Chapter 4

The Efficiency of Algorithms

Contents

Prerequisites

Appendix B		Java Classes
Chapter	2	Bag Implementations That Use Arrays
Chapter	3	A Bag Implementation That Links Data

Objectives

After studying this chapter, you should be able to

- Assess the efficiency of a given algorithm
- Compare the expected execution times of two methods, given the efficiencies of their algorithms

With amazing frequency, manufacturers introduce new computers that are faster and have larger memories than their recent predecessors. Yet we—and likely your computer science professors—ask you to write code that is efficient in its use of time and space (memory). Admittedly, such efficiency is not as pressing an issue as it was fifty years ago, when computers were much slower and their memory size was much smaller than they are now. (Computers had small memories, but they were physically

huge, occupying entire rooms.) Even so, efficiency remains an issue—in some circumstances, a critical issue.

This chapter will introduce you to the terminology and ways that computer scientists use to measure the efficiency of an algorithm. With this background, not only will you have an intuitive feel for efficiency, but you also will be able to talk about efficiency in a quantitative way.

Motivation

4.1 **Example.** Perhaps you think that you are not likely to write a program in the near future whose execution time is noticeably long. You might be right, but we are about to show you some simple Java code that does take a long time to perform its computations.

Consider the problem of computing the sum $1 + 2 + \ldots + n$ for any positive integer n. Figure 4-1 contains pseudocode showing three ways to solve this problem. Algorithm A computes the sum $0 + 1 + 2 + \ldots + n$ from left to right. Algorithm B computes $0 + (1) + (1 + 1) + (1 + 1 + 1) + \ldots + (1 + 1 + \ldots + 1)$. Finally, Algorithm C uses an algebraic identity to compute the sum.

FIGURE 4-1 Three algorithms for computing the sum $1 + 2 + \ldots + n$ for an integer $n > 0$

Algorithm A

```
sum = 0
for i = 1 to n
   sum = sum + i
```

Algorithm B

```
sum = 0
for i = 1 to n
{
   for j = 1 to i
      sum = sum + 1
}
```

Algorithm C

```
sum = n * (n + 1) / 2
```

4.2 Let's translate these algorithms into Java code. If we use `long` integers, we could write the following statements:

```
// Computing the sum of the consecutive integers from 1 to n:
long n = 10000; // ten thousand

// Algorithm A
long sum = 0;
for (long i = 1; i <= n; i++)
   sum = sum + i;
System.out.println(sum);

// Algorithm B
sum = 0;
for (long i = 1; i <= n; i++)
{
   for (long j = 1; j <= i; j++)
      sum = sum + 1;
} // end for
System.out.println(sum);

// Algorithm C
sum = n * (n + 1) / 2;
System.out.println(sum);
```

If you execute this code with n equal to ten thousand (10000), you will get the right answer of 50005000 for each of the algorithms. Now change the value of n to one hundred thousand (100000), and execute the code again. Once more, you will get the correct answer, which this time is 5000050000. However, you should notice a delay in seeing the result for Algorithm B. Now try one million (1000000) for the value of n. Again you will get the correct answer—500000500000—but you will have to wait even longer for the result from Algorithm B. The wait might be long enough for you to suspect that something is broken. If not, try a larger value of n.

The previous simple code for Algorithm B takes a noticeably long time to execute, much longer than either of the other two algorithms. If it were the only algorithm you tried, what should you do? Use a faster computer? While that might be a solution, it's clear that we should use a different algorithm.

Note: As the previous example shows, even a simple program can be noticeably inefficient.

Note: If an algorithm takes longer to execute than is practical, try to reformulate it to make it more efficient of time.

Measuring an Algorithm's Efficiency

4.3 The previous section should have convinced you that a program's efficiency matters. How can we measure efficiency so that we can compare various approaches to solving a problem? In the previous section, we computed the sum of the first n consecutive integers in three different ways. We then observed that one was noticeably slower than the others as the value of n increased. In general, however, implementing several ideas before you choose one requires too much work to be practical. Besides, a program's execution time depends in part on the particular computer and the programming language used. It would be much better to measure an *algorithm's* efficiency before you implement it.

VideoNote
Measuring efficiency

For example, suppose that you want to go to a store downtown. Your options are to walk, drive your car, ask a friend to take you, or take a bus. What is the best way? First, what is your concept of best? Is it the way that saves money, your time, your friend's time, or the environment? Let's say that the best option for you is the fastest one. After defining your criterion, how do you evaluate your options? You certainly do not want to try all four options so you can discover which is fastest. That would be like writing four different programs that perform the same task so you can measure which one is fastest. Instead you would investigate the "cost" of each option, considering the distance, the speed at which you can travel, the amount of other traffic, the number of stops at traffic lights, the weather, and so on. That is, you would consider the factors that have the most impact on the cost.

4.4 The same considerations apply when deciding what algorithm is best. Again, we need to define what we mean by best. An algorithm has both time and space requirements, called its **complexity**, that we can measure. When we assess an algorithm's complexity, we are not measuring how involved or difficult it is. Instead, we measure an algorithm's **time complexity**—the time it takes to execute—or its **space complexity**—the memory it needs to execute. Typically we analyze these requirements separately. So a "best" algorithm might be the fastest one or the one that uses the least memory.

