is called the omega notation.
8. Except for constants and a finite number of exceptions, f is bounded below by g.
9. $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. This notation is called the theta notation.
10. Except for constants and a finite number of exceptions, f is bounded above and below by g.

Section 4.3

1. $\Theta(n)$ **4.** $\Theta(n^2)$

7. $\Theta(n^2)$ **10.** $\Theta(n)$

13. $\Theta(n^2)$ **16.** $\Theta(n)$

19. $\Theta(n^2)$ **22.** $\Theta(n^3)$

25. $\Theta(n \lg n)$ **28.** $\Theta(1)$

31. When $n = 1$, we obtain

$$1 = A + B + C.$$

When $n = 2$, we obtain

$$3 = 4A + 2B + C.$$

When $n = 3$, we obtain

$$6 = 9A + 3B + C.$$

Solving this system for A, B, C, we obtain

$$A = B = \frac{1}{2}, \qquad C = 0.$$

We obtain the formula

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

which can be proved using mathematical induction (see Section 1.7).

33. $n! = n(n-1)\cdots 2 \cdot 1 \leq n \cdot n \cdots n = n^n$

36. Since $n = 2^{\lg n}$, $n^{n+1} = (2^{\lg n})^{n+1} = 2^{(n+1)\lg n}$. Thus, it suffices to show that $(n+1)\lg n \le n^2$ for all $n \ge 1$. A proof by induction shows that $n \le 2^{n-1}$ for all $n \ge 1$. Thus, $\lg n \le n-1$ for all $n \ge 1$. Therefore,

$$(n+1)\lg n \le (n+1)(n-1) = n^2 - 1 < n^2 \quad \text{for all } n \ge 1.$$

39. Since $f(n) = O(g(n))$, there exist constants $C' > 0$ and N such that

$$f(n) \le C'g(n) \quad \text{for all } n \ge N.$$

Let

$$C = \max\{C', f(1)/g(1), f(2)/g(2), \ldots, f(N)/g(N)\}.$$

For $n \le N$,

$$f(n)/g(n) \le \max\{f(1)/g(1), f(2)/g(2), \ldots, f(N)/g(N)\} \le C.$$

For $n \ge N$,

$$f(n) \le C'g(n) \le Cg(n).$$

Therefore, $f(n) \le Cg(n)$ for all n.

42. False. If the statement were true, we would have $n^n \le C2^n$ for some constant C and for all sufficiently large n. The preceding inequality may be rewritten as

$$\left(\frac{n}{2}\right)^n \le C$$

for some constant C and for all sufficiently large n. Since $(n/2)^n$ becomes arbitrarily large as n becomes large, we cannot have $n^n \le C2^n$ for some constant C and for all sufficiently large n.

44. True

46. False. A counterexample is $f(n) = n$ and $g(n) = 2n$.

49. True

52. False. A counterexample is $f(n) = 1$ and $g(n) = 1/n$.

53. $f(n) \ne O(g(n))$ means that for every positive constant C, $|f(n)| > C|g(n)|$ for infinitely many positive integers n.

56. We first find nondecreasing positive functions f_0 and g_0 such that for infinitely many n, $f_0(n) = n^2$ and $g_0(n) = n$. This implies that $f_0(n) \ne O(g_0(n))$. Our functions also satisfy $f_0(n) = n$ and $g_0(n) = n^2$ for infinitely many n [obviously different n than those for which $f_0(n) = n^2$ and $g_0(n) = n$]. This implies that $g_0(n) \ne O(f_0(n))$. If we then set $f(n) = f_0(n) + n$ and $g(n) = g_0(n) + n$, we obtain increasing positive functions for which $f(n) \ne O(g(n))$ and $g(n) \ne O(f(n))$.

We begin by setting $f_0(2) = 2$ and $g_0(2) = 2^2$. Then

$$f_0(n) = n, \quad g_0(n) = n^2, \qquad \text{if } n = 2.$$

Because g_0 is nondecreasing, the least n for which we may have $g_0(n) = n$ is $n = 2^2$. So we define $f_0(2^2) = 2^4$ and $g_0(2^2) = 2^2$. Then

$$f_0(n) = n^2, \quad g_0(n) = n, \qquad \text{if } n = 2^2.$$

The preceding discussion motivates defining

$$f_0(2^{2^k}) = \begin{cases} 2^{2^k} & \text{if } k \text{ is even} \\ 2^{2^{k+1}} & \text{if } k \text{ is odd} \end{cases}$$

$$g_0(2^{2^k}) = \begin{cases} 2^{2^{k+1}} & \text{if } k \text{ is even} \\ 2^{2^k} & \text{if } k \text{ is odd.} \end{cases}$$

