CHAPTER 15

Running Time Analysis

CHAPTER CONTENTS

Introduction

Today's Internet websites have millions of users. With the success of Web 2.0 Internet sites, the databases storing data on the web servers have grown in size dramatically, to accommodate both the growing number of users and the growing volume of data that is posted by these users.

Scientific applications have also experienced a data explosion. Sensors used in these scientific applications, such as meteorology or fluid mechanics, are becoming more precise at the same time they are getting cheaper. More and more sensors are being used, and application programs have to manage more and more data.

Programs that handle and manipulate this ever-increasing amount of data need to use algorithms that are well-designed and efficient so that they minimize waiting time for users. Two programs that solve the same problem by using two different algorithms can result in two completely different levels of performance—everything else, in particular the hardware platform, being equal. For example, two search engines performing the same search could run at two different speeds: one could return results in tenths of a second while the other could take several seconds to return results.

Most programmers tend to disregard speed and space (memory utilization) issues when writing code. They rely on increasing hardware performance to solve speed problems and the decreasing cost of memory to solve space problems. However, with the data explosion and the resulting data processing issues that we are experiencing today across many industries, designing efficient algorithms has become more and more important. In this chapter, we will focus on algorithms' speed performance rather than space utilization.

When we measure the performance of an algorithm, we use the expression **running time**. We cannot predict a single, precise running time for many algorithms, because the amount of processing depends in large part on the number of inputs and the values of those inputs. So we express the running time of an algorithm as a mathematical function of its inputs. This allows us to compare the relative performance of multiple algorithms. For example, the running time for computing the factorial of an integer varies according to the value for which we are computing the factorial. Factorials of larger numbers require more processing to compute than factorials of smaller numbers. If we can express the running time of multiple algorithms that compute a factorial as a function of their input, then we can compare the relative efficiency of each algorithm. In other cases, such as sorting an array of integers, the running time depends on the number of array elements. Similarly, if we express the running time as a function of the number of elements in the array, we can compare the relative efficiency of multiple sorting algorithms.

The input value or number of inputs for an algorithm represents the size of the problem for which we are trying to compute the running time. We will call that number ***n***. We are interested in relative time, independently of the hardware platform used, not absolute time. Furthermore, we are typically interested in the order of magnitude of the algorithm, rather than a precise mathematical expression as a function of n. Indeed, if n is very large (for example, 1 million or more), performance does not vary noticeably if the algorithm takes n steps or $n + 17$ steps to complete.

However, if an algorithm has a running time expressed as n^2, then the number of inputs has a big impact on performance. For example, we can predict that 10 inputs will require the execution of 100 statements and 1,000 inputs will require the execution of 1 million statements.

The objectives of this chapter are:

- To be able to evaluate the running time of a given algorithm through various methods
- To understand that how we code an algorithm directly impacts its running time

15.1 Orders of Magnitude and Big-Oh Notation

Table 15.1 shows examples of various orders of magnitude for an algorithm as a function of the number of inputs n, along with the corresponding number of statement executions for different values of n.

Let's look at an example to see how you can use these values. Sequential Search has a running time of n, and Binary Search has a running time of *log n*. Thus, if we are searching an array of 1 million users for a particular user name, a Sequential Search will take, on average, the execution of an order of 1 million statements, while a Binary Search will require the execution of

TABLE 15.1 Comparisons of Various Functions Representing Running Times

Order of Magnitude	Number of Statements Executed			
	$n = 10$	$n = 20$	$n = 1{,}000$	$n = 1$ million
$\log n$	2.23	3.23	Approx. 10	Approx. 20
n	10	20	1000	10^6
$n \log n$	22.3	64.6	Approx. 10,000	Approx. $20 * 10^6$
n^2	100	400	10^6	10^{12}
n^3	1,000	8,000	10^9	10^{18}
2^n	1,024	Approx. 10^6	Approx. 10^{300}	Approx. 10^{300000}

only 20 statements. Remember, however, that for a Binary Search to work, the array must already be sorted. Later in this chapter, we will discuss how to compute these running times.

REFERENCE POINT

Sequential Search and Binary Search are explained in Chapter 8.

As you can see from the table, algorithms that have a running time where n is the exponent of the function, such as 2^n, take a very large number of statement executions and are very slow; they should be used only if no better algorithm can be found.

Running times of algorithms are often represented using the **Big-Oh** or the **Big-Theta** notation, as in $O(n)$ or $\Theta(n^2)$, for example. The mathematical definition of Big-Theta is as follows:

A function $f(n)$ is Big-Theta of another function $g(n)$, or $\Theta(g(n))$, if and only if:

1. $f(n)$ is **Big-Omega** of $g(n)$, or $\Omega(g(n))$, i.e., there exist two positive constants, $n1$ and $c1$, such that for any $n >= n1$, $f(n) >= c1 * g(n)$.

   In other words, for n sufficiently big, $g(n)$ is a lower bound of $f(n)$; that is, $g(n)$ is smaller than $f(n)$, if we ignore the constants.

and

2. $f(n)$ is Big-Oh of $g(n)$, or $O(g(n))$, i.e., there exist two positive constants, $n2$ and $c2$, such that for any $n >= n2$, $f(n) <= c2 * g(n)$.

   In other words, for n sufficiently big, $g(n)$ is an upper bound of $f(n)$; that is, $g(n)$ is bigger than $f(n)$, if we ignore the constants.

It has become common in the industry to say Big-Oh instead of Big-Theta. Indeed, we are really interested in an upper bound running time (Big-Oh), and as tight an upper bound as possible (Big-Theta).

Although the preceding definition may sound a bit complex, when trying to estimate the Big-Oh of a particular function representing a running time, the following rules can be used:

- Keep only the dominant term, i.e., the term that grows the fastest as n grows.
- Ignore the coefficient of the dominant term.

Table 15.2 shows a few examples illustrating these rules.

As an example, we will show that the function $f(n) = 3 * n^2 + 6 * n + 12$ is $\Theta(n^2)$.

First we show that $f(n)$ is $\Omega(n^2)$:

For $n >= 0$,

$f(n) = 3 * n^2 + 6 * n + 12 >= 3 * n^2$

So if we choose $n1 = 0$ and $c1 = 3$, we just proved by definition that $f(n)$ is $\Omega(n^2)$.

Now we show that the same function $f(n)$ is $O(n^2)$.