Time and space complexity

Note: What's best?
Usually the "best" solution to a problem balances various criteria such as time, space, generality, programming effort, and so on.

The process of measuring the complexity of algorithms is called the **analysis of algorithms**. We will concentrate on the time complexity of algorithms, because it is usually more important than the space complexity. You should realize that an inverse relationship often exists between an algorithm's time complexity and its space complexity. If you revise an algorithm to save execution time, you usually will need more space. If you reduce an algorithm's space requirement, it likely will require more time to execute. Sometimes, however, you will be able to save both time and space.

Your measure of the complexity of an algorithm should be easy to compute, certainly easier than implementing the algorithm. You should express this measure in terms of the size of the problem. This **problem size** is the number of items that an algorithm processes. For example, if you are searching a collection of data, the problem size is the number of items in the collection. Such a measure enables you to compare the relative cost of algorithms as a function of the size of the problem. Typically, we are interested in large problems; a small problem is likely to take little time, even if the algorithm is inefficient.

Problem size

4.5 Realize that you cannot compute the actual time requirement of an algorithm. After all, you have not implemented the algorithm in Java and you have not chosen the computer. Instead, you find a function of the problem size that behaves like the algorithm's actual time requirement. Therefore, as the time requirement increases by some factor, the value of the function increases by the same factor, and vice versa. The value of the function is said to be **directly proportional** to the time requirement. Such a function is called a **growth-rate function** because it measures how an algorithm's time requirement grows as the problem size grows. Because they measure time requirements, growth-rate functions have positive values. By comparing the growth-rate functions of two algorithms, you can see whether one algorithm is faster than the other for large-size problems.

4.6 **Example.** Consider again the problem of computing the sum $1 + 2 + \ldots + n$ for any positive integer n. Figure 4-1 gives three algorithms—A, B, and C— to perform this computation. Algorithm A computes the sum $0 + 1 + 2 + \ldots + n$ from left to right. Algorithm B computes $0 + (1) + (1 + 1) + (1 + 1 + 1) + \ldots + (1 + 1 + \ldots + 1)$, and Algorithm C uses an algebraic identity to compute the sum. By executing the Java code in Segment 4.2, we found that Algorithm B is the slowest. We now want to predict this behavior without actually running the code.

So how can we tell which algorithm is slowest and which is fastest? We can begin to answer these questions by considering both the size of the problem and the effort involved. The integer n is a measure of the problem size: As n increases, the sum involves more terms. To measure the effort, or time requirement, of an algorithm, we must find an appropriate growth-rate function. To do so, we might begin by counting the number of operations required by the algorithm.

For example, Algorithm A in Figure 4-1 contains the pseudocode statement

```
for i = 1 to n
```

This statement represents the following loop-control logic:

```
i = 1
while (i <= n)
{
   ...
   i = i + 1
}
```

This logic requires an assignment to i, $n + 1$ comparisons between i and n, n additions to i, and n more assignments to i. In total, the loop-control logic requires $n + 1$ assignments, $n + 1$ comparisons, and n additions. Furthermore, Algorithm A requires for its initialization and loop body another $n + 1$ assignments and n additions. All together, Algorithm A requires $2n + 2$ assignments, $2n$ additions, and $n + 1$ comparisons.

These various operations probably take different amounts of time to execute. For example, if each assignment takes no more than $t_=$ time units, each addition takes no more than t_+ time units, and each comparison takes no more than t_c time units, Algorithm A would require no more than

$$(2n + 2)\, t_= + (2n)\, t_+ + (n + 1)\, t_c \text{ time units}$$

If we replace $t_=$, t_+, and t_c with the largest of the three values and call it t, Algorithm A requires no more than $(5n + 3)\, t$ time units. We conclude that Algorithm A requires time directly proportional to $5n + 3$.

What is important, however, is not the exact count of operations, but the general behavior of the algorithm. The function $5n + 3$ is directly proportional to n. As you are about to see, we do not have to count every operation to see that Algorithm A requires time that increases linearly with n.

Counting Basic Operations

4.7 An algorithm's **basic operation** is the most significant contributor to its total time requirement. For example, Algorithms A and B in Figure 4-1 have addition as their basic operation. An algorithm that sees whether an array contains a particular object has comparison as its basic operation. Realize that the most frequent operation is not necessarily the basic operation. For example, assignments are often the most frequent operation in an algorithm, but they rarely are basic.

Ignoring operations that are not basic, such as initializations of variables, the operations that control loops, and so on, will not affect our final conclusion about algorithm speed. For example, Algorithm A requires n additions of i to sum in the body of the loop. We can conclude that Algorithm A requires time that increases linearly with n, even though we ignored operations that are not basic to the algorithm.

Whether we look at the number, n, of basic operations or the total number of operations, $5n + 3$, we can draw the same conclusion: Algorithm A requires time directly proportional to n. Thus, Algorithm A's growth-rate function is n.

