~~**Program 9-7**~~ ~~*(continued)*~~

~~**Program Output With Example Input Shown in Bold**~~

```
~~Please enter 6 integers separated by spaces.~~
~~9 4 8 6 3 1[Enter]~~

~~The unsorted values entered are:~~
~~9 4 8 6 3 1~~
~~The sorted values are:~~
~~1 3 4 6 8 9~~
```

~~Notice the similarities and differences between Program 9-7 and Program 9-4. The code in Program 9-7 that sorts vectors is almost identical to the code in Program 9-4 that sorts arrays. The differences lie in some details of initialization and argument passing.~~

~~First, notice that in Program 9-4 the array data is provided in an initialization list when the array is created, but in Program 9-7 the data to be stored in the vector is input by the user. This is done because vectors do not accept initialization lists. Second, notice that in Program 9-7 the vector is passed by reference to the `sortVector` function. This is necessary because, unlike arrays, vectors are passed by value unless the programmer uses a reference variable as a parameter. Finally, notice that in Program 9-7 it is not necessary to pass the size of the vector to the functions that work with it. This is because vectors have the `size` member function to return the number of elements in the vector.~~

9.6 Introduction to Analysis of Algorithms

CONCEPT: We can estimate the efficiency of an algorithm by counting the number of steps it requires to solve a problem.

An algorithm is a mechanical step-by-step procedure for solving a problem and is the basic strategy used in designing a program. There is often more than one algorithm that can be used to solve a given problem. For example, we saw earlier in this chapter that the problem of searching a sorted array can be solved by two different methods: *sequential search* and *binary search*.

How can we decide which of two algorithms for solving a problem is better? To answer this question, we need to establish criteria for judging the "goodness" or efficiency of an algorithm. The two criteria most often used are space and time. The *space* criterion refers to the amount of memory the algorithm requires to solve the problem, while the *time* criterion refers to the length of execution time. In this chapter, we will use the time criterion to evaluate the efficiency of algorithms.

One possibility for comparing two algorithms is to code them and then time the execution of the resulting C++ programs. This experimental approach can yield useful information, but it has the following shortcomings:

- It measures the efficiency of programs rather than algorithms.
- The results obtained depend on the programming language used to code the algorithms, and on the quality of the compiler used to generate machine code. The programs may run faster or slower if they are coded in a different language, or compiled by a different compiler.

- The results obtained depend on how the operating system executes programs, and on the nature of the hardware on which the programs are executing. The execution times may be different if we run the programs on a different computer and a different operating system.
- The results obtained apply only to those inputs that were part of the execution runs and may not be representative of the performance of the algorithms using a different set of inputs.

A better approach is to count the number of basic steps an algorithm requires to process an input of a given size. To make sense of this approach, we need more precise definitions of what we mean by computational problem, problem input, input size, and basic step.

Computational Problems and Basic Steps

A *computational problem* is a problem to be solved using an algorithm. Such a problem is a collection of *instances*, with each instance specified by input data given in some prescribed format. For example, if the problem P is to sort an array of integers, then an instance of P is just a specific integer array to be sorted. The *size* of an instance refers to the amount of memory needed to hold the input data. The input size is usually given as a number that allows us to infer the total number of bits occupied by the input data. If the number of bits occupied by each entry of the array is fixed, say at 64 bits, then the length of the array is a good measure of input size. In contrast, the length of the array is not a good measure of input size if the size of array elements can vary and there is no fixed upper bound on the size of these elements.

A step executed by an algorithm is a *basic step* (also called a *basic operation*) if the algorithm can execute the step in time bounded by a constant regardless of the size of the input. In sorting an array of integers, the step

Swap the elements in positions k and k+1

is basic because the time required to swap two array elements remains constant even if the length of the array increases. In contrast, a step such as

Find the largest element of the array

is not basic because the time required to complete the step depends on the length of the array. Intuitively, a basic step is one that could conceivably be built into the hardware of some physical computer.

The definition of a basic step does not specify the size of the constant that bounds the time required to execute the step. Ignoring the exact value of these constants reflects the reality that the same operation may be executed with different speeds on different hardware, and that an operation that can be executed with one hardware instruction on one computer may require several hardware instructions on another computer. A consequence of this definition is that we can count any constant number of basic steps as one basic step. For example, an algorithm that executes $5n$ basic steps can accurately be described as executing n basic steps.