For $n >= 1$, we can rewrite $f(n)$ as

$f(n) = n^2 * (3 + 6 / n + 12 / n^2)$

TABLE 15.2 Examples of Functions Representing Running Times and Their Respective Big-Oh

$f(n)$	Dominant Term	Big-Oh
$2 * n + 19$	$2 * n$	$O(n)$
$3 * n^2 + 6 * n + 12$	$3 * n^2$	$O(n^2)$
$n^3 + 9 * n^2 + 5 * n + 2$	n^3	$O(n^3)$
$3 * 2^n + 5 * n^3 + 3 * n + 7$	$3 * 2^n$	$O(2^n)$
$n + 7 * \log n$	n	$O(n)$
$2 * n * \log n + 8 * n + \log n + 8$	$2 * n * \log n$	$O(n * \log n)$
$3 * \log n + 35$	$3 * \log n$	$O(\log n)$

For $n >= 6$, we have

$6 / n <= 1$ and $12 / n^2 < 1$

therefore,

$f(n) <= n^2 * (3 + 1 + 1) = 5 * n^2$

So if we choose $n2 = 6$ and $c2 = 5$, we just proved by definition that $f(n)$ is $O(n^2)$.

Since $f(n)$ is both Big-Omega(n^2) and Big-Oh(n^2), then $f(n)$ is Big-Theta(n^2).

To show that a polynomial function is Big-Oh of its most dominant term, we simply factor by the most dominant term as follows:

For $n > 0$,

$f(n) = a_p\, n^p + a_{p-1}\, n^{p-1} + \ldots + a_2\, n^2 + a_1\, n + a_0$ where a_p is strictly positive

$f(n) = a_p\, n^p\, (1 + (a_{p-1} / a_p)\, 1 / n + \ldots + (a_2 / a_p)\, 1 / n^{p-2} + (a_1 / a_p)\, 1 / n^{p-1} + (a_p / a_0)\, 1 / n^p)$

$f(n) <= a_p\, n^p\, (1 + |(a_{p-1} / a_p)|\, 1 / n + \ldots + |(a_2 / a_p)|\, 1 / n^{p-2} + |(a_1 / a_p)|\, 1 / n^{p-1} + |(a_p / a_0)|\, 1 / n^p)$

All a_i's are constants; let M be the maximum of all $|(a_i / a_p)|$.

Thus,

$f(n) <= a_p\, n^p\, (1 + M\, 1 / n + \ldots + M\, 1 / n^{p-2} + M\, 1 / n^{p-1} + M\, 1 / n^p)$

$f(n) <= a_p\, n^p\, (1 + M\, (1 / n + \ldots + 1 / n^{p-2} + 1 / n^{p-1} + 1 / n^p))$

$f(n) <= a_p\, n^p\, (1 + M\, (-1 + 1 + 1 / n + \ldots + 1 / n^{p-2} + 1 / n^{p-1} + 1 / n^p))$

since we know mathematically that

$1 + a + a^2 + \ldots + a^p = \Sigma\, a^i$ from $i = 0$ to p is equal to $(1 - a^{p+1}) / (1 - a)$ for a different from 1.

Using $a = 1/n$, we get

$f(n) <= a_p\, n^p\, (1 + M\, (-1 + (1 - 1 / n^{p+1}) / (1 - 1 / n)))$

$f(n) <= a_p\, n^p\, (1 + M\, (-1 + (1 - 1 / n^{p+1}) * (n / (n - 1))))$

Thus,

$f(n) <= a_p\, n^p\, (1 + M\, (-1 + (n / (n - 1))))$

$f(n) <= a_p\, n^p\, (1 + M\, ((-n + 1 + n) / (n - 1)))$

$f(n) <= a_p\, n^p\, (1 + M\, (1 / (n - 1)))$

Thus,

$f(n) <= a_p\, n^p\, (1 + M)$ for $n >= 2$

choosing $n_0 = 2$ and $c_0 = a_p\,(1 + M)$.

For $n >= n_0$, we have

$f(n) <= c_0\, n^p$

and therefore,

$f(n)$ is $O(n^p)$, i.e., $f(n)$ is Big-Oh of its most dominant term.

15.2 Running Time Analysis of Algorithms: Counting Statements

One simple method to analyze the running time of a code sequence or a method is simply to count the number of times each statement is executed and to calculate a total count of statement executions.

Example 15.1 is a method that calculates the total value of all the elements of an array of size n and returns the sum.

```
public static int addElements( int [ ] arr )
{
  int sum = 0;                  // ( 1 )
  int i = 0;                    // ( 2 )
  while ( i < arr.length )      // ( 3 )
  {
     sum += arr[i];             // ( 4 )
     i++;                       // ( 5 )
  }
  return sum;                   // ( 6 )
}
```

EXAMPLE 15.1 A Single Loop

Let's count how many times each statement is executed.

Assuming the array has n elements, we can develop the following analysis:

Statement	# Times Executed
(1)	1
(2)	1
(3)	$n + 1$
(4)	n
(5)	n
(6)	1

Note that the loop condition, $i < arr.length$, is executed one more time than each statement of the loop body: when i is equal to *arr.length*, we evaluate the loop condition, but we exit the loop and thus do not execute the two statements in the loop body. Thus, the total number of statements executed, $T(n)$, is equal to:

$$\begin{aligned} T(n) &= 1 + 1 + (n+1) + n + n + 1 \\ &= 3n + 4 \\ &= O(n) \end{aligned}$$

So we can say that the running time of the *addElements* method is O(n). Note that in the end, we do not need an exact count of the statements executed, since we are really interested in the Big-Oh running time of the function.

REFERENCE POINT

Multidimensional arrays are explained in Chapter 9.

Example 15.2 is a method that determines the maximum value in a two-dimensional array of *ints*.

```
public static int calculateMaximum( int [ ][ ] arr )
{
  int maximum  = arr[0][0];                        // ( 1 )
  for ( int i = 0; i < arr.length; i++ )           // ( 2 )
  {
   for ( int j = 0; j < arr[i].length; j++ )  // ( 3 )
   {
    if ( maximum < arr[i][j] )                    // ( 4 )
         maximum  = arr[i][j];                     // ( 5 )
   }
  }
  return maximum;                                  // ( 6 )
}
```

EXAMPLE 15.2 A Double Loop

Let's count how many times each statement is executed. Assuming the array has n rows and each row has n columns, we can develop the following analysis:

Statement	# Times Executed
(1)	1
(2)	$1 + (n+1) + n = 2 * n + 2$
(3)	$n * (1 + (n+1) + n) = 2 * n^2 + 2 * n$
(4)	$n * n = n^2$
(5)	between 0 and $n * n$
(6)	1

Statement (2) actually contains three statements: *int* $i = 0$ is executed 1 time, $i <$ *arr.length* is executed $(n + 1)$ times as i goes from 0 to n, and i++ is executed n times as i is incremented n times.

