[PART] II

ALGORITHMS AND Data Structures

Part I focused on the first two stages of software development: problem specification and object-oriented design. We have defined the problem and specified its solution in terms of classes, interfaces, and methods.

We are ready to begin the next phase: choosing the algorithms and data structures to implement these methods. As we learn in Chapter 5, this choice will have a profound impact on the efficiency of our solution. Chapters 6, 7, and 8 introduce a range of interesting and important data structures. This presentation is based on a classification scheme that will allow you to better understand this large and complex topic.

Finally, Chapter 9 shows that in a modern software development environment, most data structures already exist as part of a software library. In the case of Java, this library is the Java Collection Framework. Today, the most important concern about data structures is not learning how to design and implement them, but learning how to analyze and select the right one to solve a problem.

[CHAPTER] 5

THE ANALYSIS OF Algorithms

5.1 Introduction

We have reached the point in the software development process where we have specified the problem and designed a solution in terms of classes, objects, and methods. We are now ready to implement these methods by choosing our *data structures* and coding the *algorithms* that manipulate the information stored in these structures. These are important decisions, as our choices profoundly affect the running time of the finished code. Selecting an inappropriate structure can reduce program efficiency by orders of magnitude.

The enormous increases in processor speed in the last few years may have lulled you into believing that efficiency is no longer a concern during software development. That view is totally incorrect. While processor speeds have increased dramatically, so has the size of the computational problems being addressed. Today it is common to run enormous simulation models, search massive terabyte databases, and process extremely high-resolution images. Efficiency in the 21st century is just as important as when computing was in its infancy 50 to 60 years ago.

THE MOTHER OF ALL COMPUTATIONS

Climatic changes occur slowly, often taking hundreds or thousands of years to complete. For example, the ice ages were periods when large areas of the Earth were covered by glaciers. Individual ice ages were separated by thousands of years during which the Earth became warmer and the glaciers receded. To study global climate change, researchers cannot look at data for only a few dozen years. Instead, they must examine changes over long periods of time.

To provide this type of data, scientists at the National Center for Atmospheric Research (NCAR) used a supercomputer at the U.S. Department of Energy's National Energy Research Scientific Computing Center (NERSC) to carry out a 1000-year simulation of climatic changes on Earth. NCAR used a 6000-processor IBM-SP supercomputer to run its Community Climate System Model (CCSM2). This massive machine worked on the problem continuously, seven days a week, modeling decade after decade and century after century of climate changes. Finally, after more than 200 days of uninterrupted computing and about a hundred billion billion (10^{20}) computations, the multimillion-dollar machine announced that it had completed its task.

continued

Data from this simulation is being shared with the environmental research community to further the study of changes to our climate and investigate such phenomena as global warming, polar ice cap melting, and El Nino ocean currents. Plans are already underway at NCAR to carry out longer and even more complex simulations that demand up to a thousand times more computational power.

This chapter introduces the mathematical tools needed to analyze the performance of the algorithms and data structures discussed in the upcoming chapters. The data structures we will present—lists, stacks, queues, and trees—are not simply interesting, but allow us to create more efficient algorithms for such basic tasks as insertion, deletion, retrieval, searching, and sorting. How can we demonstrate, however, that our claims of efficiency are valid? How do we show that algorithm X is truly superior to algorithm Y without relying on informal arguments or being unduly influenced by either the programming language used to code the algorithm or the hardware that runs it? Specifically, how can we demonstrate that one algorithm is superior to another without being misled by any of the following conditions?

- Special cases—Every algorithm has certain inputs that allow it to perform far better than would otherwise be expected. For example, a sequential search algorithm works surprisingly well if the item you are looking for appears at the front of the list—for example, searching for AAA Auto Rental in the telephone book. However, this case is obviously special and not indicative of how the algorithm performs for an arbitrary piece of data.
- Small data sets—An algorithm may not display its inefficiencies if a data set is too small. For example, walking is a much slower way to travel than driving. However, that may not be true if the distance traveled is only a few feet. In that case, the overhead of starting a car overwhelms any benefits of its increased speed. In general, a problem must be sufficiently large to demonstrate the benefits of one algorithm over another.
- Hardware differences—If an inefficient algorithm is running on a multimillion-dollar supercomputer and an efficient algorithm is running on a PDA, the inefficient algorithm may appear to be superior because of speed differences between the two machines.
- Software differences—If an inefficient algorithm was coded by a team of professionals in optimized assembly language and an efficient algorithm was written by a first-year computer science student in interpreted BASIC, the inefficient algorithm may appear to be superior because of differences in the languages and the skill of the programmers.

Are any of these concerns valid? Could we use any of them to argue the superiority of one algorithm over another? If they are not valid, why not, and how can we formally refute them? This chapter presents the mathematical tools needed to answer these questions. It introduces a technique called **asymptotic analysis**, which is the fundamental tool for studying the run-time performance of algorithms. The results of this analysis are expressed using a technique called big-O notation. We will use these tools to analyze the algorithms and data structures presented in succeeding chapters.

5.2 The Efficiency of Algorithms

The **efficiency** of an algorithm measures the amount of resources consumed in solving a problem of size n. In general, the resource that interests us most is *time*—how fast an algorithm can solve a problem of size n. We can use the same techniques to analyze the consumption of other resources, such as memory space.

It would seem that the most obvious way to measure the efficiency of an algorithm is to run it with some specific input and measure how much processor time is needed to produce the correct solution. This type of "wall clock" timing is called **benchmarking**. However, this produces a measure of efficiency for only one particular case, and is inadequate for predicting how the algorithm would perform on a different data set. As mentioned in the previous section, our data set may be too small, or it may have special characteristics not present in other data sets. For example, an algorithm that finds a name in a telephone book by searching sequentially from *A* to *Z* works well if we test it on a book containing 100 entries, but it would be unacceptable for use with the New York City directory. Benchmarking is a technique for examining whether a finished program meets the timing constraints in the problem specification document described in Section 1.2.1 of this book. It is not an appropriate way to mathematically analyze the general properties of algorithms *before* we begin coding.

Instead, we need a way to formulate general guidelines that allow us to state that, for any arbitrary inputs, one method is likely better than another. In the phone book example, a more helpful statement would be: "Never use sequential lookup with a telephone book containing more than a few hundred entries, as it is probably too slow."

The time it takes to solve a problem is usually an increasing function of its size—the bigger the problem, the longer it takes to solve. We need a formula that associates n, the problem size, with t, the processing time required to obtain a solution. For example, if we are searching or sorting a list, n would be the number of items in the list. If we are performing matrix operations on an $r \times r$ array, the problem size would be r, the dimensions of the array. If we are placing an object into a queue, the problem size might be the number of items in the queue.

The relationship between n and t can sometimes be expressed in terms of an explicit formula $t = f(n)$. Using such a formula, we could plug in a value for n and determine exactly how many seconds it would take to solve a problem of that size. Given that information for a number of different techniques that solve the same problem, we could select the algorithm that runs fastest for problems of size n. However, such explicit formulas are rarely used. They are difficult to obtain because they rely on highly technical machine-dependent parameters that we may not know, such as instruction cycle time, memory access time, or the compiler's internal characteristics. Furthermore, we usually would not want to use $t = f(n)$ to compute exact timings for specific cases. Instead, as mentioned earlier, we want a *general* method that allows us to study the performance of an algorithm on data sets of arbitrary size.

We can get such information using a technique called asymptotic analysis. This analysis allows us to develop expressions of the following form:

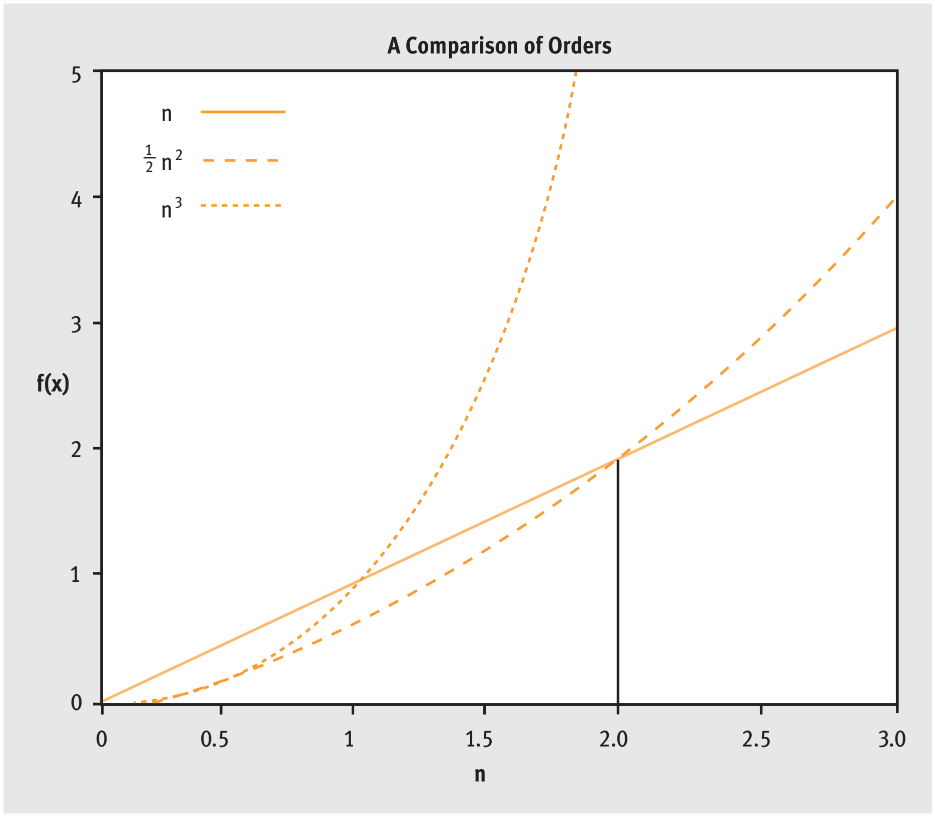
$$t \approx O(f(n))$$

which is read "t is on the order of $f(n)$." This representation is called **big-O notation**.

