Chapter 8
Algorithmic Analysis

Without analysis, no synthesis.
— Friedrich Engels, *Herr Eugen Duhring's Revolution in Science,* 1878

Objectives

- To recognize that algorithms for solving a problem vary widely in their performance.
- To understand the concept of computational complexity as a qualitative measure of how running time changes in proportion to the size of a problem.
- To learn to express computational complexity using big-O notation.
- To appreciate how divide-and-conquer techniques can improve the efficiency of sorting algorithms.
- To be able to identify the most common complexity classes and understand the performance characteristics of each class.
- To know how to prove simple properties using mathematical induction.

In Chapter 5, you were introduced to two different recursive implementations of the function **`Fib(n)`**, which computes the **`n`**th Fibonacci number. The first is based directly on the mathematical definition

$$\texttt{Fib(n)} = \begin{cases} \texttt{n} & \textit{if } \texttt{n} \textit{ is 0 or 1} \\ \texttt{Fib(n - 1) + Fib(n - 2)} & \textit{otherwise} \end{cases}$$

and turns out to be wildly inefficient. The second implementation, which uses the notion of additive sequences to produce a version of **`Fib(n)`** that is comparable in efficiency to traditional iterative approaches, demonstrates that recursion is not really the root of the problem. Even so, examples like the simple recursive implementation of the Fibonacci function have such high execution costs that recursion sometimes gets a bad name as a result.

As you will see in this chapter, the ability to think recursively about a problem often leads to new strategies that are considerably *more* efficient than anything that would come out of an iterative design process. The power of recursive divide-and-conquer algorithms is enormous and has a profound impact on many problems that arise in practice. By using recursive algorithms of this form, it is possible to achieve dramatic increases in efficiency that can cut the solution times, not by factors of two or three, but by factors of a thousand or more.

Before looking at these algorithms, however, it is important to ask a few questions. What does the term *efficiency* mean in an algorithmic context? How would you go about measuring that efficiency? These questions form the foundation for the subfield of computer science known as **analysis of algorithms.** Although a detailed understanding of algorithmic analysis requires a reasonable facility with mathematics and a lot of careful thought, you can get a sense of how it works by investigating the performance of a few simple algorithms.

8.1 The sorting problem

The easiest way to illustrate the analysis of algorithms is to consider a problem domain in which different algorithms vary widely in their performance. Of these, one of the most interesting is the problem of **sorting,** which consists of reordering the elements in an array or vector so that they fall in some defined sequence. For example, suppose you have stored the following integers in the variable **`vec`**, which is a **`Vector<int>`**:

`vec`

56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

Your mission is to write a function **`Sort(vec)`** that rearranges the elements into ascending order, like this:

`vec`

19	25	30	37	56	58	73	95
0	1	2	3	4	5	6	7

The selection sort algorithm

There are many algorithms you could choose to sort an vector of integers into ascending order. One of the simplest is called **selection sort.** Given a vector of size *N*, the selection sort algorithm goes through each element position and finds the value which should occupy that position in the sorted vector. When it finds the appropriate element, the algorithm exchanges it with the value which previously occupied the desired position to ensure that no elements are lost. Thus, on the first cycle, the algorithm finds the smallest element and swaps it with the first vector position. On the second cycle, it finds the smallest remaining element and swaps it with the second position. Thereafter, the algorithm continues this strategy until all positions in the vector are correctly ordered. An implementation of `Sort` that uses selection sort is shown in Figure 8-1.

For example, if the initial contents of the vector are

56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

the first cycle through the outer `for` loop identifies the 19 in index position 5 as the smallest value in the entire vector and then swaps it with the 56 in index position 0 to leave the following configuration:

Figure 8-1 Implementation of the selection sort algorithm

```
/*
 * Function: Sort
 * --------------
 * This implementation uses an algorithm called selection sort,
 * which can be described in English as follows.  With your left
 * hand (lh), point at each element in the vector in turn,
 * starting at index 0.  At each step in the cycle:
 *
 * 1. Find the smallest element in the range between your left
 *    hand and the end of the vector, and point at that element
 *    with your right hand (rh).
 *
 * 2. Move that element into its correct position by exchanging
 *    the elements indicated by your left and right hands.
 */

void Sort(Vector<int> & vec) {
   int n = vec.size();
   for (int lh = 0; lh < n; lh++) {
      int rh = lh;
      for (int i = lh + 1; i < n; i++) {
         if (vec[i] < vec[rh]) rh = i;
      }
      int temp = vec[lh];
      vec[lh] = vec[rh];
      vec[rh] = temp;
   }
}
```