Suppose that $n = 2^{2^k}$. If k is odd, $f_0(n) = n^2$ and $g_0(n) = n$; if k is even, $f_0(n) = n$ and $g_0(n) = n^2$. Now f_0 and g_0 are defined only for $n = 2^{2^k}$, but they are nondecreasing on this domain. To extend their domains to the set of positive integers, we may simply define $f_0(1) = g_0(1) = 1$ and make them constant on sets of the form $\{i \mid 2^{2^k} \le i < 2^{2^{k+1}}\}$.

60. No

62. (a) The sum of the areas of the rectangles below the curve is equal to

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

This area is less than the area under the curve, which is equal to

$$\int_1^n \frac{1}{x}dx = \log_e n.$$

The given inequality now follows immediately.

(b) The sum of the areas of the rectangles whose bases are on the x-axis and whose tops are above the curve is equal to

$$1 + \frac{1}{2} + \cdots + \frac{1}{n-1}.$$

Since this area is greater than the area under the curve, the given inequality follows immediately.

(c) Part (a) shows that

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

Since $\log_e n = \Theta(\lg n)$ (see Example 4.3.6),

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

Similarly, we can conclude from part (b) that

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

Therefore,

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

64. Replacing a by b in the sum yields

$$\frac{b^{n+1} - a^{n+1}}{b-a} = \sum_{i=0}^{n} a^i b^{n-i} < \sum_{i=0}^{n} b^i b^{n-i} = \sum_{i=0}^{n} b^n = (n+1)b^n.$$

67. By Exercise 65, the sequence $\{(1 + 1/n)^n\}_{n=1}^{\infty}$ is increasing. Therefore

$$2 = \left(1 + \frac{1}{1}\right)^1 \le \left(1 + \frac{1}{n}\right)^n$$

for every positive integer n. Exercise 66 shows that

$$\left(1 + \frac{1}{n}\right)^n < 4$$

for every positive integer n. Taking logs to the base 2, we obtain

$$1 = \lg 2 \le \lg\left(1 + \frac{1}{n}\right)^n < \lg 4 = 2.$$

Since

$$\lg\left(1 + \frac{1}{n}\right)^n = n\lg\left(1 + \frac{1}{n}\right) = n\lg\left(\frac{n+1}{n}\right) = n[\lg(n+1) - \lg n],$$

From: Parberry et al.
Problems on Algorithms

Chapter 3

Big-O and Big-Ω

Big-O notation is useful for the analysis of algorithms since it captures the asymptotic growth pattern of functions and ignores the constant multiple (which is out of our control anyway when algorithms are translated into programs). We will use the following definitions (although alternatives exist; see Section 3.5). Suppose that $f, g : \mathbb{N} \to \mathbb{N}$.

- $f(n)$ is $O(g(n))$ if there exists $c, n_0 \in \mathbb{R}^+$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.
- $f(n)$ is $\Omega(g(n))$ if there exists $c, n_0 \in \mathbb{R}^+$ such that for all $n \geq n_0$, $f(n) \geq c \cdot g(n)$.
- $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.
- $f(n)$ is $o(g(n))$ if $\lim_{n\to\infty} f(n)/g(n) = 0$.
- $f(n)$ is $\omega(g(n))$ if $\lim_{n\to\infty} g(n)/f(n) = 0$.
- $f(n) \sim g(n)$ if $\lim_{n\to\infty} f(n)/g(n) = 1$.

We will follow the normal convention of writing, for example, "$f(n) = O(g(n))$" instead of "$f(n)$ is $O(g(n))$", even though the equality sign does not indicate true equality. The proper definition of "$O(g(n))$" is as a set:

$O(g(n)) = \{f(n) \mid \text{there exists } c, n_0 \in \mathbb{R}^+ \text{ such that for all } n \geq n_0,\ f(n) \leq c \cdot g(n)\}$.

Then, "$f(n) = O(g(n))$" can be interpreted as meaning "$f(n) \in O(g(n))$."