4.8 

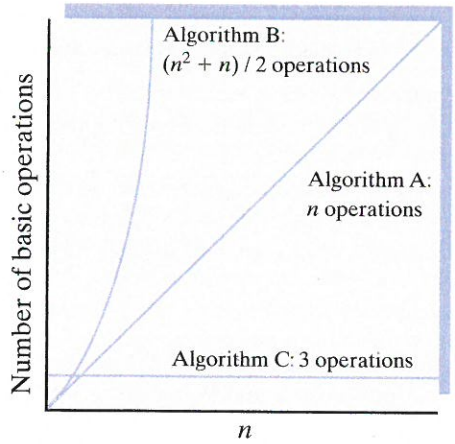
Example continued. Now let's count the number of basic operations required by Algorithms B and C. The basic operation for Algorithm B is addition; for Algorithm C, the basic operations are addition, multiplication, and division. Figure 4-2 tabulates the number of basic operations that Algorithms A, B, and C require. Remember, these counts do not include assignments and the operations that control the loops. Our discussion in the previous segment should have convinced you that we can ignore these operations.

FIGURE 4-2 The number of basic operations required by the algorithms in Figure 4-1

	Algorithm A	Algorithm B	Algorithm C
Additions	n	$n\,(n + 1) / 2$	1
Multiplications			1
Divisions			1
Total basic operations	n	$(n^2 + n) / 2$	**3**

Algorithm B requires time directly proportional to $(n^2 + n) / 2$, and Algorithm C requires time that is constant and independent of the value of n. Figure 4-3 plots these time requirements as a function of n. You can see from this figure that as n grows, Algorithm B requires the most time.

FIGURE 4-3 The number of basic operations required by the algorithms in Figure 4-1 as a function of n



Question 1 For any positive integer n, the identity

$$1 + 2 + \ldots + n = n (n + 1) / 2$$

is one that you will encounter while analyzing algorithms. Can you derive it? If you can, you will not need to memorize it. *Hint*: Write $1 + 2 + \ldots + n$. Under it write $n + (n - 1) + \ldots + 1$. Then add the terms from left to right.

Question 2 Can you derive the values in Figure 4-2? *Hint*: For Algorithm B, use the identity given in Question 1.

Note: Useful identities

$1 + 2 + \ldots + n = n (n + 1) / 2$

$1 + 2 + \ldots + (n - 1) = n (n - 1) / 2$

4.9 Typical growth-rate functions are algebraically simple. Why? Recall that since you are not likely to notice the effect of an inefficient algorithm when the problem is small, you should focus on large problems. Thus, if we care only about large values of n when comparing the algorithms, we can consider only the dominant term in each growth-rate function.

For example, $(n^2 + n) / 2$ behaves like n^2 when n is large. First, n^2 is much larger than n for large values of n, so $(n^2 + n) / 2$ behaves like $n^2 / 2$. Moreover, $n^2 / 2$ behaves like n^2 when n is large. In other words, for large n, the difference between the value of $(n^2 + n) / 2$ and that of n^2 is relatively small and can be ignored. So instead of using $(n^2 + n) / 2$ as Algorithm B's growth-rate

function, we can use n^2—the term with the largest exponent—and say that Algorithm B requires time proportional to n^2. On the other hand, Algorithm C requires time that is independent of n, and we saw earlier that Algorithm A requires time proportional to n. We conclude that Algorithm C is the fastest and Algorithm B is the slowest.

Note: The relative magnitudes of common growth-rate functions
The growth-rate functions that you are likely to encounter grow in magnitude as follows when $n > 10$:

$$1 < \log(\log n) < \log n < \log^2 n < n < n \log n < n^2 < n^3 < 2^n < n!$$

The logarithms given here are base 2. As you will see later in Segment 4.16, the choice of base does not matter.

Figure 4-4 tabulates the magnitudes of these functions for increasing values of the problem size n. From this data you can see that algorithms whose growth-rate functions are log(log n), log n, or $\log^2 n$ take much less time than algorithms whose growth-rate function is n. Although the value of $n \log n$ is significantly larger than n, either of those functions describes a growth rate that is markedly faster than n^2.

FIGURE 4-4 Typical growth-rate functions evaluated at increasing values of n

n	$\log(\log n)$	$\log n$	$\log^2 n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	2	3	11	10	33	10^2	10^3	10^3	10^5
10^2	3	7	44	100	664	10^4	10^6	10^{30}	10^{94}
10^3	3	10	99	1000	9966	10^6	10^9	10^{301}	10^{1435}
10^4	4	13	177	10,000	132,877	10^8	10^{12}	10^{3010}	$10^{19,335}$
10^5	4	17	276	100,000	1,660,964	10^{10}	10^{15}	$10^{30,103}$	$10^{243,338}$
10^6	4	20	397	1,000,000	19,931,569	10^{12}	10^{18}	$10^{301,030}$	$10^{2,933,369}$

Note: When analyzing the time efficiency of an algorithm, consider large problems. For small problems, the difference between the execution times of two solutions to the same problem is usually insignificant.

Best, Worst, and Average Cases

4.10 For some algorithms that operate on a data set, the execution time depends only on the size of the data set. For example, the time needed to find the smallest integer in an array of integers depends only on the number of integers, not on the integers themselves. Finding the smallest of 100 integers takes the same amount of time regardless of the values of the integers.