19	25	37	58	95	56	73	30
0	1	2	3	4	5	6	7

On the second cycle, the algorithm finds the smallest element between positions 1 and 7, which turns out to be the 25 in position 1. The program goes ahead and performs the exchange operation, leaving the vector unchanged from the preceding diagram. On each subsequent cycle, the algorithm performs a swap operation to move the next smallest value into its appropriate final position. When the **`for`** loop is complete, the entire vector is sorted.

Empirical measurements of performance

How efficient is the selection sort algorithm as a strategy for sorting? To answer this question, it helps to collect empirical data about how long it takes the computer to sort an vector of various sizes. When I did this experiment some years ago on a computer that is significantly slower than machines available today, I observed the following timing data for selection sort, where *N* represents the number of elements in the vector:

N	**Running time**
10	0.12 msec
20	0.39 msec
40	1.46 msec
100	8.72 msec
200	33.33 msec
400	135.42 msec
1000	841.67 msec
2000	3.35 sec
4000	13.42 sec
10,000	83.90 sec

For an vector of 10 integers, the selection sort algorithm completes its work in a fraction of a millisecond. Even for 1000 integers, this implementation of **`Sort`** takes less than a second, which certainly seems fast enough in terms of our human sense of time. As the vector sizes get larger, however, the performance of selection sort begins to go downhill. For an vector of 10,000 integers, the algorithm requires over a minute of computing time. If you're sitting in front of your computer waiting for it to reply, a minute seems like an awfully long time.

Even more disturbing is the fact that the performance of selection sort rapidly gets worse as the vector size increases. As you can see from the timing data, every time you double the number of values in the vector, the running time increases by about a factor of four. Similarly, if you multiply the number of values by 10, the time required to sort the vector goes up a hundredfold. If this pattern continues, sorting a list of 100,000 numbers would take two and a half hours. Sorting a vector of a million numbers by this method would take approximately 10 days. Thus, if your business required you to solve sorting problems on this scale, you would have no choice but to find a more efficient approach.

Analyzing the performance of selection sort

What makes selection sort perform so badly as the number of values to be sorted becomes large? To answer this question, it helps to think about what the algorithm has to do on each cycle of the outer loop. To correctly determine the first value in the vector, the selection sort algorithm must consider all *N* elements as it searches for the smallest value.

Thus, the time required on the first cycle of the loop is presumably proportional to N. For each of the other elements in the vector, the algorithm performs the same basic steps but looks at one fewer element each time. It looks at N–1 elements on the second cycle, N–2 on the third, and so on, so the total running time is roughly proportional to

$$N + N-1 + N-2 + \ . \ . \ . + 3 + 2 + 1$$

Because it is difficult to work with an expression in this expanded form, it is useful to simplify it by applying a bit of mathematics. As you may have learned in an algebra course, the sum of the first N integers is given by the formula

$$\frac{N\,(N+1)}{2}$$

or, multiplying out the numerator,

$$\frac{N^2+N}{2}$$

You will learn how to prove that this formula is correct in the section on “Mathematical induction” later in this chapter. For the moment, all you need to know is that the sum of the first N integers can be expressed in this more compact form.

If you write out the values of the function

$$\frac{N^2+N}{2}$$

for various values of N, you get a table that looks like this:

N	$\frac{N^2+N}{2}$
10	55
20	210
40	820
100	5050
200	20,100
400	80,200
1000	500,500
2000	2,001,000
4000	8,002,000
10,000	50,005,000

Because the running time of the selection sort algorithm is presumably related to the amount of work the algorithm needs to do, the values in this table should be roughly proportional to the observed execution time of the algorithm, which turns out to be true. If you look at the measured timing data for selection sort, for example, you discover that the algorithm requires 83.90 seconds to sort 10,000 numbers. In that time, the selection sort algorithm has to perform 50,005,000 operations in its innermost loop. Assuming that there is indeed a proportionality relationship between these two values, dividing the time by the number of operations gives the following estimate of the proportionality constant:

$$\frac{83.90 \text{ seconds}}{50{,}005{,}000} = 0.00167 \text{ msec}$$

If you then apply this same proportionality constant to the other entries in the table, you discover that the formula

$$0.00167 \text{ msec} \times \frac{N^2 + N}{2}$$

is indeed a good approximation of the running time, at least for large values of *N*. The observed times and the estimates calculated using this formula are shown in Table 8-1, along with the relative error between the two.