Formally, the expression states that there are positive constants M and N_0 such that if $t \approx O(f(n))$, then $0 \leq t \leq Mf(n)$ for all $n > N_0$. This formidable-looking definition is not as difficult as it may appear. It simply states that an algorithm's computing time grows no faster than (i.e., is bounded by) a constant times a function of the form $f(n)$. If the order of the algorithm were $O(n^3)$, for example, then the relationships between t and n would be given by a formula such as $t = kn^3$ + "lower-order terms," although we generally do not know anything about the value of the constant k or the lower-order terms. However, by knowing this relationship, we can say that if the size of the problem doubles, the total time needed to solve it increases about eightfold (2^3). If the problem size triples, the overall running time is about 27 times greater (3^3). This information can be enormously helpful.

For reasonably large problems, we always want to select an algorithm of the lowest possible order. If algorithm A is $O(f(n))$ and algorithm B is $O(g(n))$, then algorithm A is said to be of a **lower order** than B if $f(n) < g(n)$ for all n greater than some constant k. For example, $O(n^2)$ is a lower-order running time than $O(n^3)$ because $n^2 < n^3$ for all $n > 1$. Similarly, $O(n^3)$ is a lower-order running time than $O(2^n)$ because $n^3 < 2^n$ for all $n > 9$. Intuitively, this means that the expression n^2 grows more slowly than n^3, and the expression n^3 grows more slowly than the expression 2^n. If possible, then, we want to select an $O(n^2)$ algorithm to solve a problem rather than an $O(n^3)$ or $O(2^n)$ algorithm. When one algorithm is of a lower order than another, it is **asymptotically superior**.

If we choose an asymptotically superior algorithm to solve a problem, we will not know exactly how much time is required, but we know that as the problem size increases there will always be a point beyond which the lower-order method takes less time than the higher-order algorithm. That is, once the problem size becomes sufficiently large, the asymptotically superior algorithm always executes more quickly. Figure 5-1 demonstrates this behavior for algorithms of order $O(n)$, $O(n^2)$, and $O(n^3)$. For small problems, the choice of algorithms is not critical; in fact, the $O(n^2)$ or $O(n^3)$ may even be superior. However, in this example, when n becomes larger than 2.0, the $O(n)$ algorithm in Figure 5-1 always has a superior running time than the other two algorithms, and it improves as n increases.



[FIGURE 5-1] Graphical comparison of complexity measures $O(n)$, $O(n^2)$, and $O(n^3)$

Programmers are generally not concerned with efficiency for small data sets, because any correct algorithm usually runs fast enough. However, as the problem size increases, efficiency becomes much more important. Big-O notation, which describes the asymptotic behavior of algorithms on large problems, provides exactly the type of information we need. It is therefore the fundamental technique for describing the efficiency properties of algorithms.

Figure 5-2 lists a number of common complexity classes ordered by increasing complexity function. In the next section, we will learn how to determine the complexity of these well-known algorithms.

COMPLEXITY	NAME	EXAMPLES
$O(1)$	Constant time	Accessing an element in an array
$O(\log n)$	Logarithmic time	Binary search
$O(n)$	Linear time	Sequential search
$O(n \log n)$	Optimal sorting time	Quicksort, merge sort
$O(n^2)$	Quadratic time	Selection sort, bubble sort
$O(n^3)$	Cubic time	Matrix multiplication
$O(2^n)$, $O(n!)$	Exponential time	The traveling salesperson problem

[FIGURE 5-2] Common computational complexities

5.3 Asymptotic Analysis

5.3.1 Average-Case vs. Worst-Case Analysis

When analyzing an algorithm, we can investigate three distinct behaviors—its **best-case**, **average-case**, and **worst-case** behavior.

We rarely do a best-case analysis, which investigates the optimal behavior of an algorithm, because it usually displays this optimal performance only under special or unusual conditions. Remember, we are trying to develop *general* guidelines that describe the overall behavior of an algorithm under all possible circumstances, not its performance in rare and exceptional circumstances.

Given this goal, it would seem that the most important condition to investigate is average-case behavior, which attempts to categorize the average performance of an algorithm over all possible inputs. Although this is an important measure, it is often mathematically difficult to do an average-case analysis because it is difficult to determine an average input. Furthermore, software is often designed and implemented to meet strict timing specifications. If we select algorithms on the basis of their average performance, we may not be able to assert that the finished program will *always* meet design specifications, but only that it will perform within specifications most of the time. Under certain "pathological" conditions, the algorithm may display worse performance and fail to meet user requirements. An average-case analysis does not warn us in this situation; only a worst-case analysis can provide this type of information.

For example, both the merge sort and Quicksort take an average time of $O(n \log n)$ to sort a random list of length n. (We explain how to determine this time in Section 5.5.) These two algorithms were implemented in Java, along with the $O(n^2)$ selection sort, and run with random arrays of length 10,000 and 100,000. The results are shown in Figure 5-3.

METHOD	LIST SIZE OF 10,000 (seconds)	LIST SIZE OF 100,000 (seconds)
Merge sort of randomized list	0.6	6.7
Quicksort of randomized list	0.5	5.4
Selection sort of randomized list	11.2	105.6
Merge sort of reverse ordered lists	0.8	7.2
Quicksort of reverse ordered lists	3.4	89.5
Selection sort of reverse ordered lists	11.5	109.9

[FIGURE 5-3] Running times for three different sorting algorithms

The merge sort and Quicksort are both $O(n \log n)$ in the average case. In the example shown in Figure 5-3, Quicksort executes a bit faster than merge sort because of slightly lower constant factors in its complexity function. Because of this behavior, we might opt for Quicksort when implementing any methods that require sorting. Also, notice that for larger data sets, both merge sort and Quicksort are significantly faster than the inefficient $O(n^2)$ selection sort, as we would expect.

However, what if the problem specification document stated that "the finished program must sort *any* list of length 100,000, regardless of its initial state, in 10 seconds or less?" It seems there is no problem, because Quicksort sorted the randomized list of 100,000 items in 5.4 seconds. However, if the list is either already sorted or in reverse order (i.e., backward from what we want), then the performance of Quicksort deteriorates badly. In this worst-case scenario, Quicksort executes more like an $O(n^2)$ algorithm than $O(n \log n)$.

We see this behavior in Figure 5-3, where we reran the problem with lists that were in reverse order. The times for the merge sort did not change markedly, as both its worst-case and average-case behavior were $O(n \log n)$. However, Quicksort "blew up," taking a minute and a half to solve the problem, almost as much time as the inefficient selection sort. Given a firm performance requirement that all inputs of length 100,000 be sorted in under 10 seconds, we could not use Quicksort in our software. However, if the performance requirements were changed to read "*most of the time* the program must sort a list of length 100,000 in under 10 seconds," then we would likely still choose Quicksort. The backward data set of Figure 5-3 is extremely rare and would not be expected to occur with any frequency. The usually superior performance of Quicksort on the data sets would be more important in this case.

You can modify Quicksort so its worst-case performance approaches O(n log n), independent of the input. This "optimized Quicksort" is the method of choice in most sorting libraries. See the Challenge Work Exercises at the end of this chapter.

The previous example clearly shows that a thorough, in-depth analysis of an algorithm requires study of both its average-case behavior under expected conditions and its worst-case behavior under the least favorable conditions.

5.3.2 The Critical Section of an Algorithm

When analyzing an algorithm, we do not care about the behavior of every statement. Such a detailed analysis is far too complicated and time consuming. Besides, many parts of an algorithm are not executed often, and analyzing their individual behavior is not important in determining overall efficiency. For example, initialization is frequently done only once, and its behavior will not significantly affect the overall running time.

Instead, we focus our analysis on one part of the algorithm, called a **critical section**, where the algorithm spends the greatest amount of its time. The critical section, which may be either a single operation, a single statement, or a small group of statements, has the following characteristics:

- It is central to the functioning of the algorithm, and its behavior typifies the overall behavior of the algorithm.
- It is inside the most deeply nested loops of the algorithm and is executed as often as any other section of the algorithm.

The critical section is at the heart of the algorithm. Asymptotic analysis dictates that we can characterize the overall efficiency of an algorithm by counting how many times this critical section is executed as a function of the problem size. (In an average-case analysis, it is the average number of times executed; in a worst-case analysis, it is the maximum number of times.) The critical section of the algorithm dominates the completion time, which allows us to disregard the contributions of the other sections of code. To describe this characteristic another way, if an algorithm is divided into two parts, the first taking $O(f(n))$, followed by a second that takes $O(g(n))$, then the overall complexity of the algorithm is $O(\max[f(n), g(n)])$. The slowest and most time-consuming section determines the overall efficiency of the algorithm.

As an analogy, imagine that you must deliver a gift to someone in another city thousands of miles away. The job involves the following three steps:

1. Wrapping the gift
2. Driving 3000 miles to the destination
3. Delivering the package to the proper person

The critical section of this algorithm is obviously Step 2. The time it takes to perform this step characterizes the completion time of the entire task, and Steps 1 and 3 can be ignored without significant loss of accuracy. Changing Step 2 to flying instead of driving would have a profound impact on the completion time. However, taking a class to learn how to wrap packages more quickly would have no discernable effect.

Similarly, consider the following fragment of code:

```
step 1;
for (int i = 0; i < 1000000; i++)
       step 2;
step 3;
```

Steps 1 and 3 are executed only once. Their contribution to efficiency is negligible and may be disregarded without losing accuracy. It is Step 2, the critical section that is done one million times, that characterizes the overall running time of this code fragment.

To summarize, asymptotic analysis is a counting process that determines, for either the average case or the worst case, how many times the critical section of the algorithm is executed as a function of n, the problem size. This relationship, expressed using big-O notation, characterizes the inherent efficiency of the algorithm.

5.3.3 Examples of Algorithmic Analysis

In this section we examine a few simple examples of the analysis of algorithms. Succeeding chapters contain many more examples.

Figure 5-4 shows a simple **sequential search** algorithm. It looks at every item in an n-item list to locate the first occurrence of a specific target.

```
/**
 * Searches the given array for the given integer.  If found, returns
 * its position.  If not, returns -1.
 *
 * @param array The array of integers to search
 * @param target The integer to search for
 * @return target's position if found, -1 otherwise
 */

public static int search( int array[], int target ) {
    // Assume the target is not in the array
    int position = -1;
```

continued

```
    // Step through the array until the target is found
    // or the entire array has been searched.
    for ( int i = 0; i < array.length && position == -1; i++) {
        // Is the current element what we are looking for?
        if ( array[ i ] == target ) {
            position = i;
        }
    }

    // Return the element's position
    return position;
}
```

[FIGURE 5-4] The sequential search algorithm

The critical operation in this algorithm is the comparison `if (array[i] == target)...`. This operation is the heart of the search procedure; being inside the **for** loop, it is executed as often as any other operation in the algorithm. In the average case, the target is somewhere in the middle of the list, and the algorithm will do roughly $n/2$ comparisons, which is $O(n)$. In the worst case, which occurs when the target is not in the list, the comparison must be performed n times. Therefore, the sequential search algorithm of Figure 5-4 has both an average-case and a worst-case behavior of $O(n)$. This is called a **linear algorithm**. If the list were to double in size, the time required to search it using the algorithm in Figure 5-4 would also approximately double.