Other algorithms, however, have time requirements that depend not only on the size of the data set, but also on the data itself. For example, imagine that an array contains a certain value, and we want to know where in the array it occurs. Suppose our search algorithm examines each value in the array until it finds the desired one. If the algorithm finds this desired value in the first array element it examines, it makes only one comparison. In this **best case**, the algorithm takes the least

time. The algorithm can do no better than its best-case time. If the best-case time is still too slow, you need another algorithm.

Now suppose that the algorithm locates the desired value after comparing it to every value in the array. This would be the algorithm's **worst case**, since it requires the most time. If you can tolerate this worst-case time, your algorithm is acceptable. For many algorithms, the worst and best cases rarely occur. Thus, we consider an algorithm's **average case**, when it processes a typical data set. The average-case time requirement of an algorithm is more useful, but harder to estimate. Note that the average-case time is not the average of the best-case and worst-case times.

Note: The time requirements of some algorithms depend on the data values given to them. Those times range from a minimum, or best-case, time to a maximum, or worst-case, time. Typically, the best and worst cases do not occur. A more useful measure of such an algorithm's time requirement is its average-case time.

Some algorithms, however, do not have a best, worst, and average case. Their time requirements depend only on the number of data items given them, not on the values of that data.

Big Oh Notation

4.11 Computer scientists use a notation to represent an algorithm's complexity. For example, consider the algorithms A, B, and C given in Figure 4-1 and the number of basic operations that each requires, as shown in Figure 4-2. Instead of saying that Algorithm A has a time requirement proportional to n, we say that A is **O(*n*)**. We call this notation **Big Oh** since it uses the capital letter O. We read O(n) as either "Big Oh of n" or "order of at most n." Similarly, since Algorithm B has a time requirement proportional to n^2, we say that B is O(n^2). Algorithm C always requires three basic operations. Regardless of the problem size n, this algorithm requires the same time. We say that Algorithm C is O(1).

4.12  **Example.** Imagine that you are at a wedding reception, seated at a table of n people. In preparation for the toast, the waiter pours champagne into each of n glasses. That task is O(n). Someone makes a toast. It is O(1), even if the toast seems to last forever, because it is independent of the number of guests. If you clink your glass with everyone at your table, you perform an O(n) operation. If everyone at your table does likewise, a total of O(n^2) clinks are performed.

4.13 Big Oh notation has a formal mathematical meaning that can justify our discussion in the previous sections. You saw that an algorithm's actual time requirement is directly proportional to a function f of the problem size n. For example, $f(n)$ might be $n^2 + n + 1$. In this case, we would conclude that the algorithm is of order at most n^2—that is, O(n^2). We essentially have replaced $f(n)$ with a simpler function—let's call it $g(n)$. In this example, $g(n)$ is n^2.

What does it really mean to say that a function $f(n)$ is of order at most $g(n)$—that is, $f(n)$ is O($g(n)$), or $f(n)$ = O($g(n)$)? Formally, its meaning is described by the following mathematical definition:

Note: **Formal definition of Big Oh**

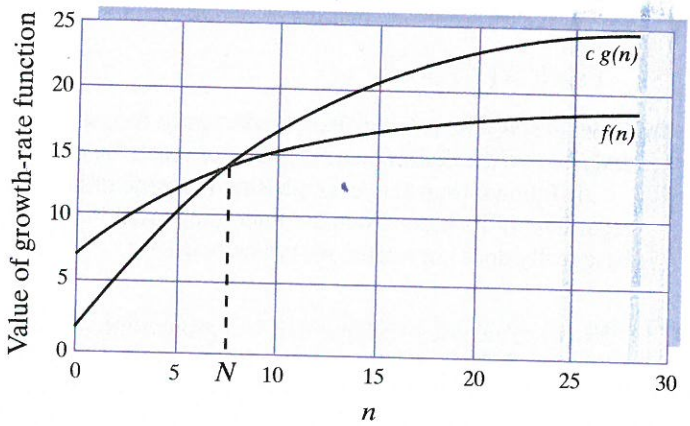
A function $f(n)$ is of order at most $g(n)$—that is, $f(n)$ is O($g(n)$)—if

- A positive real number c and positive integer N exist such that $f(n) \leq c \times g(n)$ for all $n \geq N$. That is, $c \times g(n)$ is an upper bound on $f(n)$ when n is sufficiently large.

In simple terms, $f(n)$ is $O(g(n))$ means that $c \times g(n)$ provides an **upper bound** on $f(n)$'s growth rate when n is large enough. For all data sets of a sufficient size, the algorithm will always require fewer than $c \times g(n)$ basic operations.

Figure 4-5 illustrates the formal definition of Big Oh. You can see that when n is large enough—that is, when $n \geq N$—$f(n)$ does not exceed $c \times g(n)$. The opposite is true for smaller values of n. That is unimportant, since we can ignore these values of n.

FIGURE 4-5 An illustration of the definition of Big Oh



4.14 **Example.** In Segment 4.6, we said that if an algorithm uses $5n + 3$ operations, it requires time proportional to n. We now can show that $5n + 3$ is $O(n)$ by using the formal definition of Big Oh.

When $n \geq 3$, $5n + 3 \leq 5n + n = 6n$. Thus, if we let $f(n) = 5n + 3$, $g(n) = n$, $c = 6$, and $N = 3$, we have shown that $f(n) \leq 6\ g(n)$ for $n \geq 3$, or $5n + 3 = O(n)$. That is, if an algorithm requires time directly proportional to $5n + 3$, it is $O(n)$.