8.2 Computational complexity and big-O notation

The problem with carrying out a detailed analysis like the one shown in Table 8-1 is that you end up with too much information. Although it is occasionally useful to have a formula for predicting exactly how long a program will take, you can usually get away with more qualitative measures. The reason that selection sort is impractical for large values of *N* has little to do with the precise timing characteristics of a particular implementation running on a specific machine. The problem is much simpler and more fundamental. At its essence, the problem with selection sort is that doubling the size of the input vector increases the running time of the selection sort algorithm by a factor of four, which means that the running time grows more quickly than the number of elements in the vector.

The most valuable qualitative insights you can obtain about algorithmic efficiency are usually those that help you understand how the performance of an algorithm responds to changes in problem size. Problem size is usually easy to quantify. For algorithms that operate on numbers, it generally makes sense to let the numbers themselves represent the problem size. For most algorithms that operate on arrays or vectors, you can use the number of elements. When evaluating algorithmic efficiency, computer scientists traditionally use the letter *N* to indicate the size of the problem, no matter how it is calculated. The relationship between *N* and the performance of an algorithm as *N* becomes large is called the **computational complexity** of that algorithm. In general, the most important measure of performance is execution time, although it also possible to apply complexity analysis to other concerns, such as the amount of memory space required. Unless otherwise stated, all assessments of complexity used in this text refer to execution time.

Table 8-1 Observed and estimated times for selection sort

N	Observed time	Estimated time	Error
10	0.12 msec	0.09 msec	23%
20	0.39 msec	0.35 msec	10%
40	1.46 msec	1.37 msec	6%
100	8.72 msec	8.43 msec	3%
200	33.33 msec	33.57 msec	1%
400	135.42 msec	133.93 msec	1%
1000	841.67 msec	835.84 msec	1%
2000	3.35 sec	3.34 sec	< 1%
4000	13.42 sec	13.36 sec	< 1%
10,000	83.90 sec	83.50 sec	< 1%

Big-O notation

Computer scientists use a special notation to denote the computational complexity of algorithms. Called **big-O notation,** it was introduced by the German mathematician Paul Bachmann in 1892—long before the development of computers. The notation itself is very simple and consists of the letter *O,* followed by a formula enclosed in parentheses. When it is used to specify computational complexity, the formula is usually a simple function involving the problem size *N*. For example, in this chapter you will soon encounter the big-O expression

$$O(N^2)$$

which is read aloud as "big-oh of *N* squared."

Big-O notation is used to specify qualitative approximations and is therefore ideal for expressing the computational complexity of an algorithm. Coming as it does from mathematics, big-O notation has a precise definition, which appears later in this chapter in the section entitled "A formal definition of big-O." At this point, however, it is far more important for you—no matter whether you think of yourself as a programmer or a computer scientist—to understand what big-O means from a more intuitive point of view.

Standard simplifications of big-O

When you use big-O notation to estimate the computational complexity of an algorithm, the goal is to provide a *qualitative* insight as to how changes in *N* affect the algorithmic performance as *N* becomes large. Because big-O notation is not intended to be a quantitative measure, it is not only appropriate but desirable to reduce the formula inside the parentheses so that it captures the qualitative behavior of the algorithm in the simplest possible form. The most common simplifications that you can make when using big-O notation are as follows:

1. *Eliminate any term whose contribution to the total ceases to be significant as N becomes large.* When a formula involves several terms added together, one of those terms often grows much faster than the others and ends up dominating the entire expression as *N* becomes large. For large values of *N,* this term alone will control the running time of the algorithm, and you can ignore the other terms in the formula entirely.
2. *Eliminate any constant factors.* When you calculate computational complexity, your main concern is how running time changes as a function of the problem size *N*. Constant factors have no effect on the overall pattern. If you bought a machine that was twice as fast as your old one, any algorithm that you executed on your machine would run twice as fast as before for every value of *N*. The growth pattern, however, would remain exactly the same. Thus, you can ignore constant factors when you use big-O notation.