We must say approximately *rather than* exactly *because the computation does not account for the contributions of statements outside the critical section.*

Linear time algorithms are common in computer science. For example, the time it takes to sum the elements in a one-dimensional array is a linear function of the array size. Similarly, the time needed to insert a new value at the beginning of an n-element array is $O(n)$, as we must move all n elements forward one position to make room for the new value.

In general, the structure of a linear time algorithm looks like the following:

```
for (i = 0; i < n; i++) {      // a loop executed n times,
                               // where n is the problem size
    S;                         // S is the critical section
}
```

Linear algorithms are generally quite efficient because solution time grows at the same rate as the increase in problem size. However, remember that there is a "hidden" constant in the big-O expression that may be quite large. For example, the actual relationship between

t and n in an O(n) linear algorithm may be $t = 1{,}000{,}000{,}000\ n$. In this case, it would be faster for a small n to use an algorithm whose running time is $t = 100\ n^2$, even though n is a lower order than n^2. All that asymptotic analysis tells us is that *eventually* there will be a problem size N_0, such that for all problems of size $n > N_0$, a linear algorithm to solve a specific problem always runs faster than a quadratic algorithm that solves the same problem. It does not say where the point N_0 lies. (In this example, $N_0 = 10^7$.)

If the list we are searching is sorted, we can do much better than O(n) by using the well-known **binary search** technique. Figure 5-5 shows an iterative version of the binary search algorithm.

```
/**
 * Search the given array for the given integer using a binary
 * search.  This method assumes that the elements in the array
 * are in sorted order.  If the element is found, the method
 * returns the position of the element; otherwise it returns -1.
 *
 * @param array The array of integers to search
 * @param target The integer to search for
 * @return target's position if found, -1 otherwise
 */

public static int search( int array[], int target ) {
    int start = 0;                  // The start of the search region
    int end = array.length - 1;     // The end of the search region
    int position = -1;              // Position of the target

    // While there is still something left in the list to search
    // and the element has not been found
    while ( start <= end && position == -1 ) {
        int middle = (start + end) / 2;  // Location of the middle

        // Determine whether the target is smaller than, greater than,
        // or equal to the middle element
        if ( target < array[ middle ] )  {
            // Target is smaller; must be in left half
             end = middle - 1;
        } else  if ( target > array[ middle ] ) {
            // Target is larger; must be in right half
            start = middle + 1;
        } else {
            // Found it!
            position = middle;
        }
    }
```

continued

```
        // Return location of target
        return position;
    }
```

[FIGURE 5-5] The binary search algorithm

The algorithm works by comparing the target for which we are searching with the item in the middle position of the array. If it matches, we have found the desired item, and we are finished. If not, we ignore the half of the array that cannot contain the desired target and repeat the process. Eventually we find what we want or eliminate all the elements. With each iteration of the loop we (approximately) halve the size of the array being searched—in other words, it becomes 1/2, 1/4, 1/8, and so on of its original size. In the worst case (when the target is not in the list), the process continues until the length of the list still under consideration is zero. If we consider the comparison `if (target < array[middle])...` to be the critical operation, then in the worst case, the maximum number of comparisons required by the binary search method is k, where k is the first integer such that:

$$2^k > n \qquad \text{(where } n \text{ is the size of the list)}$$

Another way to write this is:

$$k = \text{ceiling}(\log_2 n)$$

where ceiling() is the smallest integer larger than or equal to $\log_2 n$ and the efficiency of the binary search algorithm is $O(\log n)$. The time needed to find an element in a list using a binary search is proportional to the logarithm of the list size, rather than to the size of the list itself. This is a lower-order algorithm than the $O(n)$ sequential search because the function $(\log n)$ grows more slowly than n. From Section 5.2, we know there always exists a problem size N_0 such that for all sorted lists of size $n > N_0$, the binary search algorithm always performs more efficiently than the sequential search algorithm.

In general, the structure of an algorithm that displays logarithmic behavior is:

```
x = n;                    // n is the problem size
while (x > 0) and "the problem is not solved"   {
       S;               //  the critical operation
       x = x / k        //  reduce problem to 1/k its current size
}
```

We start with a problem of size n and, in each iteration of the loop, we reduce the problem to $1/k$ its current size until we either reach a size of 0 or the problem is solved. In the worst case, this requires a maximum of $(\log_k n)$ iterations.

$\log_k$ n = O($\log_2$ n) because $\log_k$ n and $\log_2$ n differ only by a constant value.

We can even improve on logarithmic behavior for performing a search operation. In Chapter 8 we present a search technique called *hashing*, which can theoretically reduce the time needed to locate a target in a table of size n to O(1). This is called a **constant time algorithm**, which means that it can locate any item in a list in a fixed amount of time, independent of list size. Constant time algorithms are obviously the best possible class of algorithms because the time needed to solve a problem never increases, regardless of how large it becomes.

Constant time algorithms may seem impossible, but they are not uncommon in computer science. For example, the time it takes to retrieve an element from a two-dimensional array:

```
value = X[i, j];
```

is independent of the size of the array X. The previous retrieval operation takes the same amount of time whether X is (3 × 3) or (3000 × 3000). Similarly, the time it takes to insert a new value x at the end of an array (assuming the array is not full) is O(1):

```
A[i+1] = x;  // assume the last element is currently in location i
```

The model for a constant time algorithm is extremely simple:

```
S;           // S is critical section and does not include a loop
```

Recall from our earlier discussion of binary searches that the list we search must be sorted before we can apply the algorithm. One of the best-known sorting algorithms (although one of the worst, as we will soon see) is called a **bubble sort**, as shown in Figure 5-6.

```
/**
 * Sorts the given array using the bubble sort algorithm
 *
 * @param array The array of integers to sort
 */
public static void bubbleSort( int array[] ) {
    int i = 0;      // How many elements are sorted—initially none
    boolean swap;   // Was a swap made during this pass?
    int temp;       // Temporary storage for swap
```

continued

```
    // Keep making passes through the array until it is sorted
    do {
        swap = false;

        // Make a pass through the array, swap adjacent elements that
        // are out of order
        for ( int j = 0; j < array.length - i - 1; j++ ) {
            // If the two elements are out of order, swap them
            if ( array[ j ] > array[ j + 1 ] ) {
                temp = array[ j ];
                array[ j ] = array[ j + 1 ];
                array[ j + 1 ] = temp;

                // Made a swap—array might not be sorted
                swap = true;
            }
        }

        // One more element is in the correct position
        i = i + 1;
    } while ( swap );
}
```

[FIGURE 5-6] The bubble sort algorithm

This algorithm works by interchanging adjacent elements in the list if they are out of order. After one complete pass through the list, the largest item will be in its correct location at the end of the list (assuming we are sorting into ascending order). After making a pass through the list, the algorithm checks to see if any exchanges were made. If not, the list is sorted, and we are finished. If at least one exchange was made, we must make another pass through the list. We repeat this process until no exchanges occur.

Let the critical operation again be the comparison `array[j] > array[j+1]`. In the worst case, we have to make $(n - 1)$ complete passes through the list, each time comparing adjacent pairs of values from `array[0]` through `array[j]`, where $j = n, n - 1, n - 2, \dots, 2$. In the worst case, the total number of comparison operations needed to sort the list will be:

$$\sum_{j=2}^{n} j = \left[n\left(n+1\right)/2\right] - 1 = \left(1/2\right)n^2 + \left(1/2\right)n - 1$$

As mentioned earlier, when analyzing algorithms we can ignore the contributions of both constant factors and lower-order terms. For example, if our analysis of an algorithm leads to a polynomial of the form:

$$t = an^k + bn^{k-1} + cn^{k-2} + \dots$$

we say that the algorithm is $O(n^k)$. The constant factor a only changes the location of the point where this algorithm becomes faster than a higher-order algorithm (in other words, the point 2.0 in Figure 5-1); it does not change the shape of the curve or our conclusions about the algorithm's inherent efficiency. Similarly, because we only care about the behavior of the algorithm as n becomes large, we can disregard all lower-order terms because, for large values of n, $n^k >> n^{k-1} >> n^{k-2} >>$ ....

The following theorem proves that ignoring these values does not affect our determination of the correct complexity class of an algorithm:

- Theorem: If the running time of algorithm A is $t_A = a_0 n^m + a_1 n^{m-1} + a_2 n^{m-2} + \ldots + a_m$, where $n > 1$ is the problem size, then the complexity of algorithm A is $O(n^m)$.
- Proof: $t_A = a_0 n^m + a_1 n^{m-1} + a_2 n^{m-2} + \ldots + a_m$, as specified in the theorem.

By the triangle inequality, which states that $|a + b| \leq |a| + |b|$, we can rewrite the previous equation as an inequality. Because the problem size n must be positive, we do not need to include the n^k terms inside the absolute value symbols:

$$| t_A | \leq |a_0|\, n^m + |a_1|\, n^{m-1} + |a_2|\, n^{m-2} + \ldots + |a_m|$$

Next, we divide and multiply the right side of this inequality by n^m. This operation does not change the inequality:

$$| t_A | \leq (|a_0| + |a_1|/n + |a_2|/n^2 + \ldots + |a_m|/n^m) \times n^m$$

Because $n > 1$, we can remove the n^k terms in the denominator, and the inequality remains valid:

$$| t_A | \leq (|a_0| + |a_1| + |a_2| + \ldots + |a_m|) \times n^m \qquad \text{for all } n > 1$$

The original definition of complexity from the beginning of this chapter stated that for t to be order $O[f(n)]$, there must be constants M and N_0 such that $t \leq Mf(n)$ for all $n > N_0$. If we set $M = (|a_0| + |a_1| + |a_2| + \ldots + |a_m|)$, $N_0 = 1$, and $f(n) = n^m$, we have met all the requirements of this definition and have proven that algorithm A is $O(n^m)$.