Other values for the constants c and N will also work. For example, $5n + 3 \leq 5n + 3n = 8n$ when $n \geq 1$. Thus, by choosing $c = 8$ and $N = 1$, we have shown that $5n + 3$ is $O(n)$.

You need to be careful when choosing $g(n)$. For example, we just found that $5n + 3 \leq 8n$ when $n \geq 1$. But $8n < n^2$ when $n \geq 9$. So why wouldn't we let $g(n) = n^2$ and conclude that our algorithm is $O(n^2)$? Although this conclusion is correct, it is not as good—or **tight**—as it could be. You want the upper bound on $f(n)$ to be as small as possible.

Note: The upper bound on an algorithm's time requirement should be as small as possible and should involve simple functions like the ones given in Figure 4-4.

4.15 **Example.** Let's show that $4n^2 + 50n - 10$ is $O(n^2)$. It is easy to see that

$$4n^2 + 50n - 10 \leq 4n^2 + 50n \text{ for any } n$$

Since $50n \leq 50n^2$ for $n \geq 50$,

$$4n^2 + 50n - 10 \leq 4n^2 + 50n^2 = 54n^2 \text{ for } n \geq 50$$

Thus, with $c = 54$ and $N = 50$, we have shown that $4n^2 + 50n - 10$ is $O(n^2)$.

The Complexities of Program Constructs

4.17 The time complexity of a sequence of statements in an algorithm or program is the sum of the statements' individual complexities. However, it is sufficient to take instead the largest of these complexities. In general, if $S_1, S_2, \ldots, S_k$ is a sequence of program segments, and if g_i is the growth-rate function for segment S_i, the time complexity of the sequence would be $O(\max(g_1, g_2, \ldots, g_k))$, which is equivalent to $\max(O(g_1), O(g_2), \ldots, O(g_k))$.

The time complexity of the `if` statement

```
if (condition)
   S1
else
   S2
```

is the sum of the complexity of the condition and the complexity of S_1 or S_2, whichever is largest.

The time complexity of a loop is the complexity of its body times the number of times the body executes. Thus, the complexity of a loop such as

```
for i = 1 to m
   S
```

is $O(m \times g(n))$, or $m \times O(g(n))$, where $g(n)$ is the growth-rate function for S. Note that the loop variable `i` in this example increments by 1. In the following loop, `i` is doubled at each iteration:

```
for i = 1 to m, i = 2 * i
   S
```

The complexity of this loop is $O(\log(m) \times g(n))$, or $O(\log(m)) \times O(g(n))$.

Note: The complexities of program constructs

Construct	Time Complexity
Consecutive program segments $S_1, S_2, \ldots, S_k$ whose growth-rate functions are $g_1, \ldots, g_k$, respectively	$\max(O(g_1), O(g_2), \ldots, O(g_k))$
An `if` statement that chooses between program segments S_1 and S_2 whose growth-rate functions are $g_1$and g_2, respectively	$O(condition) + \max(O(g_1), O(g_2))$
A loop that iterates m times and has a body whose growth-rate function is g	$m \times O(g(n))$

Note: To show that $f(n)$ is $O(g(n))$, replace the smaller terms in $f(n)$ with larger terms until only one term is left.

Question 3 Show that $3n^2 + 2^n$ is $O(2^n)$. What values of c and N did you use?

4.16 **Example: Show that $\log_b n$ is $O(\log_2 n)$.** Let $L = \log_b n$ and $B = \log_2 b$. From the meaning of a logarithm, we can conclude that $n = b^L$ and $b = 2^B$. Combining these two conclusions, we have

$$n = b^L = (2^B)^L = 2^{BL}$$

Thus, $\log_2 n = BL = B \log_b n$ or, equivalently, $\log_b n = (1 / B) \log_2 n$ for any $n \geq 1$. Taking $c = 1 / B$ and $N = 1$ in the definition of Big Oh, we reach the desired conclusion.

It follows from this example that the general behavior of a logarithmic function is the same regardless of its base. Often the logarithms used in growth-rate functions are base 2. But since the base really does not matter, we typically omit it.

Note: The base of a logarithm in a growth-rate function is usually omitted, since $O(\log_a n)$ is $O(\log_b n)$.

Note: Identities

The following identities hold for Big Oh notation:

$O(k\, g(n)) = O(g(n))$ for a constant k

$O(g_1(n)) + O(g_2(n)) = O(g_1(n) + g_2(n))$

$O(g_1(n)) \times O(g_2(n)) = O(g_1(n) \times g_2(n))$

$O(g_1(n) + g_2(n) + \ldots + g_m(n)) = O(\max(g_1(n), g_2(n), \ldots, g_m(n))$

$O(\max(g_1(n), g_2(n), \ldots, g_m(n)) = \max(O(g_1(n)), O(g_2(n)), \ldots, O(g_m(n)))$

By using these identities and ignoring smaller terms in a growth-rate function, you can usually find the order of an algorithm's time requirement with little effort. For example, if the growth-rate function is $4n^2 + 50n - 10$,

$$\begin{aligned} O(4n^2 + 50n - 10) &= O(4n^2) && \text{by ignoring the smaller terms} \\ &= O(n^2) && \text{by ignoring the constant multiplier} \end{aligned}$$

Question 4 If $P_k(n) = a_0 n^k + a_1 n^{k-1} + \ldots + a_k$ for $k > 0$ and $n > 0$, what is $O(P_k(n))$?