The computational complexity of selection sort

You can apply the simplification rules from the preceding section to derive a big-O expression for the computational complexity of selection sort. From the analysis in the section "Analyzing the performance of selection sort" earlier in the chapter, you know that the running time of the selection sort algorithm for a vector of *N* elements is proportional to

$$\frac{N^2 + N}{2}$$

Although it would be mathematically correct to use this formula directly in the big-O expression

$$O\left(\frac{N^2 + N}{2}\right)$$

This expression is too complicated.

you would never do so in practice because the formula inside the parentheses is not expressed in the simplest form.

The first step toward simplifying this relationship is to recognize that the formula is actually the sum of two terms, as follows:

$$\frac{N^2}{2} + \frac{N}{2}$$

You then need to consider the contribution of each of these terms to the total formula as *N* increases in size, which is illustrated by the following table:

N	$\frac{N^2}{2}$	$\frac{N}{2}$	$\frac{N^2+N}{2}$
10	50	5	55
100	5000	50	5050
1000	500,000	500	500,500
10,000	50,000,000	5000	50,005,000
100,000	5,000,000,000	50,000	5,000,050,000

As *N* increases, the term involving N^2 quickly dominates the term involving *N*. As a result, the simplification rule allows you to eliminate the smaller term from the expression. Even so, you would not write that the computational complexity of selection sort is

$$O\left(\frac{N^2}{2}\right)$$

This expression includes a constant factor.

because you can eliminate the constant factor. The simplest expression you can use to indicate the complexity of selection sort is

$$O(N^2)$$

This expression captures the essence of the performance of selection sort. As the size of the problem increases, the running time tends to grow by the square of that increase. Thus, if you double the size of the vector, the running time goes up by a factor of four. If you instead multiply the number of input values by 10, the running time will explode by a factor of 100.

Predicting computational complexity from code structure

How would you determine the computational complexity of the function

```
double Average(Vector<double> & vec) {
    int n = vec.size();
    double total = 0;
    for (int i = 0; i < n; i++) {
        total += vec[i];
    }
    return total / n;
}
```

which computes the average of the elements in a vector? When you call this function, some parts of the code are executed only once, such as the initialization of **`total`** to 0 and the division operation in the **`return`** statement. These computations take a certain amount of time, but that time is constant in the sense that it doesn't depend on the size of the vector. Code whose execution time does not depend on the problem size is said to run in **constant time,** which is expressed in big-O notation as $O(1)$.[1]

There are, however, other parts of the **`Average`** function that are executed exactly **`n`** times, once for each cycle of the **`for`** loop. These components include the expression **`i++`** in the **`for`** loop and the statement

```
total += vec[i];
```

which constitutes the loop body. Although any single execution of this part of the computation takes a fixed amount of time, the fact that these statements are executed **`n`** times means their total execution time is directly proportional to the vector size. The computational complexity of this part of the **`Average`** function is $O(N)$, which is commonly called **linear time.**

The total running time for **`Average`** is therefore the sum of the times required for the constant parts and the linear parts of the algorithm. As the size of the problem increases, however, the constant term becomes less and less relevant. By exploiting the simplification rule that allows you to ignore terms that become insignificant as N gets large, you can assert that the **`Average`** function as a whole runs in $O(N)$ time.

You could, however, predict this result just by looking at the loop structure of the code. For the most part, the individual expressions and statements—unless they involve function calls that must be accounted separately—run in constant time. What matters in terms of computational complexity is how often those statements are executed. For many programs, you can determine the computational complexity simply by finding the piece of the code that is executed most often and determining how many times it runs as a function of N. In the case of the **`Average`** function, the body of the loop is executed **`n`** times. Because no part of the code is executed more often than this, you can predict that the computational complexity will be $O(N)$.