It is important to realize that ordinary arithmetic and logic operations such as addition and comparison are not basic unless a constant bound is put on the size of the numbers being added or compared. The size of the bound does not matter as long as the bound is constant. It may be 32, 64, 128, 1024 bits, or even larger, and these operations will still be basic. In the following discussion, we assume that all the numbers used in our algorithms as inputs, outputs, or computed intermediate results are bounded in size. This allows us to consider operations on them as basic.

It only makes sense to describe an algorithm after we have described the problem the algorithm is supposed to solve. A computational problem is described by stating what the input will look like, how big it is, and what output the algorithm solving the problem is supposed to produce. These must be described clearly, so there is no ambiguity, and generally, so the algorithm can work with any data set that fits the description.

Let's look at an example. Suppose the problem *P* is to sum all the integer values in a one-dimensional array. We could describe the problem by saying that the input data is an array of *n* integer values and that the output to be produced is the integer sum of these values. Formally, this is written as follows:

INPUT: an integer array `a[]` of size *n*
SIZE OF INPUT: The number *n* of array entries
OUTPUT: An integer *sum* representing the sum total of the values stored in the array

Notice that the word INPUT used this way does not mean a set of data entered by the user, but rather means the form of the data used by the algorithm solving the problem. Likewise, the word OUTPUT used this way does not mean something displayed on the computer screen by a program. It means the result created by the algorithm that solves the problem. Because we have assumed all the array entries are of some fixed size, such as 32 or 64 bits, the number *n* of elements in the array is a good measure of input size.

Once a computational problem has been described, there can be many different algorithms designed to solve it. Some, of course, are better than others, as we will soon see. Here is one possible algorithm for solving the computational problem just described. Notice that it is expressed in pseudocode, rather than in C++ or any other particular programming language.

Algorithm 1:

```
1: sum = 0
2: k = 0 //array index
3: While k < n do
4:    sum = sum + a[k]
5:    k = k + 1
6: End While
```

Complexity of Algorithms

We can measure the complexity of an algorithm that solves a computational problem by determining the number of basic steps it requires for an input of size *n*. Let's count the number of steps required by Algorithm 1. The algorithm consists of two statements on lines 1 and 2 that are each executed once and two statements inside a loop on lines 4 and 5 that will execute once each time the loop iterates. Recall that because the statements on lines 1 and 2 perform basic operations they can be grouped together and counted as one basic operation. Let's call this operation *A*. Also, because both statements in the loop execute in constant time, independently of the size of *n*, they are also basic operations. Since the loop body contains only basic operations, the amount of time the algorithm takes to execute a *single* iteration of the loop is also constant, and not dependent on the size of n. This allows us to count each loop iteration as a single basic operation. Let's call this operation B.

Operation A executes only one time, regardless of how big n is. Operation B executes once each time the loop iterates. Because the loop iterates n times, operation B is executed n times. Thus, the total number of operations performed is $1 + n$. When $n = 10$, for example, 11 operations are performed. When $n = 1000$, 1001 operations are performed. When n = 10,000 the number of operations performed is 10,001. Notice that as n becomes large, the 1 becomes insignificant and the number of operations performed is approximately n. We therefore say that the algorithm requires execution time proportional to n to process an input set of size n.

There is another way we could look at Algorithm 1 and determine how many operations it requires. The crucial operation in summing the values in an array is the addition of each value to the variable accumulating the sum. This occurs in line 4, and there are as many additions of array values as there are loop iterations.

Thus, we could get the same result by just counting additions of array elements. It turns out that for most algorithms, it is sufficient to identify and count only one or two basic operations that are in some way crucial to the problem being solved. For example, in many array searching and sorting algorithms, it is sufficient to just count the number of comparisons between array elements.

The array-summing algorithm just considered is particularly simple to analyze because it performs the same amount of work for all input sets of a given size.

This is not the case with all algorithms. Consider the linear search algorithm introduced earlier in this chapter. It searches through an array of values, looking for one that matches a search key. Let's call the key X. The input to the algorithm is the array of n values and the key value X. The output of the algorithm is the subscript of the array location where the value was located or, if it is not found, the determination that the loop control variable has become larger than the subscript of the last array element. Formally, the problem can be stated like this:

INPUT: An integer array $a[\ ]$ of size n, and an integer X
SIZE OF INPUT: The number n of array entries
OUTPUT: An integer k in the range $0 \le k \le n - 1$ such that $a[k] = X$, or $k = n$

Algorithm 2, shown here, uses the linear search algorithm to solve the problem.