Note: Other notations

Although we will use Big Oh notation most often in this book, other notations are sometimes useful when describing an algorithm's time requirement $f(n)$. We mention them here primarily to expose you to them. Beginning with the definition of Big Oh that you saw earlier, we define **Big Omega** and **Big Theta**.

- **Big Oh.** $f(n)$ is of order at most $g(n)$—that is, $f(n)$ is $O(g(n))$—if positive constants c and N exist such that $f(n) \leq c \times g(n)$ for all $n \geq N$. That is, $c \times g(n)$ is an upper bound on the time requirement $f(n)$. In other words, $f(n)$ is no larger than $c \times g(n)$. Thus, an analysis that uses Big Oh produces a maximum time requirement for an algorithm.
- **Big Omega.** $f(n)$ is of order at least $g(n)$—that is, $f(n)$ is $\Omega(g(n))$—if $g(n)$ is $O(f(n))$. In other words, $f(n)$ is $\Omega(g(n))$ if positive constants c and N exist such that $f(n) \geq c \times g(n)$ for all $n \geq N$. The time requirement $f(n)$ is not smaller than $c \times g(n)$, its **lower bound**. Thus, a Big Omega analysis produces a minimum time requirement for an algorithm.
- **Big Theta.** $f(n)$ is of order $g(n)$—that is, $f(n)$ is $\Theta(g(n))$—if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$. Alternatively, we could say that $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$. The time requirement $f(n)$ is the same as $g(n)$. That is, $c \times g(n)$ is both a lower bound and an upper bound on $f(n)$. A Big Theta analysis assures us that the time estimate is as good as possible. Even so, Big Oh is the more common notation.

Picturing Efficiency

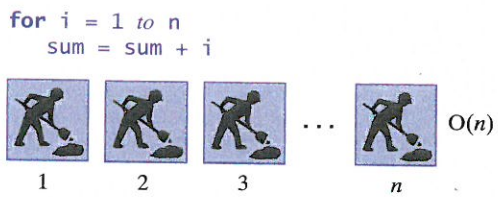
4.18 Much of an algorithm's work occurs during its repetitive phases, that is, during the execution of loops or—as you will see in Chapter 7—as a result of recursive calls. In this section, we will illustrate the time efficiency of several examples.

We begin with the loop in Algorithm A of Figure 4-1, which appears in pseudocode as follows:

```
for i = 1 to n
   sum = sum + i
```

The body of this loop requires a constant amount of execution time, and so it is O(1). Figure 4-6 represents that time with one icon, and so a row of n icons represents the loop's total execution time. This algorithm is $O(n)$: Its time requirement grows as n grows.

FIGURE 4-6 An $O(n)$ algorithm



4.19 Algorithm B in Figure 4-1 contains nested loops, as follows:

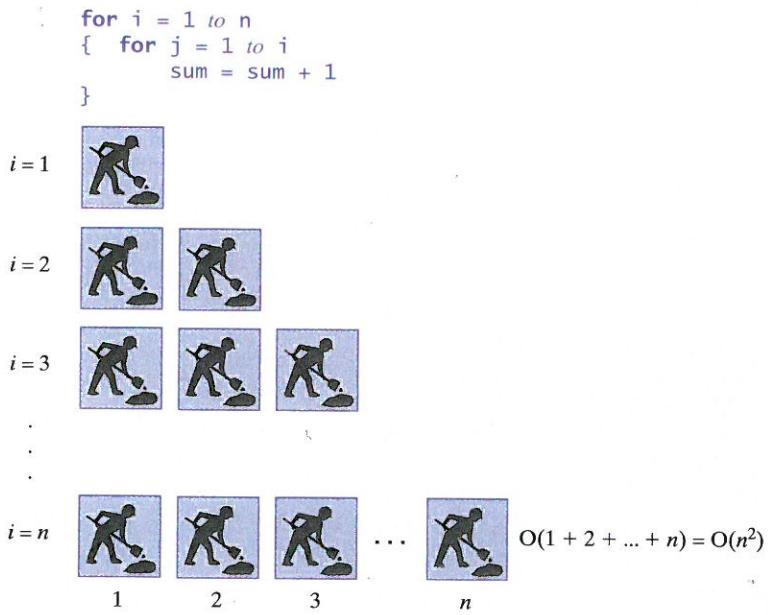
```
for i = 1 to n
{
   for j = 1 to i
      sum = sum + 1
}
```

When loops are nested, you examine the innermost loop first. Here, the body of the inner loop requires a constant amount of execution time, and so it is O(1). If we again represent that time with an icon, a row of i icons represents the time requirement for the inner loop. Since the inner loop is the body of the outer loop, it executes n times. Figure 4-7 illustrates the time requirement for these nested loops, which is proportional to $1 + 2 + \ldots + n$. Question 1 asked you to show that

$$1 + 2 + \ldots + n = n\,(n + 1) / 2$$

which is $n^2 / 2 + n / 2$. Thus, the computation is $O(n^2)$.