However, remember that because we are disregarding both constant factors and lower-order terms, we must be careful about any claims concerning the specific efficiency of one algorithm over another. For example, if algorithm A is $O(n)$ and algorithm B is $O(n^2)$, we cannot say that algorithm A is *always* superior to algorithm B, but only that it will *eventually* be superior to algorithm B. The actual complexity of algorithm A may be $t = 1000n + 500$, while the actual complexity of B may be $t = 0.01n^2$. In this example, the quadratic algorithm is actually faster than the linear one for all problems up to size $n = 100{,}000$. Beyond that point, however, the inherent efficiency of the linear method begins to dominate. At a problem size of $n = 10{,}000{,}000$, the linear algorithm becomes 100 times faster than the quadratic one!

Using the preceding theorem, we can state that the bubble sort algorithm of Figure 5-6, which requires $(1/2)n^2 + (1/2)n - 1$ comparisons, is $O(n^2)$, and is an example of a **quadratic** algorithm. The typical structure of a quadratic algorithm has the critical section inside two nested loops, each of which is executed on the order of n times, where n is the problem size:

```
for (i = 0;  i < n;  i++)           // n is the problem size
    for (j = 0;  j < n;  j++)
           S;                       // S is the critical section
```

Quadratic complexities are typical of a large class of sorting algorithms introduced in beginning programming courses, because they are generally easy to understand and implement. Besides the bubble sort, this class includes the insertion sort, exchange sort, and selection sort.

The performance improvements that accrue by switching from one algorithm to another of the same complexity are usually small—certainly smaller than the gains from switching to an asymptotically superior algorithm. For example, the **selection sort** is another $O(n^2)$ sorting algorithm that is generally superior to a bubble sort. We might be tempted to think we would gain significantly by choosing this new method in place of a bubble sort. Let's test our theory.

A selection sort works by searching the unsorted portion of an array from beginning to end (in other words, from positions 1 to n initially) to find the largest value. It then swaps the largest value for the one in the last position of the array. This puts the largest element into its correct position at the end and leaves positions 1 to $(n - 1)$ as the remaining unsorted portion of the array. Next, we sort a second time, searching the unsorted position of the array in positions 1 to $(n - 1)$ and swapping the largest value found for the value in position $(n - 1)$. This puts the second-largest value in its correct position and reduces the unsorted portion of the array to locations 1 to $(n - 2)$. After $(n - 1)$ repetitions, the values in positions 2 through n are all in the correct slots, so the value in position 1 must also be correct, leaving the entire array sorted. Pictorially, here is an example of how a selection sort works—the new value being swapped into its correct location is shown in boldface:

10	27	13	6	15	Original data
10	15	13	6	**27**	After pass 1
10	6	13	**15**	27	After pass 2
10	6	**13**	15	27	After pass 3
6	**10**	13	15	27	After pass 4

When the last four items in the list are in their correct location, the entire list is sorted. The code for the selection sort algorithm is shown in Figure 5-7.

```
/**
* Sorts the given array using the selection sort algorithm
*
* @param array The array of integers to sort
*/
public static void selectionSort( int array[] ) {
    int max;       // The index of the maximum element in the array
    int end;       // Keep track of the unsorted portion
    int temp;      // Temporary holder for swap

    end = array.length; // Set the end to the length of the array
                        // since it is completely unsorted

    // Loop through the array, finding the maximum element and
    // placing it at the end of the array
    for ( int i = array.length - 1; i >= 0; i--){

        // Initialize max to the first element
        max = 0;

        // Loop through the unsorted portion of the array and find
        // the maximum element of that section
        for ( int j = 0; j < end; j++ ) {

            // If the current element is greater than max, set max
            // to the index of the current element
            if ( array[ j ] > array[ max ] ) {
                max = j;
            }
        }

        // Decrement end to decrease the size of the unsorted
        // portion
        end = end - 1;

        // Swap the maximum element with the element on the end of
        // the unsorted portion (marked with the end variable)
        temp = array[ end ];
        array[ end ] = array[ max ];
        array[ max ] = temp;
    }
}
```

[FIGURE 5-7] The selection sort algorithm

Finding the largest value in the array the first time requires $(n - 1)$ comparisons, as we must compare the first value against the remaining $(n - 1)$ values in the array. Finding the second-largest value requires $(n - 2)$ comparisons, and so on. Thus, the total number of comparisons required to sort an array of size n using the selection sort is:

$$\sum_{i=1}^{n-1} i = (n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = (1/2)n^2 - (1/2)n$$

This compares favorably with a bubble sort, which required $(1/2)n^2 + (1/2)n - 1$ comparisons. However, how much difference does this improvement actually make? If, for example, we assume that $n = 100$, then a bubble sort requires $0.5 \times (100)^2 + 0.5 \times (100) - 1 = 5049$ comparisons, while a selection sort would need $0.5 \times (100)^2 - 0.5 \times (100) = 4950$. Thus, we might see a gain of about two percent—not very significant. (The exact amount of the improvement depends on noncritical sections of code whose contributions were not included in the analysis.) This is fairly typical when switching from one algorithm to another of the same complexity. If you want to make significant performance gains, you need to discover an asymptotically superior method.

Fortunately, we can do much better than $O(n^2)$ for sorting. A large group of algorithms, including Quicksort, merge sort, and heap sort, have running times proportional not to n^2 but to the function $(n \log n)$. This is a much better efficiency, and these algorithms typically run much faster than the $O(n^2)$ algorithms just mentioned.

We will determine the complexity of recursive sorting algorithms, such as Quicksort, merge sorts, and heap sort, in the following section and the succeeding chapter.

Figure 5-8 shows the number of operations needed to sort lists of different sizes using these two classes of sorting techniques. (This table ignores lower-order terms and assumes that all constant factors are 1.)

	MAXIMUM NUMBER OF COMPARISONS NEEDED TO SORT A LIST OF SIZE *n*	
n	**$O(n^2)$**	**$O(n \log_2 n)$**
10	100	33
100	10,000	670
1000	1,000,000	10,000
100,000	10^{10}	1.7×10^6
1,000,000	10^{12}	2×10^7

[FIGURE 5-8] Comparison of $O(n^2)$ and $O(n \log n)$ complexity classes

When n is small (for example, $n < 100$), the difference between the two complexity classes in Figure 5-8 is small, and the choice of sorting method is relatively unimportant. (In fact, for problems this small, sorting by hand might actually be fastest.) As n increases, however, the speed advantages of the $O(n \log n)$ algorithms become more apparent. When sorting a list of one million items, an $O(n \log n)$ algorithm requires almost five orders of magnitude fewer comparisons to sort the list than the simpler but slower $O(n^2)$ methods.

An example can clarify this point even further. Assume that we run one program from each of the two groups of sorting algorithms listed in Figure 5-8 and a third "compromise" $O(n^{1.5})$ algorithm on three very different computers: a \$1000 laptop, a \$100,000 mainframe, and a \$10,000,000 supercomputer. The supercomputer runs the inefficient $O(n^2)$ method, the mainframe executes the compromise $O(n^{1.5})$ algorithm, and the inexpensive laptop is assigned the efficient $O(n \log n)$ technique. Furthermore, assume that we were able to develop the following exact formulas for the running time of the algorithms (in μseconds) on each machine:

$t_1 = n^2$	For the supercomputer
$t_2 = 1000\, n^{1.5}$	For the mainframe computer
$t_3 = 10{,}000\, n \log n$	For the laptop

Notice how much larger the constants are for the mainframe and the laptop, reflecting that they are three to four orders of magnitude slower than the high-end supercomputer. Figure 5-9 shows the hypothetical results of running these three classes of sorting algorithms on the three types of computers.

	APPROXIMATE RUNNING TIMES		
n	Supercomputer $t = n^2$	Mainframe $t = 1000\, n^{1.5}$	Laptop $t = 10{,}000\, n \log_2 n$
100	0.01 second	1 second	7 seconds
1,000	1 second	31 seconds	100 seconds
10,000	1.7 minutes	17 minutes	22 minutes
100,000	3 hours	9 hours	5 hours
1,000,000	12 days	12 days	2 days
10,000,000	3 years	10 months	27 days
100,000,000	318 years	32 years	10 months

[FIGURE 5-9] Run-time comparisons of three classes of algorithms on three types of computers

For $n = 100$, the supercomputer's immense speed allows it to overcome the limitations of its inefficient method. It solves the problem 100 times faster than the mainframe and 700 times faster than the laptop. However, as the problem gets larger, this difference begins to disappear, and the inefficiencies start to overwhelm the larger machine's ability to keep up with the rapidly growing number of computations. At $n = 100{,}000$, all three computers take roughly the same amount of time to solve the problem—a few hours. At $n = 10{,}000{,}000$, the efficiency of the $O(n \log n)$ algorithm has become dominant, and the laptop is completing the sorting task 40 times faster than the supercomputer, a machine that costs 10,000 times more and runs thousands of times faster. When the problem size reaches $n = 100{,}000{,}000$, a supercomputer would not complete its task for more than three centuries!

Rewriting the $O(n^2)$ algorithm or buying a faster processor for the mainframe may postpone the problem, but it does not disappear. A fundamental property of a lower-order algorithm is that it always has a point (about $n = 100{,}000$ in this example) beyond which the lower-order algorithm always takes less time to complete, regardless of the constants of proportionality.

As a final example, we analyze the complexity of matrix multiplication, $C = A \times B$, defined as follows:

$$C_{ij} = \sum_{k=1}^{n} \left(A_{ik} \times B_{kj} \right) i = 1,\ldots,n; j = 1,\ldots,n$$

The critical operations are the additions and multiplications needed to produce the result. If we assume that both A and B are $n \times n$ matrices, then the preceding formula shows that the computation of each element of the product matrix C requires $(2n - 1)$ operations—n multiplications and $(n - 1)$ additions. These $(2n - 1)$ operations must be repeated for each of the $n \times n$ positions in the resulting matrix C. Thus, the total number of operations required to obtain C is:

$$T = n^2(2n - 1) = 2n^3 - n^2$$

We can therefore say that the complexity of matrix multiplication is $O(n^3)$, as given by the previous formula. This is a **cubic complexity** function. The general model of an algorithm with cubic complexity is:

```
for (i = 0; i < n; i++)             //  outermost loop done n times
                                    //  where n is the problem size
     for (j = 0; j < n; j++)        //  middle loop done n times
          for (k = 1; k < n; k++)   //  inner loop done n times
                S;                  //  the critical operation
```

Note that there is a faster algorithm for matrix multiplication called *Strassen's method*. The number of operations required to solve the matrix multiplication problem using this improved technique is $O(n^{2.81})$. Although the difference between n^3 and $n^{2.81}$ may not seem like much, it becomes enormously important when multiplying large matrices. For example, when $n = 1{,}000{,}000$, $n^3 = 10^{18}$ while $n^{2.81} = 7.24 \times 10^{16}$, a factor of 138. Even small differences in the complexity function can have profound effects on the overall efficiency of large problems.