In evaluating the number of times statements (3), (4), and (5) will be executed, we first note that we will enter the outer loop n times. Statement (3) also contains three statements: *int* $j = 0$ is executed each time we enter the outer loop, or n times; $j <$ *arr.length*[i] is executed $(n + 1)$ times each time we enter the outer loop, or $n * (n + 1)$ times, as j goes from 0 to n; and j++ is executed n times each time we enter the outer loop, or $n * n$ times.

Since we enter the outer loop n times and for each outer loop iteration, we enter the inner loop n times, statement (4) will be executed $n * n$ times. As for statement (5), it will be executed once each time the Boolean expression *maximum* < *arr*[i][j] evaluates to *true*. We cannot tell how many times that will happen, but we can tell that it will happen no more than $n * n$ times. We will call this unknown value x.

Thus, the total number of statements executed, $T(n)$, is equal to:

$$\begin{aligned} T(n) &= 1 + (2 * n + 2) + (2 * n^2 + 2 * n) + (n^2) + x + 1 \\ &= 3 * n^2 + 4 * n + 4 + x \end{aligned}$$

with $x <= n * n$

Furthermore, since the value of x is between 0 and n^2,

$$3 * n^2 + 4 * n + 4 <= T(n) <= 3 * n^2 + 4 * n + 4 + n^2$$

$$3 * n^2 + 4 * n + 4 <= T(n) <= 4 * n^2 + 4 * n + 4$$

since $T(n)$ has both lower and upper bounds that are $O(n^2)$, $T(n)$ is $O(n^2)$.

REFERENCE POINT

Sequential Search is explained in Chapter 8.

For our third example, let's compute the running time of a Sequential Search, implemented by the code shown in Example 15.3.

```
public int sequentialSearch( int [ ] array, int key )
{
  for ( int i = 0; i < array.length; i++ )        // ( 1 )
    if ( array[ i ] == key )                      // ( 2 )
      return i;                                   // ( 3 )
  return -1;                                      // ( 4 )
}
```

EXAMPLE 15.3 Sequential Search Algorithm

Let's count how many times each statement is executed. Assuming the array has n elements, we can develop the following analysis:

Statement	# Times Executed
(1)	1 + (between 1 and $(n + 1)$) + (between 0 and n)
(2)	between 1 and n
(3)	0 or 1
(4)	1 or 0

Thus, if $T(n)$ represents the total number of statements executed, we can say that

$$1 + (1) + (0) + 1 + 1 \;<=\; T(n) \;<=\; 1 + (n + 1) + n + n + 1$$

$$4 \;<=\; T(n) \;<=\; 3n + 3$$

$T(n) <= 3n + 3$ shows that $T(n)$ is $O(n)$.

However, we cannot really tell, from the coding of the function, how many statements will be executed as a function of n. In these situations, it is interesting to consider three running times:

- the worst-case running time
- the best-case running time
- the average-case running time

In the worst case, where the search key is not found in the array or it is found in the last element, $T(n) = 3n + 3$, and therefore $T(n)$ is $O(n)$, as mentioned earlier.

In the best case, the element we are looking for is at index 0 of the array and only four statements will be executed, independently of the value of n. Thus, the best-case running time is $O(1)$ since we do not take the multiplying constant into consideration when we compute a Big-Oh.

In the average case, we find the element we are looking for in the middle of the array, and the value of $T(n)$ will be

$$\begin{aligned} T(n) &= 1 + (n + 1)/2 + n/2 + n/2 + 1 \\ &= 3n/2 + 2\tfrac{1}{2} \\ &= O(n) \end{aligned}$$

15.3 Running Time Analysis of Algorithms and Impact of Coding: Evaluating Recursive Methods

In this section, we will learn how to compute the running time of a recursive method. We will also look at how coding a method has a direct impact on its running time.

Consider coding a recursive method that takes one parameter, *n*, and returns *2ⁿ*. There are several ways to code that method, and we will consider two of them here so that we can assess which algorithm is more efficient.

REFERENCE POINT

Recursion is explained in Chapter 13.

Our first method, *powerOf2A*, is designed using these two facts:

- when $n = 0$, $2^0 = 1$. This is our base case.
- $2^n = 2 * 2^{n-1}$. This is our general case.

This first problem formulation results in the method shown in Example 15.4.

```
public static int powerOf2A( int n ) // n >= 0
{
   if ( n == 0 )
       return 1;
   else
       return 2 * powerOf2A( n - 1 );
}
```

EXAMPLE 15.4 First Recursive Formulation of 2^n

Our second method, *powerOf2B*, is designed using these two facts:

- when $n = 0$, $2^0 = 1$. This is our base case.
- $2^n = 2^{n-1} + 2^{n-1}$. This is our general case.

This second problem formulation results in the method shown in Example 15.5.

```
public static int powerOf2B( int n ) // n >= 0
{
   if ( n == 0 )
       return 1;
   else
       return powerOf2B( n - 1 ) + powerOf2B( n - 1 );
}
```

EXAMPLE 15.5 Second Recursive Formulation of 2^n

Let's compute the running time of *powerOf2A* as a function of the input n; we will call it $T1(n)$.

In the base case (n is equal to 0), *powerOf2A* makes only one comparison and returns 1. Thus,

$$T1(0) = 1$$

Generally, since it takes $T1(n)$ to compute and return *powerOf2A*(n), then it takes $T1(n-1)$ to compute and return *powerOf2A*($n-1$).

Thus, in the general case, the comparison in the *if* statement will cost us 1 instruction; computing and returning *powerOf2A*($n-1$) will cost us $T1(n-1)$; and multiplying that result by 2 will cost us 1 instruction. Thus, the total time $T1(n)$ can be expressed as follows:

$$\begin{aligned} T1(n) &= 1 + T1(n-1) + 1 \\ &= T1(n-1) + 2 \quad \text{// Equation 15.1} \end{aligned}$$

The preceding equation, which we will call Equation 15.1, is called a recurrence relation between $T1(n)$ and $T1(n-1)$ because $T1(n)$ is expressed as a function of $T1(n-1)$.

From there, we can use a number of techniques to compute the value of $T1(n)$ as a function of n.

Handwaving Method

This method is called handwaving because it is more an estimation method, rather than a method based on strict mathematics.