Algorithm 2:
```
1: k = 0
2: While k < n and a[k] ≠ X do
3:     k = k + 1
4: End While
```

This algorithm starts at one end and searches sequentially through the array. The algorithm stops as soon as it encounters X, but will search the entire array if X is not in the array. The algorithm may stop after making only one comparison (X is found in the first entry examined), or it may not stop until it has made n comparisons (X is found in the last place examined or is not in the array). In fact, the algorithm may perform m comparisons where m is any value from 1 to n. In cases where an algorithm may perform different amounts of work for different inputs of the same size, it is common to measure the efficiency of the algorithm by the work done on an input of size n that requires the *most* work. This is called measuring the algorithm by its worst-case complexity function.

Worst Case Complexity of Algorithms

The *worst-case complexity function* $f(n)$ of an algorithm is the number of steps it performs on an input of size n that requires the most work. It gives an indication of the longest time the algorithm will ever take to solve an instance of size n, and is a good measure of efficiency to use when we are looking for a performance guarantee.

Let's determine the worst-case complexity of binary search, which was introduced earlier in this chapter. This algorithm is used to locate an item X in an array sorted in ascending order. The worst case occurs when X is not found in the array. In this case, as we will see, the algorithm performs $L + 1$ steps, where L is the number of loop iterations.

Here is the binary search algorithm to search an array of n elements.

Algorithm 3:

```
first = 0
last = n - 1    // n - 1 is the subscript of the last element.
found = false
position = -1
While found is not true and first <= last
   middle = (first + last) / 2
   If a[middle] = X
      found = true
      position = middle
   Else if a[middle] > X
      last = middle - 1
   Else
      first = middle + 1
   End If
End While
// When the loop terminates, position holds the subscript
// where the value matching X was found, or holds –1 if
// the value was not found.
```

The algorithm consists of some initialization of variables followed by a loop. The initialization requires constant time, and can therefore be considered to be one basic operation. Likewise, each iteration of the loop is a basic step because increasing the number of entries in the array does not increase the amount of time required by a single iteration of the loop. This shows that the number of steps required by binary search is $L + 1$. Now L is approximately equal to the integer part of $\log_2 n$, the logarithm of n to the base 2. To see this, notice that the size of the array to be searched is initially n, and each iteration reduces the size of the remaining portion of the array by approximately one half. Because each loop iteration performs at most two comparisons, binary search performs a total of $2 \log_2 n$ comparisons. We can summarize our findings as follows:

In the worst case, binary search requires time proportional to $\log_2 n$.

Let's look at one more algorithm to determine its worst case complexity. The computational problem to be solved is to arrange a set of n integers into ascending order.

INPUT: An array $a[\ ]$ of n integers
SIZE OF INPUT: The number n of array entries
OUTPUT: The array $a[\ ]$ rearranged so that $a[0] \leq a[1] \leq \ldots \leq a[n-1]$

The algorithm we will use is a modification of the selection sort algorithm introduced earlier in this chapter. This version scans for the largest element (instead of the smallest) and moves it to the end in each pass.

Algorithm 4:

```
1:  For (k = n-1; k ≥ 1; k --)
2:      // a[0..k] is what remains to be sorted
3:      Determine position p of largest entry in a[0..k]
4:         Swap a[p] with a[k]
5:  End For
```

To analyze the complexity of this algorithm, let's begin by determining the number of array entry comparisons it makes when sorting an array of n entries. These comparisons occur in step 3. Step 3 is clearly not a basic step, as it requires time proportional to k, and k varies with each iteration of the loop. To better see what is going on, let's restate step 3 using operations that are basic.

INPUT: array $a[0..k]$ of $k + 1$ entries
SIZE OF INPUT: number $k + 1$ of array entries

```
3.0:  p = 0 //Position of largest value in unsorted part of the array
3.1:  For (m = 1; m ≤ k; m ++)
3.2:     If a[m] > a[p] Then
3.3:       p = m
3.4:    End if
3.5:  End For
```

We can see that the loop in line 3.1 through 3.5 iterates k times and on line 3.2 makes one comparison each time it iterates. Therefore this algorithm requires k comparisons between array entries.