SPEED TO BURN

We have repeatedly emphasized that machine speed alone cannot make an inefficient algorithm efficient. However, a blindingly fast computer *can* help scientists solve large problems in an acceptable amount of time. (In "The Mother of All Computations" earlier in this chapter, a supercomputer spent seven months solving a single problem.) Because of the rapidly increasing size of the scientific problems being studied, scientists want to design and build faster supercomputers.

The first computer to achieve a speed of 1 million floating-point operations per second, 1 **megaflop**, was the Control Data 6600 in the mid-1960s. The first machine to achieve 1 billion floating-point operations per second, 1 **gigaflop**, was the Cray X-MP in the early 1980s. In 1996, Intel Corporation announced that its ULTRA computer had successfully become the world's first **teraflop** machine. This $55 million computer contained 9072 processors and achieved a sustained computational speed of 1 trillion computations per second. The Earth Simulator, completed by a group of Japanese engineers in March 2002, contained 5120 processors and could execute more than 35 trillion calculations per second. To imagine how fast this is, consider that if all 6 billion people in the world worked together on a single problem, each person would have to carry out 6000 computations per second to equal the machine's speed. Its price tag: $350 million.

However, even 35 trillion computations per second is dwarfed by the newest "super" supercomputer—the IBM BlueGene/L at Lawrence Livermore National Labs. It contains 131,072 processors, and in 2005 it achieved a sustained computational rate of 280 trillion operations per second. It is used by the Department of Energy to study problems in molecular dynamics and materials science.

And just so you don't think computer designers are relaxing, there is already a major research effort to design and build the first **petaflop** machine, a computer capable of one thousand trillion (10^{15}) computations per second. IBM is investigating how to build a 1 million-processor petaflop version of the BlueGene supercomputer that could be working by 2007. The machine will be used to study the problem of protein folding, the biochemical process by which complex molecules in the human body are constructed by instructions in our DNA.

Most of the time complexities we have seen so far have taken the form $O(n)$, $O(n^2)$, or $O(n^3)$. Algorithms whose efficiencies are of the form $O(n^d)$ are called **polynomial algorithms** because their complexity functions are polynomial functions of relatively small degrees. The computational demands of these algorithms are usually manageable, even for large problems. However, not all algorithms are of this type. A second and very distinct group of methods are the **exponential algorithms**. For these problems, no polynomial time algorithm has yet been discovered. The typical complexity displayed by this class of algorithms is $O(2^n)$, $O(n^n)$, or $O(n!)$. The time demands of these algorithms grow extraordinarily quickly and consume vast amounts of resources, even for very small problems. In most cases, it is not feasible to solve any realistically sized problem using an exponential algorithm, no matter how clever the programmer or how fast the computer.

This class of algorithms is not just of academic interest. They occur frequently in computer science, applied mathematics, and operations research. In fact, an important area of research has developed specifically to study this category of **computationally intractable** problems. One example of such an exponential algorithm is the *traveling salesperson problem*. In this problem, one of the most famous in computer science, a salesperson must travel to n other cities, visiting each one only once, and ending up back home. This is called a *tour*. We want to determine if it is possible to make such a tour within a given mileage allowance, k. That is, the sum of all the distances traveled by the salesperson must be less than or equal to k. For example, here is a mileage chart of the distance between four cities A, B, C, and D:

	A	B	C	D
A	—	500	100	800
B	500	—	900	150
C	100	900	—	600
D	800	150	600	—

Given a mileage allowance of 1500 miles, a legal tour is possible starting at A: namely, $A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$. The total length is $500 + 150 + 600 + 100 = 1350$ miles. However, if the mileage allowance were 1000 miles, no legal tour exists.

No algorithm has been discovered that solves this problem in polynomial time. For example, an exhaustive search of all possible tours might begin by selecting any one of the N cities as its starting location. It might then select any one of the remaining $(N - 1)$ cities to visit next, then any of the remaining $(N - 2)$ cities, and so on. The total number of tours that need to be examined to see if they fall within the mileage allowance is:

$$N \times (N - 1) \times (N - 2) \times \ldots \times 1 = N!$$

The complexity of this **brute force** solution to our traveling salesperson problem is $O(N!)$. Better algorithms have been developed, but they still display this characteristic exponential growth, which makes the problem unsolvable in the general case except for the tiniest values of N. For example, if $N = 50$, a computer that could evaluate 1 billion tours per second would need approximately 1 million centuries to enumerate all possible tours. Most salespeople would not be willing to wait that long!

As this example demonstrates, exponential time methods may be described theoretically, but they are computationally impractical. In cases where no known polynomial algorithm exists, we are usually limited to achieving decent approximations, or reasonable rather than optimal solutions. These types of approximation algorithms are called **heuristics**.

5.4 Other Complexity Measures

When we write the expression $f(n) \approx O[g(n)]$, we are asserting that the function $f(n)$ is bounded above by (i.e., is less than or equal to) a function whose shape is of the form $g(n)$. Two other important types of notation are used to express different complexity relationships.

The following formula:

$$f(n) \approx \Omega\ [g(n)]$$

read as "$f(n)$ is **big-omega** of $g(n)$," means that the function $f(n)$ is bounded *below* by (i.e., is greater than or equal to) a function of the form $g(n)$. Formally, it means that if $f(n) \approx \Omega[g(n)]$, then there are positive constants M and N_0 such that $f(n) \geq Mg(n)$ for all $n > N_0$.

Big-omega notation is a way to put a lower bound on the growth rate of a function—that is, to state that an algorithm's growth rate must be at least a certain value. In a sense, it is used to state that a problem requires at least a certain amount of time to solve, and cannot be solved any faster. For example, it has been formally proven that all comparison-based sorting methods such as Quicksort, merge sort, and bubble sort require at least $(n \log n)$ operations to complete their task. Stated another way, if algorithm A is a comparison-based sorting algorithm, then its running time is $t \approx \Omega[n \log n]$. Big-omega is often used to put a lower-bound constraint on the performance of a class of algorithms.

Finally, if $f(n) \approx O[g(n)]$ and $f(n) \approx \Omega[g(n)]$, then $f(n)$ is bounded both above and below by a function whose shape is of the form $g(n)$. This relationship is expressed using the following notation:

$$f(n) \approx \Theta[g(n)]$$

which is read "$f(n)$ is **big-theta** of $g(n)$." Big-theta notation allows us to state that the run-time efficiency characteristics of two algorithms are equal, at least to within a constant factor. If the running time of algorithm A is $t_1 \approx \Theta[g(n)]$ and the running time of algorithm B is $t_2 \approx \Theta[g(n)]$, then both A and B are bounded above and below by functions whose shape is $g(n)$, and A and B are said to be **asymptotically equal**.

5.5 Recursion and the Analysis of Recursive Algorithms

5.5.1 Recursion

An important theorem in computer science states that all well-formed algorithms can be expressed using only sequential, conditional, and repetition operations. In most languages, Java included, this last type—repetition of a code block—is implemented using traditional iterative operators such as **while**, **do/while**, and **for**. However, this is not the only way, and some functional languages (such as LISP) do not even include these statements. The alternative to looping constructs is the **recursive algorithm**, in which we repeat a block of code by making a **recursive method call**. That is, we invoke yet another instance of the method we are currently executing.

For example, a simple iterative solution to the problem of summing the N elements in array X would be:

```
sum = 0;
for (int i = 0; i < N; i++)
      sum = sum + X[i];
```

No explanation is needed to understand this simple code. However, we could also solve the same problem recursively, as follows:

```
public int addByRecursion(int N)     {
     if (N == 0)
              return (X[0]);
     else
              return (X[N] + addByRecursion(N-1));
}
```

Let's see what happens when we execute this method using the three-element array X = {6, 20, 12} and the call `sum = addByRecursion(2)`.

The first invocation of the method has a parameter value of $N = 2$, so we execute the following statement:

```
return (X[2] + addByRecursion(1));   // this is executed during the
                                      // first invocation
```

To determine this value, we must execute the method again, but this time with the parameter value $N = 1$. This is called the **recursive case**. To do this, let's (symbolically) put a "marker" in the method on the line just executed, which effectively says, "When you have finished the second invocation of the method, come back to this point and continue." We then begin executing `addByRecursion(1)`.

This second invocation requires us to execute the following statement:

```
return (X[1] + addByRecursion(0));    // this is executed during the
                                      // second invocation
```

This is another recursive case, so we will repeat what we did before: put another marker in the code that tells us where to return and proceed with the third invocation of the `addByRecursion()` method, this time with the parameter $N = 0$.

This time, however, we do *not* make another recursive call. The parameter is 0, so we will execute the *nonrecursive* statement:

```
return (X[0]);
```

So, the result of the method call `addByRecursion(0)` is 6, the element in position 0 of the array. This nonrecursive statement is called a **base case**—at least one base case must be present in every recursive algorithm. Execution of a base case stops the repeated invocation of a recursive method and allows us to begin "backing out" of the solution by finishing all partially completed earlier invocations. We now back out of the solution in the reverse order in which the invocations were made. That is, the most recent invocation is the one we execute first.

Because we know the value of `addByRecursion(0)`, we can complete the most recently saved implementation of the method, `addByRecursion(1)`. This was the statement we were executing in `addByRecursion(1)`:

```
return (X[1] + addByRecursion(0));    // this is executed during the
                                      // second invocation
```

We now have the information we need to complete this statement. The result is the value $X[1] + 6 = 26$, and the second invocation, `addByRecursion(1)`, is finished. We can now return to the first invocation:

```
return (X[2] + addByRecursion(1));    // this is executed during the
                                      // first invocation
```

which is X[2] + 26 = 38. Because there are no more partially completed invocations of the method, we are finished. Our original call, `addByRecursion(2)`, has produced the value 38, the correct sum of the three values {6, 20, 12}.