FIGURE 4-7 An $O(n^2)$ algorithm

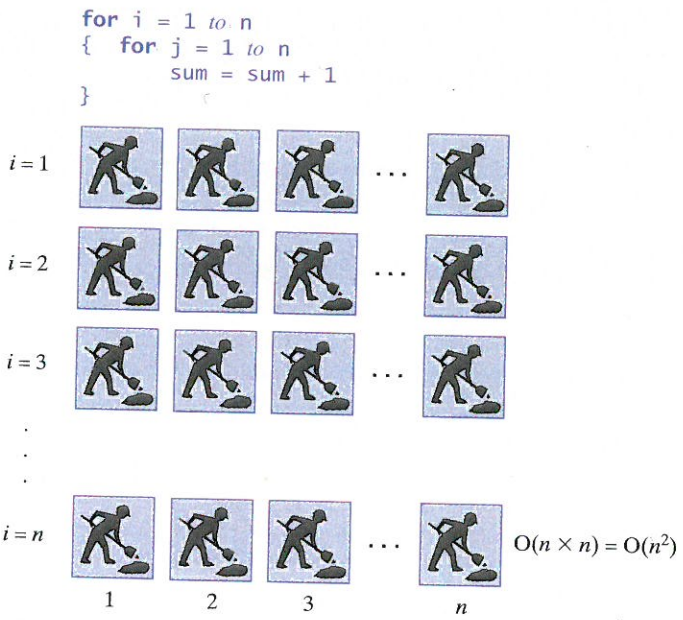


4.20 The body of the inner loop in the previous segment executes a variable number of times that depends on the outer loop. Suppose we change the inner loop so that it executes the same number of times for each repetition of the outer loop, as follows:

```
for i = 1 to n
{
   for j = 1 to n
      sum = sum + 1
}
```

Figure 4-8 illustrates these nested loops and shows that the computation is $O(n^2)$.

FIGURE 4-8 Another $O(n^2)$ algorithm



Question 5 Using Big Oh notation, what is the order of the following computation's time requirement?

```
for i = 1 to n
{
   for j = 1 to 5
      sum = sum + 1
}
```

4.21 Let's get a feel for the growth-rate functions in Figure 4-4. As we mentioned, the time requirement for an $O(1)$ algorithm is independent of the problem size n. We can apply such an algorithm to larger and larger problems without affecting the execution time. This situation is ideal, but not typical.

For other orders, what happens if we double the problem size? The time requirement for an $O(\log n)$ algorithm will change, but not by much. An $O(n)$ algorithm will need twice the time, an $O(n^2)$ algorithm will need four times the time, and an $O(n^3)$ algorithm will need eight times the time. Doubling the problem size for an $O(2^n)$ algorithm squares the time requirement. Figure 4-9 tabulates these observations.

Question 6 Suppose that you can solve a problem of a certain size on a given computer in time t by using an $O(n)$ algorithm. If you double the size of the problem, how fast must your computer be to solve the problem in the same time?

Question 7 Repeat the previous question, but instead use an $O(n^2)$ algorithm.

FIGURE 4-9 The effect of doubling the problem size on an algorithm's time requirement

Growth-Rate Function for Size n Problems	Growth-Rate Function for Size $2n$ Problems	Effect on Time Requirement
1	1	None
$\log n$	$1 + \log n$	Negligible
n	$2n$	Doubles
$n \log n$	$2n \log n + 2n$	Doubles and then adds $2n$
n^2	$(2n)^2$	Quadruples
n^3	$(2n)^3$	Multiplies by 8
2^n	2^{2n}	Squares

4.22 Now suppose that your computer can perform one million operations per second. How long will it take an algorithm to solve a problem whose size is one million? We cannot answer this question exactly without knowing the algorithm, but the computations in Figure 4-10 will give you a sense of how the algorithm's Big Oh would affect our answer. An $O(\log n)$ algorithm would take a fraction of a second, whereas an $O(2^n)$ algorithm would take many trillions of years! Note that these computations estimate the time requirement of an $O(g(n))$ algorithm as $g(n)$. Although this approximation is not universally valid, for many algorithms it is reasonable.

FIGURE 4-10 The time required to process one million items by algorithms of various orders at the rate of one million operations per second

Growth-Rate Function g	$g(10^6) / 10^6$
$\log n$	0.0000199 seconds
n	1 second
$n \log n$	19.9 seconds
n^2	11.6 days
n^3	31,709.8 years
2^n	$10^{301,016}$ years

Note: You can use $O(n^2)$, $O(n^3)$, or even $O(2^n)$ algorithms as long as your problem size is small. For example, at the rate of one million operations per second, an $O(n^2)$ algorithm would take one second to solve a problem whose size is 1000. An $O(n^3)$ algorithm would take one second to solve a problem whose size is 100. And an $O(2^n)$ algorithm would take about one second to solve a problem whose size is 20.