3.1 RANK THE FUNCTIONS

98. Consider the following eighteen functions:

$\sqrt{n}$	n	2^n
$n \log n$	$n - n^3 + 7n^5$	$n^2 + \log n$
n^2	n^3	$\log n$
$n^{1/3} + \log n$	$(\log n)^2$	$n!$
$\ln n$	$n/\log n$	$\log \log n$
$(1/3)^n$	$(3/2)^n$	6

CS404: Consider the 'single cap' problems

CS5592: Consider both 'single' + 'double' cap problems

Group these functions so that $f(n)$ and $g(n)$ are in the same group if and only if $f(n) = O(g(n))$ and $g(n) = O(f(n))$, and list the groups in increasing order.

99. Draw a line from each of the five functions in the center to the best big-Ω value on the left, and the best big-O value on the right.

$\Omega(1/n)$		$O(1/n)$
$\Omega(1)$		$O(1)$
$\Omega(\log\log n)$		$O(\log\log n)$
$\Omega(\log n)$		$O(\log n)$
$\Omega(\log^2 n)$		$O(\log^2 n)$
$\Omega(\sqrt[3]{n})$		$O(\sqrt[3]{n})$
$\Omega(n/\log n)$	$1/(\log n)$	$O(n/\log n)$
$\Omega(n)$	$7n^5 - 3n + 2$	$O(n)$
$\Omega(n^{1.00001})$	$(n^2+n)/(\log^2 n + \log n)$	$O(n^{1.00001})$
$\Omega(n^2/\log^2 n)$	$2^{\log^2 n}$	$O(n^2/\log^2 n)$
$\Omega(n^2/\log n)$	3^n	$O(n^2/\log n)$
$\Omega(n^2)$		$O(n^2)$
$\Omega(n^{3/2})$		$O(n^{3/2})$
$\Omega(2^n)$		$O(2^n)$
$\Omega(5^n)$		$O(5^n)$
$\Omega(n^n)$		$O(n^n)$
$\Omega(n^{n^2})$		$O(n^{n^2})$

For each of the following pairs of functions $f(n)$ and $g(n)$, either $f(n) = O(g(n))$ or $g(n) = O(f(n))$, but not both. Determine which is the case.

100. $f(n) = (n^2 - n)/2$, $g(n) = 6n$.

101. $f(n) = n + 2\sqrt{n}$, $g(n) = n^2$.

102. $f(n) = n + \log n$, $g(n) = n\sqrt{n}$.

103. $f(n) = n^2 + 3n + 4$, $g(n) = n^3$.

104. $f(n) = n\log n$, $g(n) = n\sqrt{n}/2$.

105. $f(n) = n + \log n$, $g(n) = \sqrt{n}$.

106. $f(n) = 2(\log n)^2$, $g(n) = \log n + 1$.

107. $f(n) = 4n\log n + n$, $g(n) = (n^2 - n)/2$.

3.2 TRUE OR FALSE?

108. $n^2 = O(n^3)$.
109. $n^3 = O(n^2)$.
110. $2n^2 + 1 = O(n^2)$.
111. $n \log n = O(n\sqrt{n})$.
112. $\sqrt{n} = O(\log n)$.
113. $\log n = O(\sqrt{n})$.
114. $n^3 = O(n^2(1 + n^2))$.
115. $n^2(1 + \sqrt{n}) = O(n^2)$.
116. $n^2(1 + \sqrt{n}) = O(n^2 \log n)$.
117. $3n^2 + \sqrt{n} = O(n^2)$.
118. $3n^2 + \sqrt{n} = O(n + n\sqrt{n} + \sqrt{n})$.
119. $\log n + \sqrt{n} = O(n)$.
120. $\sqrt{n} \log n = O(n)$.
121. $1/n = O(\log n)$.
122. $\log n = O(1/n)$.
123. $\log n = O(n^{-1/2})$.
124. $n + \sqrt{n} = O(\sqrt{n} \log n)$.
125. If $f(n) \sim g(n)$, then $f(n) = \Theta(g(n))$.
126. If $f(n) = \Theta(g(n))$, then $g(n) = \Theta(f(n))$.

For each of the following pairs of functions $f(n)$ and $g(n)$, state whether $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \Theta(g(n))$, or none of the above.

127. $f(n) = n^2 + 3n + 4$, $g(n) = 6n + 7$.
128. $f(n) = \sqrt{n}$, $g(n) = \log(n + 3)$.
129. $f(n) = n\sqrt{n}$, $g(n) = n^2 - n$.
130. $f(n) = n + n\sqrt{n}$, $g(n) = 4n \log(n^2 + 1)$.
131. $f(n) = (n^2 + 2)/(1 + 2^{-n})$, $g(n) = n + 3$.
132. $f(n) = 2^n - n^2$, $g(n) = n^4 + n^2$.