From the preceding recurrence relation, we can say that it costs us two instructions to go down one step (from n to $n-1$). Therefore, to go down n steps will cost us $2 * n$ instructions. We then add one instruction for $T(0)$, and get

$$T1(n) = 2 * n + 1$$

Iterative Method

This method involves iterating several times, starting with the recurrence relation until we can identify a pattern. In general, we can say that

$$T1(x) = T1(x-1) + 2, \text{ where } x \text{ is some integer} \quad \text{// Equation 15.2}$$

We call this Equation 15.2, which is the same as Equation 15.1, except that x has been substituted for n.

We now want to express $T(n)$ as a function of $T(n-2)$; thus, we want to replace $T(n-1)$ in Equation 15.1 by an expression using $T(n-2)$.

Substituting $n-1$ for x in Equation 15.2, we get

$T1(n-1) = T1(n-2) + 2$

Plugging in the value of $T1(n-1)$ into Equation 15.1, we get

$T1(n) = T1(n-2) + 2 + 2$

$= T1(n-2) + 2 * 2$ // Equation 15.3

Note that in Equation 15.3, we do not simplify 2 * 2. In this way, we are trying to let a pattern develop so we can easily identify it.

SOFTWARE ENGINEERING TIP

When trying to develop and identify a pattern using iteration, do not precisely compute all the terms. Instead, leave them as patterns.

Using $x = n - 2$ in Equation 15.2, we get

$T1(n-2) = T1(n-3) + 2$

Plugging in the value of $T1(n-2)$ into Equation 15.3, we get

$T1(n) = T1(n-3) + 2 + 2 * 2$

$= T1(n-3) + 2 * 3$ // Equation 15.4

Using $x = n - 3$ in Equation 15.2, we get

$T1(n-3) = T1(n-4) + 2$

Plugging in the value of $T1(n-3)$ into Equation 15.4, we get

$T1(n) = T1(n-4) + 2 + 2 * 3$

$= T1(n-4) + 2 * 4$

Now we can see the pattern as follows:

$T1(n) = T1(n-k) + 2 * k$, where k is an integer between 1 and n // Equation 15.5

Plugging in $k = n$ in Equation 15.5, we get

$T1(n) = T1(0) + 2 * n = 1 + 2 * n = 2 * n + 1$

Proof by Induction Method

If we can guess the value of $T1(n)$ as a function of n, then we can use a proof by induction in order to prove that our guess is correct. We can use the preceding iteration method to come up with a guess for $T1(n)$.

Generally, a proof by induction works as follows:

- Verify that our statement (equation in this case) is true for a base case.
- Assume that out statement is true up to n.
- Prove that it is true for $n + 1$.

Let's go through the induction steps with our guess that $T1(n) = 2 * n + 1$, which we may have generated from our iterative or handwaving method.

Step 1: Verify that the value that our guess gives to $T1(0)$ is correct.

$$\begin{aligned} T1(0) &= 2 * 0 + 1 \\ &= 1 \end{aligned}$$

Thus, our guess is correct for $T1(0)$.

Step 2: Assume that $T1(n) = 2 * n + 1$.

Step 3: Prove that $T1(n + 1) = 2 * (n + 1) + 1$.

Plugging in $x = n + 1$ in Equation 15.2, we get

$$T1(n + 1) = T1(n) + 2$$

Then, using our assumption and replacing $T1(n)$ by $2 * n + 1$, we get

$$\begin{aligned} T1(n + 1) &= 2 * n + 1 + 2 \\ &= 2 * n + 2 + 1 \\ &= 2 * (n + 1) + 1 \end{aligned}$$

Thus, we just proved, by induction, that our guess $T1(n) = 2 * n + 1$ is correct.

Other Methods

Another method is to use the Master Theorem, but that is beyond the scope of this book.

So the running time of *powerOf2A*(n) is $2 * n + 1$, or $O(n)$.

Let's now compute the running time of *powerOf2B* as a function of the input n. We will call it $T2(n)$.

In the base case (n is equal to 0), *powerOf2B* takes only one comparison to return 1. Thus,

$$T2(0) = 1$$

Generally, since it takes $T2(n)$ to compute and return *powerOf2B*(n), then it takes $T2(n-1)$ to compute and return *powerOf2B*($n-1$). Thus, in the general case, the comparison in the *if* statement will cost us one instruction; computing and returning *powerOf2B*($n-1$) will cost us $T2(n-1)$; doing it a second time will cost us another $T2(n-1)$; and adding the two and returning the sum as the result will cost us one instruction. Thus, the total time $T2(n)$ can be expressed as follows:

$$\begin{aligned} T2(n) &= 1 + T2(n-1) + T2(n-1) + 1 \\ &= 2 * T2(n-1) + 2 \qquad \text{// Equation 15.6} \end{aligned}$$

From there, we will use the iteration method in order to compute the value of $T2(n)$ as a function of n.

Substituting x for n, we can rewrite Equation 15.6 as follows:

$$T2(x) = 2 * T2(x-1) + 2 \quad \text{// Equation 15.7}$$

Using $x = n - 1$ in Equation 15.7, we get

$$T2(n-1) = 2 * T2(n-2) + 2$$

Plugging in the value of $T2(n-1)$ into Equation 15.6, we get

$$\begin{aligned} T2(n) &= 2 * (2 * T2(n-2) + 2) + 2 \\ &= 2^2 * T2(n-2) + 2^2 + 2 \qquad \text{// Equation 15.8} \end{aligned}$$

Again, we leave $2^2 + 2$ as an expression to try to let a pattern develop.

Using $x = n - 2$ in Equation 15.7, we get

$$T2(n-2) = 2 * T2(n-3) + 2$$

Plugging in the value of $T2(n-2)$ into Equation 15.8, we get

$$\begin{aligned} T2(n) &= 2^2 * (2 * T2(n-3) + 2) + 2^2 + 2 \\ &= 2^3 * T2(n-3) + 2^3 + 2^2 + 2 \qquad \text{// Equation 15.9} \end{aligned}$$

Using $x = n - 3$ in Equation 15.7, we get

$$T2(n-3) = 2 * T2(n-4) + 2$$

Plugging in the value of $T2(n-3)$ into Equation 15.9, we get

$$\begin{aligned} T2(n) &= 2^3 * (2 * T2(n-4) + 2) + 2^3 + 2^2 + 2 \\ &= 2^4 * T2(n-4) + 2^4 + 2^3 + 2^2 + 2 \qquad \text{// Equation 15.10} \end{aligned}$$

Now we can see the pattern as follows:

$T2(n) = 2^k * T2(n-k) + 2^k + 2^{k-1} + \ldots + 2^2 + 2$, where k is an integer between 1 and n // Equation 15.11

Noting that

$$\begin{aligned} 2^k + 2^{k-1} + \ldots + 2^2 + 2 &= -1 + 2^k + 2^{k-1} + \ldots + 2^2 + 2 + 1 \\ &= -1 + (2^{k+1} - 1) / (2 - 1) \\ &= 2^{k+1} - 2 \end{aligned}$$

Equation 15.11 becomes

$T2(n) = 2^k * T2(n-k) + 2^{k+1} - 2$, where k is an integer between 1 and n // Equation 15.12

Plugging in $k = n$ in Equation 15.12 in order to reach the base case of $T2(0)$, we get

$$\begin{aligned} T2(n) &= 2^n * T2(0) + 2^{n+1} - 2 \\ &= 2^n * 1 + 2^{n+1} - 2 \\ &= 2^n + 2^{n+1} - 2 \\ &= 2^n (1 + 2) - 2 \\ &= 3 * 2^n - 2 \\ &= O(2^n) \end{aligned}$$

Thus, *powerOf2A* runs in $O(n)$ while *powerOf2B* runs in $O(2^n)$, although they perform the same function.