Question 8 The following algorithm discovers whether an array contains duplicate entries within its first n elements. What is the Big Oh of this algorithm in the worst case?

```
Algorithm hasDuplicates(array, n)
for index = 0 to n - 2
   for rest = index + 1 to n - 1
      if (array[index] equals array[rest])
         return true
return false
```

The Efficiency of Implementations of the ADT Bag

VideoNote
Comparing ADT bag implementations

We now consider the time efficiency of two of the implementations of the ADT bag that we discussed in previous chapters.

An Array-Based Implementation

One of the implementations of the ADT bag given in Chapter 2 used a fixed-size array to represent the bag's entries. We can now assess the efficiency of the bag operations when implemented in this way.

4.23 **Adding an entry to a bag.** Let's begin with the operation that adds a new entry to a bag. Segment 2.10 in Chapter 2 provided the following implementation for this operation:

```
public boolean add(T newEntry)
{
   boolean result = true;
   if (isFull())
   {
      result = false;
   }
   else
   {  // assertion: result is true here
      bag[numberOfEntries] = newEntry;
      numberOfEntries++;
   } // end if

   return result;
} // end add
```

Each step in this method—detecting whether the bag is full, assigning a new entry to an array element, and incrementing the length—is an O(1) operation. It follows then that this method is O(1). Intuitively, since the method adds the new entry right after the last entry in the array, we know the index of the array element that will contain the new entry. Thus, we can make this assignment independently of any other entries in the bag.

4.24 **Searching a bag for a given entry.** The ADT bag has a method `contains` that detects whether a bag contains a given entry. The array-based implementation of the method, as given in Segment 2.29 of Chapter 2, is:

```
public boolean contains(T anEntry)
{
   return getIndexOf(anEntry) > -1;
} // end contains
```

3. Using Big Oh notation, indicate the time requirement of each of the following tasks in the worst case.

   a. Display all the integers in an array of integers in reverse order.
   b. Display all the integers in a chain of linked nodes in reverse order.
   c. Display the n^{th} integer in an array of integers from end.
   d. Compute the sum of the first n even integers in a chain of linked nodes.

4. By using the definition of Big Oh, show that

   a. $6n^2 + 3$ is $O(n^2)$
   b. $n^2 + 17n + 1$ is $O(n^2)$
   c. $5n^3 + 100\,n^2 - n - 10$ is $O(n^3)$
   d. $3n^2 + 2^n$ is $O(2^n)$

5. Algorithm X requires $n^2 + 9n + 5$ operations, and Algorithm Y requires $5n^2$ operations. What can you conclude about the time requirements for these algorithms when n is small and when n is large? Which is the faster algorithm in these two cases?

6. Show that $O(\log_a n) = O(\log_b n)$ for $a, b > 1$. *Hint*: $\log_a n = \log_b n / \log_b a$.

7. If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, use the definition of Big Oh to show that $f(n)$ is $O(h(n))$.

8. Segment 4.9 and the chapter summary showed the relationships among typical growth-rate functions. Indicate where the following growth-rate functions belong in this ordering:

   a. $n^2 \log n$
   b. $\sqrt{n}$
   c. $n^2 / \log n$
   d. 3^n

9. Show that $7n^2 + 5$ is $O(n^2)$.

10. What is the Big Oh of the following computation?

```
boolean IsStringHello (String string)
{
   if(string.equals ("Hello"))
   {
      return true;
   }
   return false;
}
```

11. What is the Big Oh of the following computation?

```
boolean checkString(String[] strings, String st)
{
   for(int i = 0; i < strings.length; i++)
   {
      if(strings[i] == st)
      {
         return true;
      }
   }
   return false;
}
```

12. Suppose that your implementation of a particular algorithm appears in Java as follows:

```
for (int pass = 1; pass <= n; pass++)
{
   for (int index = 0; index < n; index++)
   {
      for (int count = 1; count < 10; count++)
      {
         . . .
      } // end for
   } // end for
} // end for
```

The algorithm involves an array of n items. The previous code shows the only repetition in the algorithm, but it does not show the computations that occur within the loops. These computations, however, are independent of n. What is the order of the algorithm?

13. Repeat the previous exercise, but replace 10 with n in the inner loop.

14. What is the Big Oh of `method1`? Is there a best case and a worst case?

```
public static void method1(int[] array, int n)
{
   for (int index = 0; index < n - 1; index++)
   {
      int mark = privateMethod1(array, index, n - 1);
      int temp = array[index];
      array[index] = array[mark];
      array[mark] = temp;
   } // end for
} // end method1

public static int privateMethod1(int[] array, int first, int last)
{
   int min = array[first];
   int indexOfMin = first;
   for (int index = first + 1; index <= last; index++)
   {
      if (array[index] < min)
      {
         min = array[index];
         indexOfMin = index;
      } // end if
   } // end for

   return indexOfMin;
} // end privateMethod1
```

15. What is the Big Oh of `method2`? Is there a best case and a worst case?

```
public static void method2(int[] array, int n)
{
   for (int index = 1; index <= n - 1; index++)
      privateMethod2(array[index], array, 0, index - 1);
} // end method2

public static void privateMethod2(int entry, int[] array, int begin, int end)
{
   int index;
   for (index = end; (index >= begin) && (entry < array[index]); index--)
      array[index + 1] = array[index];

   array[index + 1] = entry;
} // end privateMethod2
```