Any algorithm that can be solved using iteration can be solved using a recursive solution. Recursion, which may look strange to some, is equally as powerful as any **while**, **do**, or **for** statement. So, you might ask why recursion is not used more frequently in Java. The answer has nothing to do with its power or capabilities, but with the way the language has been implemented. Like most modern programming languages, Java has been optimized for iterative solutions. The statements:

```
sum = 0;
for (int i = 0; i < N; i++)
      sum = sum + X[i];
```

only require a single extra integer variable (the loop counter *i*) and statements to increment, test, and branch back to the beginning of the loop. This is easy in Java, and the code runs rapidly. However, a recursive function such as:

```
public int addByRecursion(int N)     {
     if (N == 0)
               return (X[0]);
     else
               return (X[N] + addByRecursion(N-1));
}
```

requires much more effort in Java. Our symbolic "marker" for remembering where we left off necessitates saving the state of the recursive method on the run-time stack, creating new instances of parameters and local variables, and restarting a new instance of this method. This is more expensive in terms of Java code, and a recursive solution in Java usually runs more slowly than an iterative one.

However, we will use recursion to implement many of the data structures discussed in the following chapters. Therefore, you must understand how they are executed and, more importantly, how they are analyzed for efficiency. We discuss these topics in the next section.

ABU JA'FAR MUHAMMAD IBN MUSA AL-KHWARIZMI (780–850?)

We have used the word *algorithm* throughout this chapter. Although it is one of the most fundamental ideas in computer science, few people know the term's origins. Many people would assume that the term derives from an Old English, French, or Germanic root. The truth is much more interesting.

Algorithm is derived from the last name of Muhammad ibn Musa Al-Khwarizmi, a famous Persian mathematician, astronomer, and author of the eighth and ninth centuries. Al-Khwarizmi was a teacher at the House of Wisdom in Baghdad, a great center for education and study in the Middle East. He also authored the book *Kitab al Jabr w'al Muqabala,* which means *Rules of Restoration and Reduction*. It was one of the earliest mathematical textbooks, and its title gives us the word *algebra*, from the Arabic *al jabr*, meaning *reduction*.

In 825, Al-Khwarizmi wrote another book, *Al-Khwarizmi on the Hindu Art of Reckoning*. It described the base-10 numbering system that had recently been developed in India, along with formalized, step-by-step procedures for arithmetic operations, such as addition, subtraction, and multiplication, within the new decimal system. In the twelfth century the book was translated into Latin, the new Hindu-Arabic system was introduced to Europe, and Al-Khwarizmi's name became closely associated with the techniques. The Latin title of the book was *Algoritmi de Numero Indorum*, in which Al-Khwarizmi's last name was rendered *Algoritmi*. Eventually, the formalized procedures that he pioneered and developed became known as *algorithms* in his honor.

5.5.2 The Analysis of Recursive Algorithms

The asymptotic behavior of algorithms is used to study both iterative and recursive algorithms. However, the methods of analyzing these two types differ dramatically. With iterative methods, we focus on the loop structure of the program and count how many times a critical section of code is performed within these loops. This technique does not work for recursive algorithms because the fundamental control structure of a recursive program is not a loop but a recursive function call. For example, look at the code in Figure 5-10.

```
public int silly(int n)   {
   int a,b;
   if (n <= 1)
           return 1;
   else     {
           a = silly(n/2);
           b = silly(n/2);
           return (a + b)
   }
}
```

[FIGURE 5-10] Recursive function

It is not obvious how to determine the number of times we recursively call function `silly` as a function of the input parameter n. Determining the complexity of a recursive function is no longer a simple loop-counting operation.

The technique for analyzing recursive algorithms makes use of a mathematical construct called a **recurrence relation**. Recurrence relations typically are a pair of formulas of the following form:

$$\begin{array}{lll} \mathrm{T}(n) = f(\mathrm{T}(m)) & n > 1, m < n & \text{(the \textit{recursive} case)} \\ \mathrm{T}(1) = k & \text{where } k \text{ is a constant not dependent on } n & \text{(the \textit{base} case)} \end{array}$$

These two formulas describe the value of a function T and its parameter n in terms of the same function T, but with a simpler (i.e., smaller) value of its parameter. This is called the **recursive case**. We continue using this recursive-case formula until we have reduced the value of the parameter to 1 or some other small constant value. Then we use the second formula shown previously to directly determine the value of T(1). This is called the **base case**.

When using recurrence relations to study the behavior of recursive functions, we relate the number of times the critical section of code is executed on a problem of size n with the number of times it is executed on the smaller subproblems generated by a recursive call. For example, let $\mathrm{T}(n)$ represent the number of times the critical section is executed on a problem of size n. If we recursively call the procedure for a smaller problem of size m, we first determine the functional relationship between $\mathrm{T}(n)$ and $\mathrm{T}(m)$ and then solve the recurrence relationship we have developed. This will give us the answer.

For example, if you look at the function `silly` in Figure 5-10 and assume that the comparison `if (n <= 1)...` is the critical operation, you see that when n is less than or equal to 1, the comparison operation is performed only once. So, $\mathrm{T}(1) = 1$. If n is greater than 1, we still make the single comparison and, in addition, two recursive calls on function `silly` using a parameter that is half the size of the current problem. The number of comparisons for each of these calls can be expressed as $T(n/2)$. Therefore, you can express the total number of times the critical operation is executed as:

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

This is a recurrence relation in the format described earlier. We solve this relation by using a technique called *repeated substitution*:

$$T(n) = 2T(n/2) + 1$$

To determine the value of $T(n/2)$, we substitute $n/2$ for n into the preceding formula and get:

$$T(n/2) = 2T(n/4) + 1$$

Substituting this value back into the first equation for $T(n)$ gives:

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

We repeat the same operation, this time solving for the value of $T(n/4)$ and substituting it back into the preceding formula. These substitutions yield the following sequence of equations:

$$\begin{aligned} T(n) &= 8T(n/8) + 7 \\ &= 16T(n/16) + 15 \\ &= 32T(n/32) + 31 \\ &= \,.\,. \\ &= 2^k T(n/2^k) + (2^k - 1) \end{aligned}$$

This last line represents a general formula that describes all elements of the sequence in terms of the parameter k. You can easily check this by letting $k = 1, 2, 3, \ldots$, generating all of the expressions shown previously.

Now, let $n = 2^k$. That is, assume that the original problem size is an integral power of 2. Then:

$$\begin{aligned} T(n) &= nT(1) + n - 1 \qquad \text{(remember, } T(1) = 1\text{)} \\ &= n + n - 1 \\ &= 2n - 1 \end{aligned}$$

and the total number of operations carried out by the function `silly` on a problem of size n is $T(n) = 2n - 1$. Therefore, its complexity is $O(n)$.

As a second example, Figure 5-11 shows a recursive implementation of the binary search algorithm first shown in Figure 5-5. Our analysis of the earlier iterative version led to a complexity of $O(\log n)$. We would expect the recursive algorithm to behave similarly.

```java
/**
 * Recursively searches the given array for the given integer. If
 * found, returns its position.  If not, returns -1.
 *
 * @param array The array of integers to search
 * @param target The integer to search for
 * @param start First position included in the search range
 * @param end Last position included in the search range
 * @return target's position if found, -1 otherwise
 */
public static int search( int array[], int target,
                          int start, int end ) {
    int position = -1;    // Assume the target is not here

    // Do the search only if there are elements in the array
    if ( start <= end ) {
        // Determine where the middle is
        int middle = ( start + end ) / 2;

        if ( target < array[ middle ] )  {
            // Target is smaller than middle.  Search left half.
            position = search( array, target, start, middle - 1 );
        } else if ( target > array[ middle ] ) {
            // Target is larger than middle.  Search right half.
            position = search( array, target, middle + 1, end );
        } else {
            // Target is equal to middle-- we found it.
            position = middle;
        }
    }
    return position;
}
```

[FIGURE 5-11] A recursive implementation of a binary search

If the list has a length of less than or equal to 1, then we are finished after a single comparison. Otherwise, we must do up to two more comparisons, for a total of three, and then call the binary search procedure with a new list whose length is approximately half the size of the current list. This leads to the following recurrence relationships:

$$T(n) = 3 + T(n/2)$$
$$T(1) = 1$$

Using the method of repeated substitution yields the following sequence:

$$\begin{aligned} T(n) &= 3 + T(n/2) \\ &= 6 + T(n/4) \\ &= 9 + T(n/8) \\ &= \ldots \\ &= 3k + T(n/2^k) \qquad \text{(the general formula describing all terms)} \end{aligned}$$

Let the problem size $n = 2^k$. Then $k = \log_2 n$ and the formula becomes:

$$\begin{aligned} T(n) &= 3 \log_2 n + T(1) \\ &= 3 \log_2 n + 1 \end{aligned}$$

Thus, the recursive implementation of the binary search, like its iterative cousin, is also O(log n).