As a result, computing 2^{20} using *powerOf2A* will cost 20 statement executions while computing 2^{20} using *powerOf2B* will cost 1 million statement executions.

This simple example shows that how we code a method can have a significant impact on its running time.

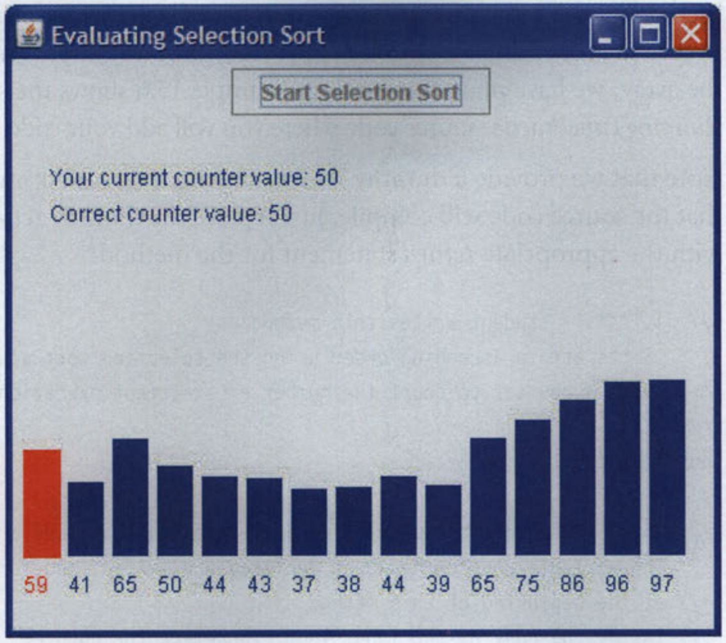
15.4 Programming Activity: Tracking How Many Statements Are Executed by a Method

REFERENCE POINT

Selection Sort is explained in Chapter 8.

In this activity, you will work with a variable-size integer array. Specifically, you will perform the following operations:

1. Write code to keep track of the number of statement executions during a selection sort.
2. Run a simulation to compute the number of statements executed as a function of the number of elements in the array.



Your current counter value: 50
Correct counter value: 50

Figure 15.1
Animation of the Programming Activity

3. Estimate the running time of Selection Sort as a function of *n*, the number of elements in the array being sorted.

The framework for this Programming Activity will animate your algorithm so that you can perform a simulation on the number of statement executions inside the *selectionSort* method compared to the number of elements in the array that is sorted. For example, Figure 15.1 shows the current number of statement executions for an array of 15 elements.

At this point, the application has executed 50 statements.

Instructions

In the Chapter 15 Programming Activity directory on the CD-ROM accompanying this book, you will find the source files needed to complete this activity. Copy all the files to a directory on your computer. Note that all files should be in the same directory.

Open the *RunningTimePractice.java* source file. Searching for five asterisks (*****) in the source code will position you at the sample method where you will add your code. In this task, you will fill in the code for the *selectionSort* method in order to keep track of the number of statement executions needed to sort an array using the Selection Sort algorithm. You should not instantiate the array; we have done that for you. Example 15.6 shows the section of the *RunningTimePractice* source code where you will add your code.

Note that we provide a dummy *return* statement (*return* 0;). We do this so that the source code will compile. Just replace the dummy *return* statement with the appropriate *return* statement for the method.

```
// 1. *****  student writes this method
/**  Sorts arr in ascending order using the selection sort algorithm
*    Adds a counter to count the number of statement executions
*/
public int selectionSort( )
{
 // Note:  To count the number of statement executions, use a counter
 // The variable counter has been declared and initialized for you
 // at the beginning of this method
 // Inside the body of the inner loop, increment the counter
 // Replace the return statement so that this method returns the value of
 // the counter. To slow down or accelerate the animation, modify the
 // argument of Pause.wait in the animate method
 int counter = 0;
 int temp, indexOfMax;
 for ( int i = 0; i < size; i++ )
 {
   // find index of largest value in the subarray
   indexOfMax = 0;
   animate( i, 0, counter );
   for ( int j = 1; j < arr.length - i; j++ )
   {

    if ( arr[j] > arr[indexOfMax] )
     indexOfMax = j;
    animate( i, j, counter );
   }
   // swap arr[indexOfMax] and arr[arr.length - i]
   temp = arr[indexOfMax];
```

```
      arr[indexOfMax] = arr[arr.length - i - 1];
      arr[arr.length - i - 1] = temp;
    }
    return 0;
  } // end of selectionSort
```

EXAMPLE 15.6 Location of Student Code in *RunningTimePractice*

Our framework will animate your algorithm so that you can watch your code work. If you want to accelerate or slow down the animation, modify the argument of *Pause.wait* in the *animate* method.

To test your code, compile and run the *RunningTimePractice* source code. When the program begins, you will be prompted for the number of elements in the array. Because the values of the array are randomly generated, the values will be different each time the program runs.

Troubleshooting

If the animation is incorrect, and you think your method does return a correct value for the counter, verify that you correctly incremented the counter inside the inner loop.

In order to derive a closed-end expression for the number of statement executions as a function of the size of the array, follow these tips:

- If n is the size of the array, compare $n, n^2, n^3, n^4, \ldots, 2^n$, to the value of the counter.
- When doing the preceding, divide $n, n^2, n^3, n^4, \ldots, 2^n$ by the number of statements executed.

? DISCUSSION QUESTIONS

1. What is the value of the counter with the following array sizes: 5, 10, 15, 20, 25?
2. In relation to *n*, the size of the array, what is the value of the counter?
3. What is the running time of Selection Sort in Big-Oh notation?
4. If the array is already sorted in either the correct or opposite order, does that make a difference in the number of statement executions? What can you say about the worst-case and best-case running times?