3.3 PROVING BIG-O

133. Prove that $(n+1)^2 = O(n^2)$.

134. Prove that $3n^2 - 8n + 9 = O(n^2)$.

135. Prove that for all $k \geq 1$ and all $a_k, a_{k-1}, \ldots, a_1, a_0 \in \mathbb{R}$,

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = O(n^k).$$

136. Prove that $\lceil \log n \rceil = O(n)$.

137. Prove that $3n\lfloor \log n \rfloor = O(n^2)$.

138. Prove that $n^2 - 3n - 18 = \Omega(n)$.

139. Prove that $n^3 - 3n^2 - n + 1 = \Theta(n^3)$.

140. Prove that $n = O(2^n)$.

141. Prove that $2n + 1 = O(2^n)$.

142. Prove that $9999n + 635 = O(2^n)$.

143. Prove that $cn + d = O(2^n)$ for all $c, d \in \mathbb{R}^+$.

144. Prove that $n^2 = O(2^n)$.

145. Prove that $cn^2 + d = O(2^n)$ for all $c, d \in \mathbb{R}^+$.

146. Prove that $cn^k + d = O(2^n)$ for all $c, d, k \in \mathbb{R}^+$.

147. Prove that $2^n = O(n!)$.

148. Prove that $n! = \Omega(2^n)$.

149. Does $n^{\log n} = O((\log n)^n)$? Prove your answer.

150. Does $n^{\log n} = \Omega((\log n)^n)$? Prove your answer.

151. Does $n^{\log \log \log n} = O((\log n)!)$? Prove your answer.

152. Does $n^{\log \log \log n} = \Omega((\log n)!)$? Prove your answer.

153. Does $(n!)! = O(((n-1)!)!(n-1)!^{n!})$? Prove your answer.

154. Does $(n!)! = \Omega(((n-1)!)!(n-1)!^{n!})$? Prove your answer.

155. Prove or disprove:

$$O\left(\left(\frac{n^2}{\log \log n}\right)^{1/2}\right) = O(\lfloor \sqrt{n} \rfloor).$$

156. Prove or disprove: $2^{(1+O(1/n))^2} = 2 + O(1/n)$.

Compare the following pairs of functions f, g. In each case, say whether $f = o(g)$, $f = \omega(g)$, or $f = \Theta(g)$, and prove your claim.

157. $f(n) = 100n + \log n$, $g(n) = n + (\log n)^2$.

158. $f(n) = \log n$, $g(n) = \log\log(n^2)$.

159. $f(n) = n^2/\log n$, $g(n) = n(\log n)^2$.

160. $f(n) = (\log n)^{10^6}$, $g(n) = n^{10^{-6}}$.

161. $f(n) = n \log n$, $g(n) = (\log n)^{\log n}$.

162. $f(n) = n2^n$, $g(n) = 3^n$.

For each of the following pairs of functions $f(n)$ and $g(n)$, find $c \in \mathbb{R}^+$ such that $f(n) \leq c \cdot g(n)$ for all $n > 1$.

163. $f(n) = n^2 + n$, $g(n) = n^2$.

164. $f(n) = 2\sqrt{n} + 1$, $g(n) = n + n^2$.

165. $f(n) = n^2 + n + 1$, $g(n) = 2n^3$.

166. $f(n) = n\sqrt{n} + n^2$, $g(n) = n^2$.

167. $f(n) = 12n + 3$, $g(n) = 2n - 1$.

168. $f(n) = n^2 - n + 1$, $g(n) = n^2/2$.

169. $f(n) = 5n + 1$, $g(n) = (n^2 - 6n)/2$.

170. $f(n) = 5\lfloor\sqrt{n}\rfloor - 1$, $g(n) = n - \lceil\sqrt{n}\rceil$.

3.4 MANIPULATING BIG-O

171. Prove that if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

172. Prove that if $f_1(n) = \Omega(g_1(n))$ and $f_2(n) = \Omega(g_2(n))$, then $f_1(n) + f_2(n) = \Omega(g_1(n) + g_2(n))$.

173. Prove that if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$.

174. Prove that if $f_1(n) = \Omega(g_1(n))$ and $f_2(n) = \Omega(g_2(n))$, then $f_1(n) + f_2(n) = \Omega(\min\{g_1(n), g_2(n)\})$.