The selection sort function can be analyzed in a similar way. The most frequently executed part of the code is the comparison in the statement

```
if (vec[i] < vec[rh]) rh = i;
```

[1] Some students find the designation $O(1)$ confusing, because the expression inside the parentheses does not depend on N. In fact, this lack of any dependency on N is the whole point of the $O(1)$ notation. As you increase the size of the problem, the time required to execute code whose running time is $O(1)$ increases in exactly the same way that 1 increases; in other words, the running time of the code does not increase at all.

That statement is nested inside two **for** loops whose limits depend on the value of N. The inner loop runs N times as often as the outer loop, which implies that the inner loop body is executed $O(N^2)$ times. Algorithms like selection sort that exhibit $O(N^2)$ performance are said to run in **quadratic time.**

Worst-case versus average-case complexity

In some cases, the running time of an algorithm depends not only on the size of the problem but also on the specific characteristics of the data. For example, consider the function

```
int LinearSearch(int key, Vector<int> & vec) {
    int n = vec.size();
    for (int i = 0; i < n; i++) {
        if (key == vec[i]) return i;
    }
    return -1;
}
```

which returns the first index position in **vec** at which the value **key** appears, or –1 if the value **key** does not appear anywhere in the vector. Because the **for** loop in the implementation executes **n** times, you expect the performance of **LinearSearch**—as its name implies—to be $O(N)$.

On the other hand, some calls to **LinearSearch** can be executed very quickly. Suppose, for example, that the key element you are searching for happens to be in the first position in the vector. In that case, the body of the **for** loop will run only once. If you're lucky enough to search for a value that always occurs at the beginning of the vector, **LinearSearch** will run in constant time.

When you analyze the computational complexity of a program, you're usually not interested in the minimum possible time. In general, computer scientists tend to be concerned about the following two types of complexity analysis:

- *Worst-case complexity.* The most common type of complexity analysis consists of determining the performance of an algorithm in the worst possible case. Such an analysis is useful because it allows you to set an upper bound on the computational complexity. If you analyze for the worst case, you can guarantee that the performance of the algorithm will be at least as good as your analysis indicates. You might sometimes get lucky, but you can be confident that the performance will not get any worse.
- *Average-case complexity.* From a practical point of view, it is often useful to consider how well an algorithm performs if you average its behavior over all possible sets of input data. Particularly if you have no reason to assume that the specific input to your problem is in any way atypical, the average-case analysis provides the best statistical estimate of actual performance. The problem, however, is that average-case analysis is usually much more difficult to carry out and typically requires considerable mathematical sophistication.

The worst case for the **LinearSearch** function occurs when the key is not in the vector at all. When the key is not there, the function must complete all **n** cycles of the **for** loop, which means that its performance is $O(N)$. If the key is known to be in the vector, the **for** loop will be executed about half as many times on average, which implies that average-case performance is also $O(N)$. As you will discover in the section on "The Quicksort algorithm" later in this chapter, the average-case and worst-case performances of an algorithm sometimes differ in qualitative ways, which means that in practice it is often important to take both performance characteristics into consideration.

A formal definition of big-O

Because understanding big-O notation is critical to modern computer science, it is important to offer a somewhat more formal definition to help you understand why the intuitive model of big-O works and why the suggested simplifications of big-O formulas are in fact justified. In mathematics, big-O notation is used to express the relationship between two functions, in an expression like this:

$$t(N) = O(f(N))$$

The formal meaning of this expression is that $f(N)$ is an approximation of $t(N)$ with the following characteristic: it must be possible to find a constant N_0 and a positive constant C so that for every value of $N \leq N_0$, the following condition holds:

$$t(N) \leq C \times f(N)$$

In other words, as long as N is "large enough," the function $t(N)$ is always bounded by a constant multiple of the function $f(N)$.

When it is used to express computational complexity, the function $t(N)$ represents the actual running time of the algorithm, which is usually difficult to compute. The function $f(N)$ is a much simpler formula that nonetheless provides a reasonable qualitative estimate for how the running time changes as a function of N, because the condition expressed in the mathematical definition of big-O ensures that the actual running time cannot grow faster than $f(N)$.