15.5 Running Time Analysis of Searching and Sorting Algorithms

In studying the running time of various searching and sorting algorithms, we will look at the following scenarios:

- best case
- worst case
- average case

Some methods have a very efficient running time. We mentioned earlier that the running time of Binary Search was log *n*. Thus, searching a sorted array of 1 billion items using Binary Search will only take 30 statement executions since log (1 billion) is approximately 30.

Example 15.7 shows the code of the recursive binary search method introduced in Chapter 13.

```
public static int recursiveBinarySearch
                ( int [ ] arr, int key, int start, int end )
{
  if ( start <= end )
  {
    // look at the middle element of the subarray
    int middle = ( start + end ) / 2;

    if ( arr[middle] == key )   // found key, base case
      return middle;
    else if ( arr[middle] > key ) // look lower
      return recursiveBinarySearch( arr, key, start, middle - 1 );
    else                          // look higher
      return recursiveBinarySearch( arr, key, middle + 1, end );
  }
  else                            // key not found, base case
    return -1;
}
```

EXAMPLE 15.7 Recursive Binary Search

In the best-case scenario, we will find the search value exactly in the middle of the array, at the array index we check first. Thus, the best-case running time of Binary Search is $O(1)$. In the worst-case scenario, we will not find the search value in the array. Let's compute the running time of the worst-case scenario.

In the general case, the comparison of the first *if* statement will cost us one instruction; the assignment statement will cost us two instructions; the comparison in the second *if* statement will cost us one instruction; the comparison in the *else/if* statement will also cost us one instruction; computing and returning *recursiveBinarySearch*(*arr, key, start, middle* – 1) or *recursiveBinarySearch*(*arr, key, middle* + 1, *end*) will cost us $T(n/2 - 1)$ or $T(n/2)$ instructions. Note that only one recursive call will be made. Thus, the total time $T(n)$ can be expressed as follows:

$$\begin{aligned} T(n) &= 1 + 2 + 1 + 1 + T(n / 2) \\ &= T(n / 2) + 5 \quad \text{// Equation 15.13} \end{aligned}$$

In the base case (n is equal to 1), *recursiveBinarySearch* makes only the first comparison, one addition, one division, the second comparison, and then returns the index of the found element or –1. Thus,

$$T(1) = 5.$$

From there, we will use the iteration method in order to compute the value of $T2(n)$ as a function of n.

Substituting x for n, we can rewrite Equation 15.13 as follows:

$$T(x) = T(x / 2) + 5 \qquad \text{// Equation 15.14}$$

Using $x = n / 2$ in Equation 15.14, we get

$$\begin{aligned} T(n / 2) &= T((n / 2) / 2) + 5 \\ &= T(n / 2^2) + 5 \end{aligned}$$

Plugging in the value of $T(n / 2)$ into Equation 15.13, we get

$$\begin{aligned} T(n) &= (T(n / 2^2) + 5) + 5 \\ &= T(n / 2^2) + 5 * 2 \quad \text{// Equation 15.15} \end{aligned}$$

Using $x = n / 2^2$ in Equation 15.14, we get

$$\begin{aligned} T(n / 2^2) &= T((n / 2^2) / 2) + 5 \\ &= T(n / 2^3) + 5 \end{aligned}$$

Plugging in the value of $T(n / 2^2)$ into Equation 15.15, we get

$$\begin{aligned} T(n) &= (T(n / 2^3) + 5) + 5 * 2 \\ &= T(n / 2^3) + 5 * 3 \quad \text{// Equation 15.16} \end{aligned}$$

Using $x = n / 2^3$ in Equation 15.14, we get

$$\begin{aligned} T(n / 2^3) &= T((n / 2^3) / 2) + 5 \\ &= T(n / 2^4) + 5 \end{aligned}$$

Plugging in the value of $T(n / 2^3)$ into Equation 15.16, we get

$$T(n) = (T(n / 2^4) + 5) + 5 * 3$$
$$= T(n / 2^4) + 5 * 4 \quad // \text{Equation 15.17}$$

Now we can see the pattern as follows:

$$T(n) = T(n / 2^k) + 5 * k,$$
$$\text{where } k \text{ is an integer between 1 and } n \quad // \text{Equation 15.18}$$

We now want to choose k such that $n / 2^k$ is equal to 1 in order to reach our base case. If $n / 2^k = 1$, then $n = 2^k$, and taking the log of each side:

$$\begin{aligned}\log n &= \log 2^k \\ &= k \log 2 \\ &= k * 1 \\ &= k\end{aligned}$$

Plugging in $k = \log n$ in Equation 15.18, we get

$$\begin{aligned}T(n) &= T(1) + 5 * \log n \\ &= 2 + 5 * \log n \\ &= O(\log n)\end{aligned}$$

Thus, Binary Search is $O(\log n)$ in the worst case. Note that the value of the original constant, here 5, does not impact the order of magnitude of the running time.

REFERENCE POINT

Insertion Sort is explained in Chapter 8.

In the average case, we will find the search value after performing half the number of comparisons as in the worst-case scenario. Thus, the average running time of binary search is also $O(\log n)$.

Now, let's calculate the running time of Insertion Sort as a function of n, the number of elements in the array. From Chapter 8, the code of the Insertion Sort method is shown in Example 15.8.

```
/**  Performs an Insertion Sort on an integer array
*    @param array   array to sort
*/
public static void insertionSort( int [ ] array )
{
  int j, temp;

  for ( int i = 0; i < array.length; i++ )
  {
        j = i;
        temp = array[i];

        while ( j != 0 && array[j - 1] > temp )
        {
           array[j] = array[j - 1];
           j--;
        }

        array[j] = temp;
     }
   }
```

EXAMPLE 15.8 Insertion Sort

The *for* loop header will execute $n + 1$ times. We will execute the body of the *for* loop n times.

In the best case, the array is already sorted. In this case, the *while* loop condition will always evaluate to *false*, and we will never execute the *while* loop body. So inside the *for* loop, the three statements and the loop condition will each execute once for each iteration of the *for* loop, thus executing a total of $4 * n$ times. Therefore, the best-case running time is $O(n)$.

In the worst case, the array is sorted in the opposite order. In this case, the *while* loop condition will always be *true* for its first evaluation, and we will enter the *while* loop every time we iterate the *for* loop. Thus, the two statements inside the *while* loop will each execute $(1 + 2 + 3 + 4 + \ldots + (n-1))$ times. Since $(1 + 2 + 3 + 4 + \ldots + (n-1)) = n * (n-1) / 2$, the worst-case running time of insertion sort is $O(n^2)$.