175. Suppose that $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$. Is it true that $f_1(n) + f_2(n) = \Theta(g_1(n) + g_2(n))$? Is it true that $f_1(n) + f_2(n) = \Theta(\max\{g_1(n), g_2(n)\})$? Is it true that $f_1(n) + f_2(n) = \Theta(\min\{g_1(n), g_2(n)\})$? Justify your answer.

176. Prove that if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.

177. Prove that if $f_1(n) = \Omega(g_1(n))$ and $f_2(n) = \Omega(g_2(n))$, then $f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdot g_2(n))$.

178. Prove or disprove: For all functions $f(n)$ and $g(n)$, either $f(n) = O(g(n))$ or $g(n) = O(f(n))$.

179. Prove or disprove: If $f(n) > 0$ and $g(n) > 0$ for all n, then $O(f(n) + g(n)) = f(n) + O(g(n))$.

180. Prove or disprove: $O(f(n)^\alpha) = O(f(n))^\alpha$ for all $\alpha \in \mathbb{R}^+$.

181. Prove or disprove: $O(x + y)^2 = O(x^2) + O(y^2)$.

182. Multiply $\log n + 6 + O(1/n)$ by $n + O(\sqrt{n})$ and simplify your answer as much as possible.

183. Show that big-O is transitive. That is, if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

184. Prove that if $f(n) = O(g(n))$, then $f(n)^k = O(g(n)^k)$.

185. Prove or disprove: If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.

186. Prove or disprove: If $f(n) = O(g(n))$, then $\log f(n) = O(\log g(n))$.

187. Suppose $f(n) = \Theta(g(n))$. Prove that $h(n) = O(f(n))$ iff $h(n) = O(g(n))$.

188. Prove or disprove: If $f(n) = O(g(n))$, then $f(n)/h(n) = O(g(n)/h(n))$.

3.5 ALTERNATIVE DEFINITIONS

Here is an alternative definition of O.

- $f(n)$ is $O_1(g(n))$ if there exists $c \in \mathbb{R}$ such that $\lim_{n\to\infty} f(n)/g(n) = c$.

189. Prove that if $f(n) = O(g(n))$, then $f(n) = O_1(g(n))$, or find a counterexample to this claim.

190. Prove that if $f(n) = O_1(g(n))$, then $f(n) = O(g(n))$, or find a counterexample to this claim.

Here are two alternative definitions of Ω.

- $f(n)$ is $\Omega_1(g(n))$ if there exists $c \in \mathbb{R}^+$ such that for infinitely many n, $f(n) \geq c \cdot g(n)$.
- $f(n)$ is $\Omega_2(g(n))$ if there exists $c \in \mathbb{R}^+$ such that for all $n_0 \in \mathbb{N}$, there exists $n \geq n_0$ such that $f(n) \geq c \cdot g(n)$.

191. Prove that if $f(n) = \Omega(g(n))$, then $f(n) = \Omega_2(g(n))$, or find a counterexample to this claim.

192. Prove that if $f(n) = \Omega_2(g(n))$, then $f(n) = \Omega(g(n))$, or find a counterexample to this claim.

193. Prove that if $f(n) = \Omega_1(g(n))$, then $f(n) = \Omega_2(g(n))$, or find a counterexample to this claim.

194. Prove that if $f(n) = \Omega_2(g(n))$, then $f(n) = \Omega_1(g(n))$, or find a counterexample to this claim.

195. Prove or disprove: If $f(n) \neq O(g(n))$, then $f(n) = \Omega(g(n))$. If $f(n) \neq O(g(n))$, then $f(n) = \Omega_2(g(n))$.

196. Define the relation $\equiv$ by $f(n) \equiv g(n)$ iff $f(n) = \Omega(g(n))$ and $g(n) = \Omega(f(n))$. Similarly, define the relation $\equiv_2$ by $f(n) \equiv g(n)$ iff $f(n) = \Omega_2(g(n))$ and $g(n) = \Omega_2(f(n))$. Show that $\equiv$ is an equivalence relation, but $\equiv_2$ is not an equivalence relation.

3.6 FIND THE FUNCTIONS

Find two functions $f(n)$ and $g(n)$ that satisfy the following relationships. If no such f and g exist, write "None."

197. $f(n) = o(g(n))$ and $f(n) \neq \Theta(g(n))$.

198. $f(n) = \Theta(g(n))$ and $f(n) = o(g(n))$.

199. $f(n) = \Theta(g(n))$ and $f(n) \neq O(g(n))$.

200. $f(n) = \Omega(g(n))$ and $f(n) \neq O(g(n))$.

201. $f(n) = \Omega(g(n))$ and $f(n) \neq o(g(n))$.