Now returning to the main sorting algorithm, we observe that there will be $n - 1$ iterations of the loop that starts at line 1 and ends at line 5, one iteration for each value of k in the range $n - 1$ to 1. On the first iteration, k equals $n - 1$, so step 3, as we learned from the analysis of lines 3.0 through 3.5, performs $n - 1$ comparisons between array elements. On the second iteration, k equals $n - 2$, so step 3 performs $n - 2$ comparisons. This continues until, on the final iteration, k equals 1, and step 3 performs 1 comparison. Here is what it looks like:

$k = n - 1$: step 3 performs $n - 1$ comparisons
$k = n - 2$: step 3 performs $n - 2$ comparisons
.
.
$k = 1$: step 3 performs 1 comparison

Generalizing, we can thus say that for every value of k from $n - 1$ to 1, on the kth iteration, the step on line 3 will perform k comparisons.

Thus the total number of comparisons performed by this simple sorting algorithm is given by the expression

$1 + 2 + 3 + \ldots + (n - 1) = (n - 1)n/2$

For large n, this expression is very close to $n^2 / 2$. So we say that:

In the worst case, selection sort requires time proportional to n^2.

Average Case Complexity

The worst-case complexity does not, however, give a good indication of how an algorithm will perform in practical situations where inputs that yield worst case performance are rare. Often we are more interested in determining the complexity of the typical, or average case. The *average-case complexity function* can be used when we know the relative frequencies with which different inputs are likely to occur in practice. The average case complexity function uses these frequencies to form a weighted average of the number of steps performed on each input. Unfortunately, although it yields a good measure of the expected performance of an algorithm, accurate estimates of input frequencies may be difficult to obtain.

Asymptotic Complexity and the Big O Notation

We can compare two algorithms F and G for solving a problem by comparing their complexity functions. More specifically, if $f(n)$ and $g(n)$ are the complexity functions for the two algorithms, we can compare the algorithms against each other by looking at what happens to the ratio $f(n)/g(n)$ when n gets large. This is easiest to understand if this ratio tends to some limit. Let us consider some specific examples. Throughout, we assume that $f(n) \geq 1$ and $g(n) \geq 1$ for all $(n) \geq 1$.

- $f(n) = 3n^2 + 5n$ and $g(n) = n^2$. In this case

$$\frac{f(n)}{g(n)} = \frac{3n^2 + 5n}{n^2} = 3 + \frac{5}{n} \to 3 \text{ as } n \to \infty$$

That is, the value of $f(n)/g(n)$ gets closer and closer to 3 as n gets large. What this means is that for very large input sizes F performs three times as many basic operations as G. However, because the two algorithms differ in performance only by a constant factor, we consider them to be equivalent in efficiency.

- $f(n) = 3n^2 + 5n$ and $g(n) = 100n$. In this case

$$\frac{f(n)}{g(n)} = \frac{3n^2 + 5n}{100n} = \frac{3n}{100} + \frac{5}{100} \to \infty \text{ as } n \to \infty$$

Here, the ratio $f(n)/g(n)$ gets larger and larger as n gets large. This means F does a lot more work than G on large input sizes. This makes G the better algorithm for large inputs.

- $f(n) = 3n^2 + 5n$ and $g(n) = n^3$. In this case

$$\frac{f(n)}{g(n)} = \frac{3n^2 + 5n}{n^3} = \frac{3}{n} + \frac{5}{n^2} \rightarrow 0 \text{ as } n \rightarrow \infty$$

This means that for large inputs the algorithm *G* is doing a lot more work than *F*, making *F* the more efficient algorithm.

In general, we can compare two complexity functions $f(n)$ and $g(n)$ by looking at what happens to $f(n)/g(n)$ as n gets large. Although thinking in terms of a limit of this ratio is helpful in comparing the two algorithms, we cannot assume that such a limit will always exist. It turns out that a limit does not have to exist for us to gain useful information from this ratio. We can usefully compare the two complexity functions if we can find a positive constant *K* such that

$$\frac{f(n)}{g(n)} \leq K \text{ for all } n \geq 1$$

If this can be done, it means that the algorithm *F* is no worse than *K* times *G* for large problems. In this case, we say that $f(n)$ is in $O(g(n))$, pronounced "*f* is in Big O of *g*." The condition that defines $f(n)$ is in $O(g(n))$ is often written like this

$$f(n) \leq Kg(n) \text{ whenever } n \geq 1.$$

Showing that $f(n)$ is in $O(g(n))$ is usually straightforward. You look at the ratio $f(n)/g(n)$ and try to find a positive constant *K* that makes $f(n)/g(n) \leq K$ for all $n \geq 1$. For example, to show that $3n^2 + 5n$ is in $O(n^2)$, look at the ratio

$$\frac{3n^2 + 5n}{n^2} = 3 + \frac{5}{n}$$

and notice that $5/n$ will be at most 5 for all $n \geq 1$. So $3 + 5/n \leq 8$. Therefore for $K = 8$, $f(n) / g(n) \leq K$.