To see how the formal definition applies, it is useful to go back to the selection sorting example. Analyzing the loop structure of selection sort showed that the operations in the innermost loop were executed

$$\frac{N^2 + N}{2}$$

times and that the running time was presumably roughly proportional to this formula. When this complexity was expressed in terms of big-O notation, the constants and low-order terms were eliminated, leaving only the assertion that the execution time was $O(N^2)$, which is in fact an assertion that

$$\frac{N^2 + N}{2} = O(N^2)$$

To show that this expression is indeed true under the formal definition of big-O, all you need to do is find constants C and N_0 so that

$$\frac{N^2 + N}{2} \leq C \times N^2$$

for all values of $N \geq N_0$. This particular example is extremely simple. All you need to do to satisfy the constraints is to set the constants C and N_0 both to 1. After all, as long as N is no smaller than 1, you know that $N^2 \geq N$. It must therefore be the case that

$$\frac{N^2 + N}{2} \leq \frac{N^2 + N^2}{2}$$

But the right side of this inequality is simply N^2, which means that

$$\frac{N^2 + N}{2} \leq N^2$$

for all values of $N \geq 1$, as required by the definition.

You can use a similar argument to show that any polynomial of degree *k,* which can be expressed in general terms as

$$a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \ . \ . \ . + a_2 N^2 + a_1 N + a_0$$

is $O(N^k)$. To do so, your goal is to find constants C and N_0 so that

$$a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \ . \ . \ . + a_2 N^2 + a_1 N + a_0 \leq C \times N^k$$

for all values of $N \geq N_0$.

As in the preceding example, start by letting N_0 be 1. For all values of $N \geq 1$, each successive power of N is at least as large as its predecessor, so

$$N^k \geq N^{k-1} \geq N^{k-2} \geq \ . \ . \ . \geq N^2 \geq N \geq 1$$

This property in turn implies that

$$\begin{aligned} &a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \ . \ . \ . + a_2 N^2 + a_1 N + a_0 \\ &\quad \leq |a_k| N^k + |a_{k-1}| N^k + |a_{k-2}| N^k + \ . \ . \ . + |a_2| N^k + |a_1| N^k + |a_0| N^k \end{aligned}$$

where the vertical bars surrounding the coefficients on the right side of the equation indicate absolute value. By factoring out N^k, you can simplify the right side of this inequality to

$$\left(|a_k| + |a_{k-1}| + |a_{k-2}| + \ . \ . \ . + |a_2| + |a_1| + |a_0|\right) N^k$$

Thus, if you define the constant C to be

$$|a_k| + |a_{k-1}| + |a_{k-2}| + \ . \ . \ . + |a_2| + |a_1| + |a_0|$$

you have established that

$$a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \ . \ . \ . + a_2 N^2 + a_1 N + a_0 \leq C \times N^k$$

This result proves that the entire polynomial is $O(N^k)$.

If all this mathematics scares you, try not to worry. It is much more important for you to understand what big-O means in practice than it is to follow all the steps in the formal derivation.

8.3 Recursion to the rescue

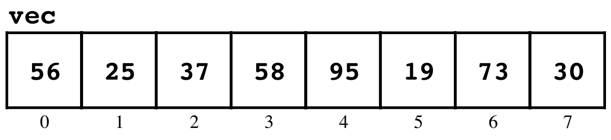
At this point, you know considerably more about complexity analysis than you did when you started the chapter. However, you are no closer to solving the practical problem of how to write a sorting algorithm that is more efficient for large vectors. The selection sort algorithm is clearly not up to the task, because the running time increases in proportion to the square of the input size. The same is true for most sorting algorithms that process the elements of the vector in a linear order. To develop a better sorting algorithm, you need to adopt a qualitatively different approach.

The power of divide-and-conquer strategies

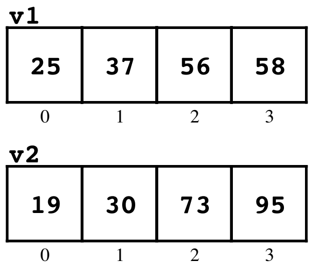
Oddly enough, the key to finding a better sorting strategy lies in recognizing that the quadratic behavior of algorithms like selection sort has a hidden virtue. The basic characteristic of quadratic complexity is that, as the size of a problem doubles, the running time increases by a factor of four. The reverse, however, is also true. If you divide the size of a quadratic problem by two, you decrease the running time by that same factor of four. This fact suggests that dividing a vector in half and then applying a recursive divide-and-conquer approach might reduce the required sorting time.