As our final example, we analyze the merge sort algorithm, which we mentioned in Section 5.3.1. The algorithm is shown in Figure 5-12.

```
/**
 * Sort an array using merge sort.
 *
 * @param array the array that contains the values to be sorted
 * @param start the start of the sorting region
 * @param end the end of the sorting region
 */
public static void mergeSort( int array[], int start, int end ) {
    int middle;  // Middle of the array
    int left;    // First element in left array
    int right;   // First element in the right array
    int temp;    // Temporary storage

    if ( start < end ) {
        // Split the array in half and sort each half
        middle = ( start + end ) /2;

        mergeSort( array, start, middle );
        mergeSort( array, middle + 1, end );

        // Merge the sorted arrays into one
        left = start;
        right = middle + 1;

        // While there are numbers in the array to be sorted
        while ( left <= middle && right <= end ) {
```

continued

```
                // If the current number in the left array
                // is larger than the current number in the right
                // array the numbers need to be moved around
                if ( array[ left ] > array[ right ] ) {
                    // Remember the first number in the right array
                    temp = array[ right ];

                    // Move the left array right one position to make
                    // room for the smaller number
                    for ( int i = right - 1; i >= left; i-- ) {
                            array[ i + 1 ] = array[ i ];
                    }

                    // Put the smaller number where it belongs
                    array[ left ] = temp;

                    // The right array and the middle need to shift right
                    right = right + 1;
                    middle = middle + 1;
                }

                // No matter what the left array moves right
                left = left + 1;
            }
        }
    }
```

[FIGURE 5-12] The merge sort algorithm

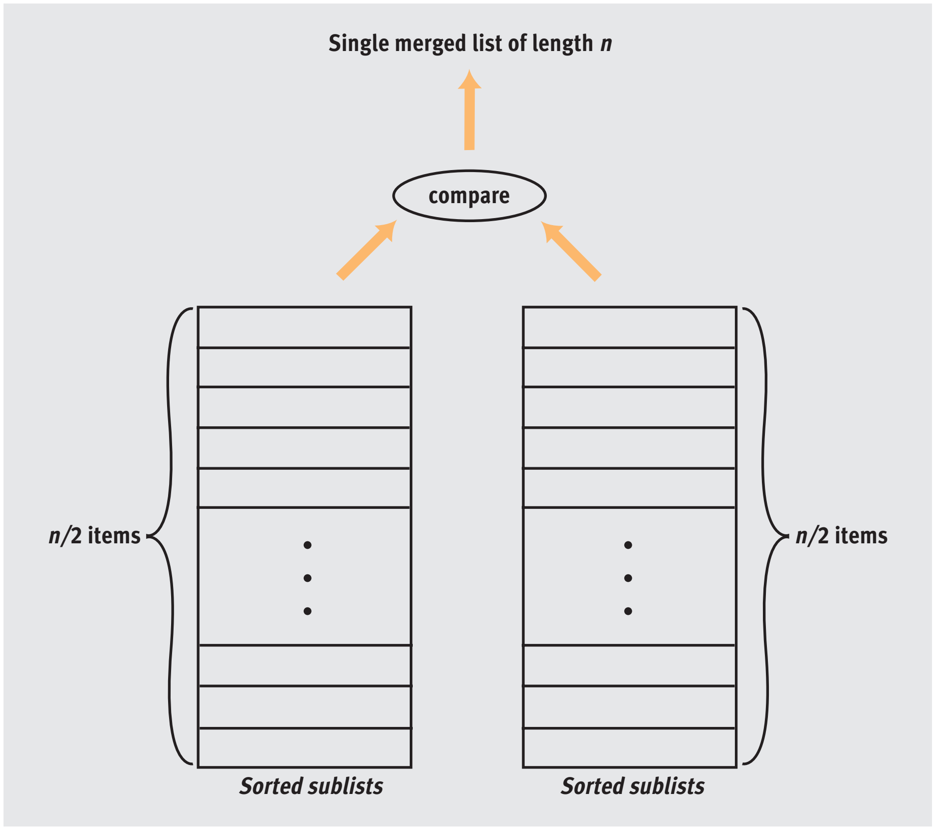
The algorithm splits the n-element list to be sorted into two lists, called `lowHalf` and `highHalf`, of approximately equal size, $n/2$. These two lists are sorted using a merge sort and then merged together to produce the final result. This recursive process continues until we have a list containing only one item, which is returned directly.

If $T(n)$ is the number of comparisons done by a merge sort on a list of length n, then the sorting phase of the merge sort requires $2T(n/2)$ comparisons. If you assume that the merge phase takes an unspecified number of comparisons called $f(n)$, then the following recurrence relation gives the total number of comparisons required by a merge sort:

$T(n) = 2T(n/2) + f(n)$ (where $f(n)$ is the number of operations required to do the merge)

$T(1) = 1$

You can determine the value of $f(n)$ by carefully analyzing Figure 5-13:



[FIGURE 5-13] Merge sort

To merge two sorted sublists into a single sorted list, we compare the top item of each sublist, select the larger of the two, and move that item into the next position in the final sorted list. Thus, a single comparison determines a single value to be moved. Because you will move a total of n items, the merge operation requires $O(n)$ comparisons and $f(n) = an$, where a is some constant value. Putting this value into our recurrence relation and again using the technique of repeated substitution yields the following:

$$\begin{aligned} T(n) &= 2T(n/2) + an \\ &= 4T(n/4) + 2an \\ &= 8T(n/8) + 3an \\ &= \ldots \\ &= 2^k T(n/2^k) + kan \qquad \text{(general formula)} \end{aligned}$$

Again, let the size of the list being sorted be $n = 2^k$. Then $k = \log_2 n$.

$$= n\mathrm{T}(1) + an \log_2 n$$
$$= n + an \log_2 n$$

As mentioned earlier, when performing an asymptotic analysis we disregard both lower-order terms (n) and constant values (a). Thus, the total number of comparisons T needed by a merge sort to sort a list of length n is $O(n \log_2 n)$.

This is a significant improvement over the $O(n^2)$ behavior of the bubble sort algorithm shown in Figure 5-6 or the selection sort method in Figure 5-7. An important theorem in computational complexity states that any sorting method based on comparing elements of a sequence must require at least $O(n \log_2 n)$ comparisons—sorting cannot be done any faster. So, to within a constant factor, a merge sort is an optimal algorithm for sorting. Another nice characteristic of a merge sort is that its worst-case behavior is also $O(n \log_2 n)$. In other words, regardless of the initial state of the list, a merge sort never requires more than $O(n \log_2 n)$ comparisons. Finally, a merge sort is a **stable sorting algorithm**, meaning that it maintains the same relative order of two objects that have equal sorting keys. If $R = S$ and R comes before S in the original list, then R will always come before S in the final sorted list. For all these reasons, the merge sort was chosen as the sorting algorithm for the Java Collection Framework. However, there are many other well-known $O(n \log_2 n)$ sorting algorithms, including Quicksort (see the Challenge Work Exercises), tree sort (Section 7.4.3), and heap sort (Section 7.6.3).

5.6 Summary

This chapter introduced you to the topic of efficiency analysis of both iterative and recursive algorithms. The technique for analyzing these algorithms is not to time them and see how fast they run. This approach is too greatly influenced by the characteristics of the selected data set, the language used to implement the algorithms, and the machine used to run them. Instead, we characterize the asymptotic or limiting behavior of an algorithm as the problem size grows very large, which allows us to determine the complexity, or order, of the algorithm. Then we can choose the lowest-order algorithm we can find. This guarantees that, to within a constant factor, we will achieve the highest possible level of efficiency.

The central point to remember about software efficiency is the critical importance of choosing the right algorithm. Selecting the best method is ultimately much more important than the programming language, the hardware, or how well we wrote the code.

Looking back at Figure 5-9, we see that using an $O(n^2)$ algorithm to solve a problem of size 10^7 took almost three years to run on a large, expensive supercomputer. No amount of clever coding or choosing a different language can reduce that running time to a reasonable value. It is an inherent problem of the technique we selected.

OBSERVATION 1: *You cannot make an inefficient algorithm efficient by your choice of implementation or machine.*

Similarly, if you select a highly efficient algorithm, almost nothing you do about its implementation can destroy that inherent efficiency. As long as the algorithm is implemented correctly, you won't gain much by spending hours poring over each line, statement, and procedure trying to squeeze out every wasted nanosecond. Tweaking program code usually has a minimal effect on its overall performance.

OBSERVATION 2: *It is virtually impossible to ruin an efficient algorithm by your choice of implementation or machine.*

By putting these two observations together, we come up with the following fundamental rule that summarizes the idea we stressed throughout this chapter: The "efficiency game" is won or lost once you select the algorithms and data structures to use in your solution. This inherent level of efficiency is not significantly affected by how well or poorly you implement the code.

This rule does not mean that we support or encourage sloppy code development. Sloppiness can detract from the legibility of your program and can impede future programmers from reading, understanding, and modifying your code. Such impediments can make it much more difficult, time consuming, and expensive to locate and correct errors. However, your coding techniques will not greatly affect the run-time efficiency of the program, which is inherent in the methods you choose. That is why your choice of algorithm is so important.

In the following chapters, we will examine a number of interesting data structures as well as algorithms for manipulating the information stored in the structures. We will use the tools presented in this chapter to select data structures that produce the highest possible level of efficiency for a given problem.

EXERCISES

1 Time complexities generally have coefficients other than 1 and have lower-order terms that are not considered. For example, the actual time or space complexity of an $O(n^3)$ algorithm might be:

$$(1/2)n^3 + 500n^2 - 1$$

However, if $f(n)$ is of a lower order than $g(n)$, there will always be a point N_0 such that, for all $n > N_0$, $O[f(n)] < O[g(n)]$. Find the point N_0 where the first complexity function is always less than the second.

a	$(1/2)n^2 + (1/2)n - 1$	$(1/8)n^3$
b	$\log_2 n$	$10{,}000n$
c	$3n^3 + 50$	$(1/50)2^{n/2}$
d	5×10^6	$(1/10{,}000)n$

2 Looking at Figure 5-4 (the sequential search), why would it be inappropriate to let the critical operation be the assignment statement

```
int retVal = -1;
```

on the first line of the program? If this statement was incorrectly called the critical operation, what would be the time complexity of the method?

3 In Figure 5-4, what other operations besides the comparison `array[i] == target` could properly be treated as the critical section and produce a correct time complexity of $O(n)$?

4 Consider the following outline of a program:

```
S1          // S1, S2, S3 are any O(1) Java statement
for (int i = 1; i < n; i++) {
     S2;
     for (int j = 1, j < n; j++)
          S3;
}
```

Assuming that n is the problem size, what would be the critical section of this program? Why?

5 What is the complexity of the following algorithmic structures with respect to problem size n? Assume that `S` is the critical operation and `a` is a constant value greater than 1.

a
```
for (i = 1; i <= n; i++)
     for (j = i; j <= i; j++)
          S;
```

b
```
for (i = 1; i <= n; i++)
     for (j = 1; j <= a; j++)
          S;
```

c
```
for (i = 1; i <= a; i++)
     for (j = i; j <= a; j++)
          S;
```

d
```
x = 1;
     do
          S;          // any statement that runs in O(1) time
          X = X * a;
     while X <= n
```

e
```
for (i = 1; i <= n; i += a)
          S;
```

6 The function e^x can be approximated using the following formula:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^k}{k!}$$

What is the complexity of this evaluation as a function of k, the number of terms in the expansion? For the critical operations, use the total number of arithmetic operations that are performed.

7 What starting conditions are necessary to produce the worst-case behavior in the bubble sort method shown in Figure 5-6?

8 Is the following argument valid?

An O(1) algorithm whose running time is independent of the problem size will always be superior to an O(n^3) algorithm whose running time grows as the cube of the problem size.