To show that $f(n)$ is not in $O(g(n))$, you have to show that there is no way to find a positive *K* that will satisfy $f(n) / g(n) \leq K$ for all $n \geq 1$. For example, the function $3n^2 + 5n$ is not in $O(n)$ because there is no constant *K* that satisfies

$$\frac{3n^2 + 5n}{n} = 3n + 5 \leq K \text{ for all } n \geq 1.$$

Although defined for functions, the "Big O" notation and terminology is also used to characterize algorithms and computational problems. Thus, we say that an algorithm *F* is in $O(g(n))$ for some function $g(n)$ if the worst case complexity function $f(n)$ of *F* is in Big O of $g(n)$. Accordingly, sequential search of an array is in $O(n)$ whereas binary search is in $O(\log_2 n)$.

Similarly, a computational problem is said to be in $O(g(n))$ if there exists an algorithm for the problem whose worst case complexity function is in $O(g(n))$. Thus, the problem of sorting an array is in $O(n^2)$, whereas the problem of searching a sorted array is in $O(\log_2 n)$.

If $g(n)$ is a function, $O(g(n))$ can be regarded as a family of functions that grow no faster than $g(n)$. These families are called *complexity classes,* and a few of them are important enough to merit specific names. We list them here in order of their rate of growth:

1. O(1): A function $f(n)$ is in this class if there is a constant $K > 0$ such that $f(n) \leq K$ for all $n \geq 1$. An algorithm whose worst case complexity function is in this class is said to run in *constant time*.
2. $O(\log_2 n)$: Algorithms in this class run in *logarithmic time*. Because $\log n$ grows much slower than n, a huge increase in the size of the problem results in only a small increase in the running time of the algorithm. This complexity is characteristic of search problems that eliminate half of the search space with each basic operation. The *binary search* algorithm is in this class.
3. $O(n)$: Algorithms in this class run in *linear time*. Any increase in the size of the problem results in a proportionate increase in the running time of the algorithm. This complexity is characteristic of algorithms like *sequential search* that make a single pass, or a constant number of passes, over their input.
4. $O(n \log_2 n)$: This class is called "$n \log n$" *time*. An increase in the size of the problem results in a slight increase in the running time of the algorithm. The average case complexity of *Quicksort,* a sorting algorithm you will learn about in Chapter 14, is in this class.
5. $O(n^2)$: This class is called *quadratic time*. This performance is characteristic of algorithms that make multiple passes over the input data using two nested loops. An increase in the size of the problem causes a much greater increase in the running time of the algorithm. The worst case complexity functions of *bubble sort, selection sort,* and *Quicksort* all lie in this class.

Checkpoint

9.10 What is a basic operation?

9.11 What is the worst case complexity function of an algorithm?

9.12 One algorithm needs $10n$ basic operations to process an input of size n, and another algorithm needs $25n$ basic operations to process the same input. Which of the two algorithms is more efficient? Or are they equally efficient?

9.13 What does it mean to say that $f(n)$ is in $O(g(n))$?

9.14 Show that $100n^3 + 50n^2 + 75$ is in $O(20n^3)$ by finding a positive K that satisfies the equation $(100n^3 + 50n^2 + 75) / 20n^3 \leq K$.

9.15 Assuming $g(n) \geq 1$ for all $n \geq 1$, show that every function in $O(g(n) + 100)$ is also in $O(g(n))$.

~~9.7 Case Studies~~

~~The following case studies, which contain applications of material introduced in Chapter 9, can be found on the student CD.~~

~~Demetris Leadership Center—Parts 1 & 2~~

~~Chapter 9 included programs illustrating how to search and sort arrays, including arrays of objects. These two case studies illustrate how to search and sort arrays of structures. Both studies develop programs for DLC, Inc., a fictional company that publishes books, DVDs, and audio CDs. DLC's inventory data, used by both programs, is stored in an array of structures.~~

~~Creating an Abstract Array Data Type—Part 2~~

~~The `IntList` class, begun as a case study in Chapter 8, is extended to include array searching and sorting capabilities.~~