To make this idea more concrete, suppose you have a large vector that you need to sort. What happens if you divide the vector into two halves and then use the selection sort algorithm to sort each of those pieces? Because selection sort is quadratic, each of the smaller vectors requires one quarter of the original time. You need to sort both halves, of course, but the total time required to sort the two smaller vectors is still only half the time that would have been required to sort the original vector. If it turns out that sorting two halves of an vector simplifies the problem of sorting the complete vector, you will be able to reduce the total time substantially. More importantly, once you discover how to improve performance at one level, you can use the same algorithm recursively to sort each half.

To determine whether a divide-and-conquer strategy is applicable to the sorting problem, you need to answer the question of whether dividing a vector into two smaller vectors and then sorting each one helps to solve the general problem. As a way to gain some insight into this question, suppose that you start with a vector containing the following eight elements:



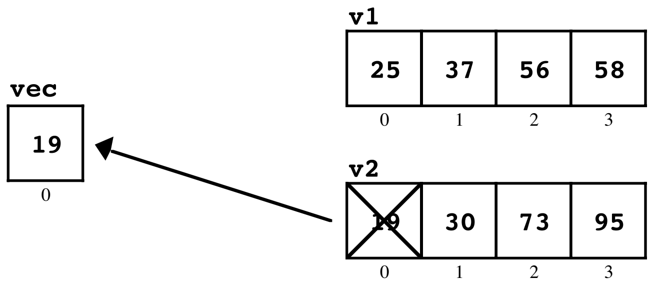
If you divide the vector of eight elements into two vectors of length four and then sort each of those smaller vectors—remember that the recursive leap of faith means you can assume that the recursive calls work correctly—you get the following situation in which each of the smaller vectors is sorted:



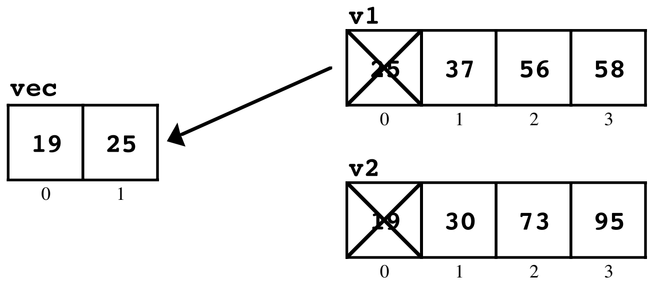
How useful is this decomposition? Remember that your goal is to take the values out of these smaller vectors and put them back into the original vector in the correct order. How does having these smaller sorted vectors help you in accomplishing that goal?

Merging two vectors

As it happens, reconstructing the complete vector from the smaller sorted vectors is a much simpler problem than sorting itself. The required technique, called **merging,** depends on the fact that the first element in the complete ordering must be either the first element in **`v1`** or the first element in **`v2`**, whichever is smaller. In this example, the first element you want in the new vector is the 19 in **`v2`**. If you add that element to an empty vector **`vec`** and, in effect, cross it out of **`v2`**, you get the following configuration:



Once again, the next element can only be the first unused element in one of the two smaller vectors. You compare the 25 from **`v1`** against the 30 in **`v2`** and choose the former:



You can easily continue this process of choosing the smaller value from **`v1`** or **`v2`** until you have reconstructed the entire vector.

The merge sort algorithm

The merge operation, combined with recursive decomposition, gives rise to a new sorting algorithm called **merge sort,** which you can implement in a straightforward way. The basic idea of the algorithm can be outlined as follows:

1. Check to see if the vector is empty or has only one element. If so, it must already be sorted, and the function can return without doing any work. This condition defines the simple case for the recursion.
2. Divide the vector into two smaller vectors, each of which is half the size of the original.
3. Sort each of the smaller vectors recursively.
4. Clear the original vector so that it is again empty.
5. Merge the two sorted vectors back into the original one.