In the average case, we will enter the *while* loop half the times we try. The average case is still $O(n^2)$.

Bubble Sort, presented in Programming Activity 2 of Chapter 8, like Insertion Sort, is implemented with a double loop and also is $O(n^2)$.

Merge Sort and Quick Sort are two sorting algorithms implemented recursively.

The pseudocode for Merge Sort, which is the subject of the Group Project of Chapter 13, is as follows:

- If the array has only one element, it is already sorted, thus do nothing; otherwise:
    - Merge sort the left half of the array.
    - Merge sort the right half of the array.
    - Merge the two sorted half-arrays into one in a sorted manner.

The last operation involves looping through all the elements of the two half-arrays; it takes $O(n)$; thus, we can derive the following recursive formulation for its running time of Merge Sort:

$$\begin{aligned} T(n) &= T(n / 2) + T(n / 2) + n \\ &= 2\ T(n / 2) + n \end{aligned}$$

Using derivation, we get

$$\begin{aligned} T(n) &= 2\ T(n / 2) + n \\ &= 2\ (2\ T(n / 2^2) + n / 2) + n \\ &= 2^2\ T(n / 2^2) + 2n \end{aligned}$$

Continuing to iterate,

$$\begin{aligned} T(n) &= 2^2\ T(n / 2^2) + 2n \\ &= 2^2\ (2\ T(n / 2^3) + n / 2^2) + 2n \\ &= 2^3\ T(n / 2^3) + 3n \\ T(n) &= 2^3\ T(n / 2^3) + 3n \\ &= 2^3\ (2\ T(n / 2^3) + n / 2^3) + 3n \\ &= 2^4\ T(n / 2^4) + 4n \end{aligned}$$

Thus, we identify the general pattern

$$T(n) = 2^k\ T(n / 2^k) + kn$$

Choosing k so that $n / 2^k = 1$ in order to reach the base case, i.e., $n = 2^k$, $k = \log n$, we get

$$\begin{aligned} T(n) &= n\ T(1) + n \log n \\ &= O(n \log n) \end{aligned}$$

So Merge Sort is $O(n \log n)$, better than Insertion Sort, Bubble Sort, and Selection Sort. It is the same for best-case, worst-case, and average-case scenarios.

The analysis of the running time of Quick Sort is the subject of the Group Project for this chapter.

CODE IN ACTION

On the CD-ROM included with this book, you will find a Flash movie with a step-by-step illustration of how to compute running times for various methods. Click on the link for Chapter 15 to start the movie.

Skill Practice
with these end-of-chapter questions

15.7.1	Multiple Choice Exercises	
		Questions 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
15.7.2	Compute the Running Time of a Method	
		Questions 11, 12, 13, 14, 15, 16, 17, 18
15.7.4	Technical Writing	
		Question 27

15.6 Chapter Summary

SUMMARY

- The running time of an algorithm is expressed as a function of its inputs or its number of inputs.
- Orders of magnitude are, in increasing order of execution time: constant, log, polynomial, and exponential. Exponential running times are undesirable.
- Big-Oh notation is the industry standard notation for running times.
- Considering a mathematical function that represents a running time of an algorithm, that function is Big-Oh of its most dominant term.
- The coding of a method directly impacts its running time.

15.7 Exercises, Problems, and Projects

15.7.1 Multiple Choice Exercises

1. What is the Big-Oh of this function:

   ```
   T(n) = n² - 2 n + 99
   ```

   ❑ $O(n^2)$

   ❑ $O(99)$

   ❑ $O(n)$

   ❑ $O(1)$

2. What is the Big-Oh of this function:

   ```
   T(n) = n³ + 10 n² + 20 n + 30
   ```

   ❑ $O(n^3)$

   ❑ $O(n^2)$

   ❑ $O(n)$

   ❑ $O(1)$

3. What is the Big-Oh of this function:

   ```
   T(n) = n² + n * log n + 12 n + 5
   ```

   ❑ $O(n * \log n)$

   ❑ $O(n^2)$

   ❑ $OO(n)$

   ❑ $O(1)$

4. We have the following recurrence relation representing the running time of a function; what is the running time of that function?

   ```
   T(n) = T(n -  1) + 1
   ```

   ❑ $O(2^n)$

   ❑ $O(n * \log n)$

   ❑ $O(n^2)$

   ❑ $O(n)$

5. Which of these running times is the worst?

   ❑ $O(n^5)$

   ❑ $O(2^n)$

   ❑ $O(n * \log n)$

   ❑ $O(n)$

6. Look at the following method:

```
public static int foo1( int n )
{
  if ( n > 1 )
      return ( 2 * foo1( n / 4 ) );
  else
      return 1;
}
```

   What recurrence formulation best illustrates the running time of the preceding method?

   ❑ $T(n) = T(n * 4) + 3$

   ❑ $T(n) = T(n / 4) + 3$

   ❑ $T(n) = T(n - 4) + 3$

   ❑ $T(n) = T(n + 4) + 3$

7. What is $\Sigma\ i$ for $i = 1$ to n equal to?

   ❑ n^2

   ❑ $n * (n + 1) / 2$

   ❑ $2n$

   ❑ n

8. What is $\Sigma\ 1$ for $i = 1$ to n equal to?

   ❑ n^2

   ❑ n

   ❑ $n * (n + 1) / 2$

   ❑ i

9. What is the running time of the *foo2* method?

```
public static void foo2( int n )
{
  for ( int i = n; i > 0; i– )
  {
   for ( int j = 0; j < n; j++ )
     System.out.println( "Hello" );
  }
}
```

❑ $O(n^4)$

❑ $O(n^3)$

❑ $O(n^2)$

❑ $O(n)$

10. What is the running time of the *foo3* method?

```
public static void foo3( int n )
{
  for ( int i = 0; i < n; i++ )
  {
   for ( int j = 0; j < i; j++ )
     System.out.println( "Hello" );
  }
}
```

❑ $O(n^4)$

❑ $O(n^3)$

❑ $O(n^2)$

❑ $O(n)$

15.7.2 Compute the Running Time of a Method

11. What is the running time of the *foo4* method (assume that the parameter *arr* is a two-dimensional array of *n* rows and *n* columns)?

```
public static void foo4( int [ ][ ] arr )
{
  for ( int i = 0; i < arr.length; i++ )
  {
   for ( int j = arr[i].length - 1; j >= 0; j++ )
     System.out.println( "Hello world" );
  }
}
```