3.7 HINTS

98. There are a lot of groups.

99. Be careful with $(n^2+n)/(\log^2 n + \log n)$.

136. When solving problems that require you to prove that $f(n) = O(g(n))$, it is a good idea to try induction first. That is, pick a c and an n_0, and try to prove that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$. Try starting with $n_0 = 1$. A good way of guessing a value for c is to look at $f(n)$ and $g(n)$ for $n = n_0, \cdots, n_0 + 10$. If the first function seems to grow faster than the second, it means that you must adjust your n_0 higher — eventually (unless the thing you are trying to prove is false), the first function will grow more slowly than the second. The ratio of $f(n_0)/g(n_0)$ gives you your first cut for c. Don't be afraid to make c higher than this initial guess to help you make it easier to prove the inductive step. This happens quite often. You might even have to go back and adjust n_0 higher to make things work out correctly.

171. Start by writing down the definitions for $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$.

3.8 SOLUTIONS

133. We are required to prove that $(n+1)^2 = O(n^2)$. We need to find a constant c such that $(n+1)^2 \leq cn^2$. That is, $n^2 + 2n + 1 \leq cn^2$ or, equivalently, $(c-1)n^2 - 2n - 1 \geq 0$. Is this possible? It is if $c > 1$. Take $c = 4$. The roots of $3n^2 - 2n - 1$ are $(2 \pm \sqrt{4+12})/2 = \{-1/3, 1\}$. The second root gives us the correct value for n_0. Therefore, for all $n \geq 1$, $(n+1)^2 \leq 4n^2$, and so by definition, $(n+1)^2 = O(n^2)$.

136. We are required to prove that $\lceil \log n \rceil = O(n)$. By looking at $\lceil \log n \rceil$ for small values of n, it appears that for all $n \geq 1$, $\lceil \log n \rceil \leq n$. The proof is by induction on n. The claim is certainly true for $n = 1$. Now suppose that $n > 1$, and $\lceil \log(n-1) \rceil \leq n - 1$. Then,

$$\begin{aligned} \lceil \log n \rceil &\leq \lceil \log(n-1) \rceil + 1 \\ &\leq (n-1) + 1 \quad \text{(by the induction hypothesis)} \\ &= n. \end{aligned}$$

Hence, we can take $c = 1$ and $n_0 = 1$.

137. We are required to prove that $3n\lfloor \log n \rfloor = O(n^2)$. By looking at $3n\lfloor \log n \rfloor$ for small values of n, it appears that for all $n \geq 1$, $3n\lfloor \log n \rfloor \leq 3n^2$. The proof is by induction on n. The claim is certainly true for $n = 1$. Now suppose that

$n > 1$, and $3(n-1)\lfloor\log(n-1)\rfloor \leq 3(n-1)^2$. Then,

$$\begin{aligned}
& 3n\lfloor\log n\rfloor \\
\leq\ & 3n(\lfloor\log(n-1)\rfloor + 1) \\
=\ & 3(n-1)(\lfloor\log(n-1)\rfloor + 1) + 3(\lfloor\log(n-1)\rfloor + 1) \\
=\ & 3(n-1)\lfloor\log(n-1)\rfloor + 3(n-1) + 3(\lfloor\log(n-1)\rfloor + 1) \\
\leq\ & 3(n-1)^2 + 3(n-1) + 3(\lfloor\log(n-1)\rfloor + 1) \\
& \quad \text{(by the induction hypothesis)} \\
\leq\ & 3(n-1)^2 + 3(n-1) + 3n \quad \text{(see the solution to Problem 136)} \\
=\ & 3n^2 - 6n + 3 + 3n - 3 + 3n \\
=\ & 3n^2.
\end{aligned}$$

Hence, we can take $c = 3$ and $n_0 = 1$.

171. We are required to prove that if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n)+f_2(n) = O(g_1(n)+g_2(n))$. Suppose for all $n \geq n_1$, $f_1(n) \leq c_1 \cdot g_1(n)$ and for all $n \geq n_2$, $f_2(n) \leq c_2 \cdot g_2(n)$. Let $n_0 = \max\{n_1, n_2\}$ and $c_0 = \max\{c_1, c_2\}$. Then for all $n \geq n_0$, $f_1(n) + f_2(n) \leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n) \leq c_0(g_1(n) + g_2(n))$.

3.9 COMMENTS

173. This is often called the *sum rule* for big-Os.

176. This is often called the *product rule* for big-Os.