If the argument is valid, explain why. If it is not, give a counterexample.

9 A simple algorithm called *copy sort* sorts an array into ascending order. Copy sort searches an array A to find the smallest element, copies it to B[1], and "destroys" the original value in A by setting it to a very large value. The process of searching A, copying the value into the next cell of B, and destroying the value in A is repeated until the entire array has been copied, in ascending order, into array B. What is the time complexity of copy sort?

10 What is the time complexity of the following matrix multiplication operation:

$$C = A \times B$$

if A and B are no longer both $n \times n$, but A is $n \times p$ and B is $p \times m$?

11 What is the efficiency of matrix transposition for an $n \times n$ matrix? Transposition is defined as:

$$\text{Interchange}(A_{ij}, A_{ji}) \quad i = 1, \ldots, n \qquad j = 1, \ldots, i-1$$

Sketch the algorithm and analyze its complexity.

12 Assume that we have text that contains n characters $T_1, \ldots, T_n$. We also have a pattern that contains m characters $P_1, \ldots, P_m$, where $m \leq n$. We want to develop an algorithm to determine if the pattern $P_1, \ldots, P_m$ occurs as a substring anywhere within the text T. Our method is to line up P_1 with T_1 and compare up to the next m characters to see if they are all identical. If they all match, we have found our answer. If we ever encounter a mismatch, we stop the comparison. We "slide" the pattern forward, line up P_1 with T_2, and compare the next m characters. We continue in this way until we either find a match or know that no such match exists.

Sketch an algorithm for this generalized pattern-matching process and determine its time complexity.

13 **a** The following polynomial:

$$P = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

can be evaluated in many ways. The straightforward way is to perform the multiplications and additions in exactly the order specified previously:

$$P = (a_n \times x \times x \times \ldots \times x) + (a_{n-1} \times x \times \ldots \times x) + \ldots + a_0$$

Write a procedure to evaluate a polynomial with coefficients $a_0 \ldots a_n$ at point x using this straightforward technique. Determine how many multiplications, additions, and assignments are required as a function of the degree n of the polynomial. What is the time complexity of this algorithm?

b An alternative way to evaluate P is to factor the polynomial in the following manner (called **Horner's rule**):

$$P = (\ldots ((a_n \times x + a_{n-1}) \times x + a_{n-2}) \times x + \ldots + a_1) \times x + a_0$$

Write a procedure to evaluate a polynomial with coefficients $a_0 \ldots a_n$ at point x using Horner's rule. How many multiplications, additions, and assignments are required as a function of the degree n? What is the time complexity of this improved version of a polynomial evaluation algorithm?

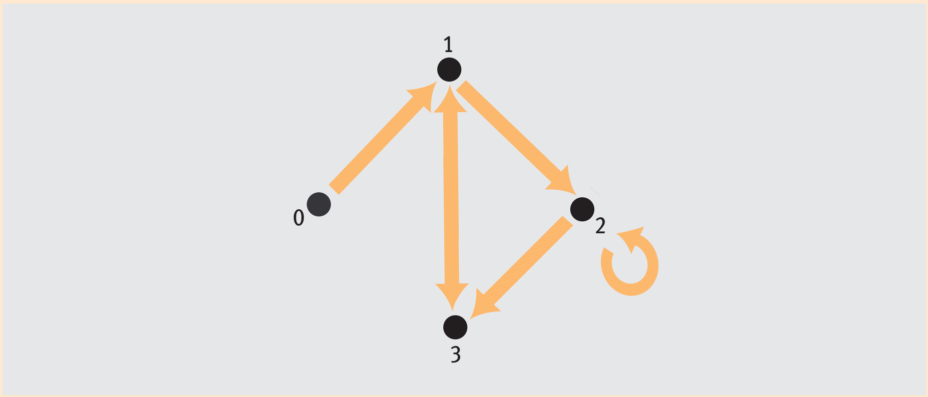
14 Assume you must develop an algorithm to determine, for a given set of cities and direct flights, whether an airline can fly directly or in multiple steps between any two arbitrarily selected cities i and j. This algorithm would be useful, for example, to help travel agents determine how to route passengers from one city to another.

You are given a matrix M[i,j] that describes the direct airline connections between cities:

M[i,j] = 1 if there is a direct connection from city i to city j.

M[i,j] = 0 if there is no direct connection from city i to city j.

Examine our example connections in Figure 5-14.



[FIGURE 5-14] Direct flight connections

M looks like the following:

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

There is a path from city 0 to city 3 ($0 \rightarrow 1 \rightarrow 3$), but there is no path from city 2 to city 0. Develop an algorithm that inputs a matrix M specifying the direct connections between n different cities. Next, your algorithm should input two indices i and j, and determine if there is a path from city i to city j. What is the time complexity of your algorithm as a function of n, the number of cities?

15 In the example of Figure 5-9, the supercomputer was assigned the inefficient $O(n^2)$ algorithm, whose actual relationship between time and size was $t = n^2$. If we changed the relationship to $t = 0.00001n^2$, would it change our conclusion? At what point (if any) would the laptop and the supercomputer demonstrate identical performance?

16 Assume that we analyzed algorithm P and found that its time complexity was $O(\log_2(\log_2 n))$. Where would that complexity function fit within the ordered list of functions in Figure 5-2?

17 Try to write the most efficient program you can to solve the following problem: You are given an array A of length n. The elements of A are all signed integers. Your program should find the subscript values i and j such that the sum of the contiguous values:

$$A[i] + A[i + 1] + \ldots + A[j - 1] + A[j]$$

is the largest possible value. For example, given the following seven-element array:

A	10	–18	–2	40	21	–5	16
index	0	1	2	3	4	5	6

your program would output the two values (3, 6) because the largest sum of values in contiguous elements of the array is contained in A[3] through A[6] (the sum is 72). After writing the program, evaluate it for efficiency in terms of its time complexity as a function of n, the size of the array.

Be careful with this problem; there are many different ways to solve it, with complexities running all the way from $O(n^3)$ to $O(n)$.

18 A matrix is called *sparse* when it has very few nonzero elements. For example, the following 4×4 matrix would be considered sparse:

$$\begin{array}{llll} 0 & 0 & 1.2 & 0 \\ 0 & 0 & 0 & 0 \\ 3.4 & 0 & 0 & 0 \\ 0 & 0 & 0 & -5.9 \end{array}$$

Rather than storing these sparse matrices in their "regular" $n \times n$ representation, which requires n^2 array elements, we can use a more space-efficient representation in which we only store information about the location of nonzero elements:

row	*column*	*value*
0	2	1.2
2	0	3.4
3	3	–5.9

a What are the space requirements of this alternate representation?

b Write a procedure called `insert` to add a new value x to position $A[i,j]$, when A is stored in the sparse matrix representation shown previously. The calling sequence of the method is **`public void`** `insert(A, i, j, x)`. Your procedure should look up the row and column indices (i,j) in the sparse representation of array A. If it is there, the procedure should change the value field to x. If it is not there, the procedure should add it so that the table is still sorted by row and column index. What is the time complexity of the insert procedure?

19 Can you design an O(1) algorithm to determine $n!$ for $1 \le n \le 25$?

20 An $n \times n$ *symmetric matrix* is one in which $A_{ij} = A_{ji}$ for $0 \le i < n$, and $0 \le j < n$. For example, the following is a 4×4 symmetric matrix.

$$\begin{array}{rrrr} 1 & 8 & -13 & 4 \\ 8 & 70 & 82 & 2 \\ -13 & 82 & 0 & -6 \\ 4 & 2 & -6 & 99 \end{array}$$

Develop a space-efficient representation for symmetric matrices that takes less than n^2 cells (assuming one integer value per cell). Next, write a method to do matrix addition in your new representation. Does it take longer than addition using the regular $n \times n$ representation? What is the time complexity of your addition method?

21 In Exercise 18 you developed a space-efficient representation for sparse matrices, and then wrote an insert program to put new information into the structure. In Exercise 20 you developed a more space-efficient representation for symmetric matrices, and then wrote a method to add two matrices that are in this representation. Compare the time complexities of these two routines with the time complexities of the methods for manipulating matrices in their regular format. What does this comparison indicate about what typically happens to program running times when you attempt to save memory space? (This relationship is usually called the **time-space trade-off**.)

22 Solve the following recurrence relation:

$$T(n) = \begin{cases} 2T(n/2) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

CHALLENGE WORK EXERCISES

Even though the merge sort is the algorithm used within the Java Collection Framework, as discussed at the end of Section 5.5.2, the world's most widely used sorting method is Quicksort, in one of its many optimized variations. Sir Charles A. R. Hoare, a world-renowned British computer scientist and professor at Oxford University, designed Quicksort in 1960. The algorithm works in three steps:

1 Pick an element from the list, called the pivot. Although it is common to select the first element in the list as the pivot, it can be any element.

2 By making exchanges, partition the list into two sublists—the first contains all elements less than the pivot value, and the second contains all elements greater than the pivot.

3 Exchange the pivot with the last element in the first sublist, which puts the pivot value into its correct position in the original list.

We now have two smaller subproblems—sorting each of the two sublists, each of which should be about half the size of the original, if we are lucky. Here is an example of how Quicksort works:

Step 1: The original data **15** 6 17 28 10 21 3
(The pivot is in boldface)

Step 2: The partition **15** $\underbrace{6 \quad 3 \quad 10}_{\text{subsequence 1}} \quad \underbrace{28 \quad 21 \quad 17}_{\text{subsequence 2}}$

Step 3: The exchange 10 6 3 **15** 28 21 17
(Exchange pivot and 10, the last value in sublist 1)

The subproblems are to sort the sublist {10, 6, 3} and the sublist {28, 21, 17}. We can solve each of these problems recursively using Quicksort.

1 Find a good article on the Web about Quicksort and how it works. A good place to start is *http://en.wikipedia.org/wiki/Quicksort*. Once you understand the algorithm, use the recurrence relation techniques introduced in this chapter, and the assumption that the pivot splits the original list into two sublists of equal size, to prove that the efficiency of Quicksort is $O(n \log n)$.

2 Determine the worst-case running time of Quicksort using the recurrence relation techniques presented. The worst case occurs when the pivot divides a list of size n into two sublists of sizes 0 and $(n - 1)$.

3 Describe how you could modify Quicksort to improve the worst-case performance problem you demonstrated in Question 2. These modifications led to the optimized variations of Quicksort that are so widely used.