12. What is the running time of the *foo5* method (assume that the parameter *arr* is a three-dimensional array where each dimension has exactly *n* elements)?

```
public static void foo5( int [ ][ ][ ] arr )
{
  for ( int i = 0; i < arr.length; i++ )
  {
   for ( int j = 0; j < arr[i].length; j++ )
   {
    for ( int k = 0; k < arr[i][j].length; k++ )
     System.out.println( "Hello world" );
   }
  }
}
```

13. What is the running time of the *foo6* method?

```
public static void foo6( int n )
{
  if ( n <= 0 )
     System.out.println( "Hello world" );
  else
     foo6( n - 1 );
}
```

14. What is the running time of the *foo7* method?

```
public static int foo7( int n )
{
  // n is guaranteed to be >= 0
  if ( n == 0 )
      return 0;
  else
      return ( n + foo7( n - 1 ) );
}
```

15. What is the running time of the *foo8* method?

```
public static int foo8( int n )
{
  // n is guaranteed to be >= 1
  if ( n == 1 || n == 2 )
     return 1;
  else
     return ( foo8( n - 1 ) + foo8( n - 2 ) );
}
```

Hint: Note that $T(n-2) <= T(n-1)$.

16. What is the running time of the *foo9* method?

```
public static void foo9( int n )
{
  // n is guaranteed to be >= 0
  if ( n == 0 )
     System.out.println( "done" );
  else
     foo9( n / 2 );
}
```

17. What is the running time of the *foo10* method as a function of *n* and *p*?

```
public static void foo10( int n, int p )
{
  // n and p are guaranteed to be >= 1
  if ( p >= n )
     System.out.println( "done" );
  else
     foo10( n, 2 * p );
}
```

18. What is the running time of the *foo11* method?

```
public static void foo11( int n )
{
  // n is guaranteed to be >= 0
  if ( n == 0 )
      return 0;
  else
      return ( 5 + 2 * foo11( n - 1 ) );
}
```

15.7.3 Programming Projects

19. Write a program that includes a method taking a single-dimensional array of *ints* as its only parameter, and returning the average of all the elements of the array. Add the necessary code to count how many statements are executed in the innermost loop. Run several simulations depending on the number of elements in the parameter integer array. What is the running time of that method as a function of the number of elements of the parameter array?

20. Write a program that includes a method converting a two-dimensional array of *ints* to a two-dimensional array of *boolean* values. If the integer value is greater than or equal to 0, then the corresponding *boolean* value is *true;* otherwise it is *false.* Add the necessary code to count how many statements are executed in the innermost loop. Run several simulations depending on the number of rows and columns in the argument integer array. What is the running time of that method as a function of the number of rows and columns of the parameter array? (You should assume that each row has the same number of columns.)

21. Write a program that includes a method computing the largest element of a given column (represented by a parameter of the method) of a two-dimensional array of *ints.* Add the necessary code to count how many statements are executed in the innermost loop. Run several simulations depending on the number of rows and columns in the parameter integer array, as well as the index of the column for which the method calculates the largest element. Does the running time of the method depend on the column index? the number of rows? the number of columns? What is the running time of that method as a function of the number of rows and columns of the parameter array and the column index? (You should assume that each row has the same number of columns.)

22. Write a program that includes a method taking a two-dimensional array of *ints* as its only parameter, and returning a single dimensional array of *ints* such that each element of the returned array is the sum of the corresponding row in the parameter array. Add the necessary code to count how many statements are executed in the innermost loop. Run several simulations depending on the number of rows and columns in the parameter integer array. What is the running time of that method as a function of the number of rows and columns of the parameter array? (You should assume that each row has the same number of columns.)

23. Write a program that implements a recursive Binary Search, and add the necessary code to count how many times *binarySearchRecursive* is being called. Run several simulations on arrays of 32, 64, and 128 elements. How many times is the method called in the best-case scenario and worst-case scenario? Does that match our analysis in the chapter?

24. Write a program that implements the recursive method to compute the factorial of a number from Chapter 13 and add the necessary code to count how many times the method is being called. Run several simulations depending on the value of *n*. How many times is the method called? What is the running time of this method?

25. Write a program that includes a method converting a *String* of 0s and 1s to its equivalent decimal number and add the necessary code to count how many times the method is being called. Run several simulations depending on the length of the input *String*. How many times is the method called? What is the running time of that method?

26. Write a program that includes a method converting a decimal number to its equivalent binary number represented by a *String* of 0s and 1s and add the necessary code to count how many times the method is being called. Run several simulations depending on the decimal number. How many times is the method called? What is the running time of that method?

15.7.4 Technical Writing

27. Explain why it is important to consider running time when coding algorithms. Use an example to illustrate your point. Your example, web-based or not, should deal with a lot of data.

15.7.5 Group Project (for a group of 1, 2, or 3 students)

28. Write a class with an *int* array as its only instance variable. Write a recursive method that uses the Quick Sort algorithm in order to sort the array. (Quick Sort is explained below.) You will then add the appropriate code and perform the appropriate simulations to evaluate the running time of the method as a function of the number of elements in the array.

    Here is how Quick Sort works:

    - ❑ Partition the array so that all the elements to the left of a certain index are smaller than the element at that index and all the elements to the right of that index are greater than or equal to the element at that index. You should code a separate method to partition the array. (See explanation that follows.)

- ❑ Sort the left part of the array using Quick Sort (this is a recursive call).
- ❑ Sort the right part of the array using Quick Sort (this is another recursive call).

To partition the array elements in the manner previously explained, you should code another method (this one nonrecursive) as explained as follows:

- ❑ Choose an element of the array (for example, the first element). We call this element **pivot**.
- ❑ This method partitions the array elements so that all the elements left of pivot are less than pivot, and all the element right of pivot are greater than or equal to pivot.
- ❑ This method returns an *int* representing the array index of pivot (after the elements have been partitioned in the order described previously).
- ❑ In order to rearrange the array elements as previously described, implement the following pseudocode.

The following is pseudocode to partition a subarray whose lower index is *low* and higher index is *high*:

```
Assign element at index low to pivot
Initialize j to low
Loop from (low + 1) to high with variable i
   If (array element at index i is smaller than pivot)
        Increase j by 1
        Swap array elements at indexes i and j
Swap array elements at index low and j
Return j
```

Using a counter, keep track of the number of statement executions performed when using Quick Sort to sort an array of *n* elements. In particular, you should run simulation runs on these two situations:

- ❑ The array is not sorted
- ❑ The array is presorted in the correct order

You should perform a mathematical analysis of the running time of Quick Sort in the average case based on its recursive formulation (using iteration, as we did in the chapter examples).