The code for the merge sort algorithm, shown in Figure 8-2, divides neatly into two functions: **`Sort`** and **`Merge`**. The code for **`Sort`** follows directly from the outline of the algorithm. After checking for the special case, the algorithm divides the original vector into two smaller ones, **`v1`** and **`v2`**. As soon as the code for **`Sort`** has copied all of the elements into either **`v1`** or **`v2`**, have been created, the rest of the function sorts these vectors recursively, clears the original vector, and then calls **`Merge`** to reassemble the complete solution.

Figure 8-2 Implementation of the merge sort algorithm

```
/*
 * Function: Sort
 * --------------
 * This function sorts the elements of the vector into
 * increasing numerical order using the merge sort algorithm,
 * which consists of the following steps:
 *
 * 1. Divide the vector into two halves.
 * 2. Sort each of these smaller vectors recursively.
 * 3. Merge the two vectors back into the original one.
 */

void Sort(Vector<int> & vec) {
   int n = vec.size();
   if (n <= 1) return;
   Vector<int> v1;
   Vector<int> v2;
   for (int i = 0; i < n; i++) {
      if (i < n / 2) {
         v1.add(vec[i]);
      } else {
         v2.add(vec[i]);
      }
   }
   Sort(v1);
   Sort(v2);
   vec.clear();
   Merge(vec, v1, v2);
}

/*
 * Function: Merge
 * ---------------
 * This function merges two sorted vectors (v1 and v2) into the
 * vector vec, which should be empty before this operation.
 * Because the input vectors are sorted, the implementation can
 * always select the first unused element in one of the input
 * vectors to fill the next position.
 */

void Merge(Vector<int> & vec, Vector<int> & v1, Vector<int> & v2) {
   int n1 = v1.size();
   int n2 = v2.size();
   int p1 = 0;
   int p2 = 0;
   while (p1 < n1 && p2 < n2) {
      if (v1[p1] < v2[p2]) {
         vec.add(v1[p1++]);
      } else {
         vec.add(v2[p2++]);
      }
   }
   while (p1 < n1) vec.add(v1[p1++]);
   while (p2 < n2) vec.add(v2[p2++]);
}
```

Most of the work is done by the **`Merge`** function, which takes the destination vector, along with the smaller vectors **`v1`** and **`v2`**. The indices **`p1`** and **`p2`** mark the progress through each of the subsidiary vectors. On each cycle of the loop, the function selects an element from **`v1`** or **`v2`**—whichever is smaller—and adds that value to the end of **`vec`**. As soon as the elements in either of the two smaller vector are exhausted, the function can simply copy the elements from the other vector without bothering to test them. In fact, because one of these vectors is already exhausted when the first **`while`** loop exits, the function can simply copy the rest of each vector to the destination. One of these vectors will be empty, and the corresponding **`while`** loop will therefore not be executed at all.

The computational complexity of merge sort

You now have an implementation of the **`Sort`** function based on the strategy of divide-and-conquer. How efficient is it? You can measure its efficiency by sorting vectors of numbers and timing the result, but it is helpful to start by thinking about the algorithm in terms of its computational complexity.

When you call the merge sort implementation of **`Sort`** on a list of N numbers, the running time can be divided into two components:

1. The amount of time required to execute the operations at the current level of the recursive decomposition
2. The time required to execute the recursive calls

At the top level of the recursive decomposition, the cost of performing the nonrecursive operations is proportional to N. The loop to fill the subsidiary vectors accounts for N cycles, and the call to **`Merge`** has the effect of refilling the original N positions in the vector. If you add these operations and ignore the constant factor, you discover that the complexity of any single call to **`Sort`**—not counting the recursive calls within it—requires $O(N)$ operations.

But what about the cost of the recursive operations? To sort an vector of size N, you must recursively sort two vectors of size $N / 2$. Each of these operations requires some amount of time. If you apply the same logic, you quickly determine that sorting each of these smaller vectors requires time proportional to $N / 2$ at that level, plus whatever time is required by its own recursive calls. The same process then continues until you reach the simple case in which the vectors consist of a single element or no elements at all.

The total time required to solve the problem is the sum of the time required at each level of the recursive decomposition. In general, the decomposition has the structure shown in Figure 8-3. As you move down through the recursive hierarchy, the vectors get smaller, but more numerous. The amount of work done at each level, however, is always directly proportional to N. Determining the total amount of work is therefore a question of finding out how many levels there will be.

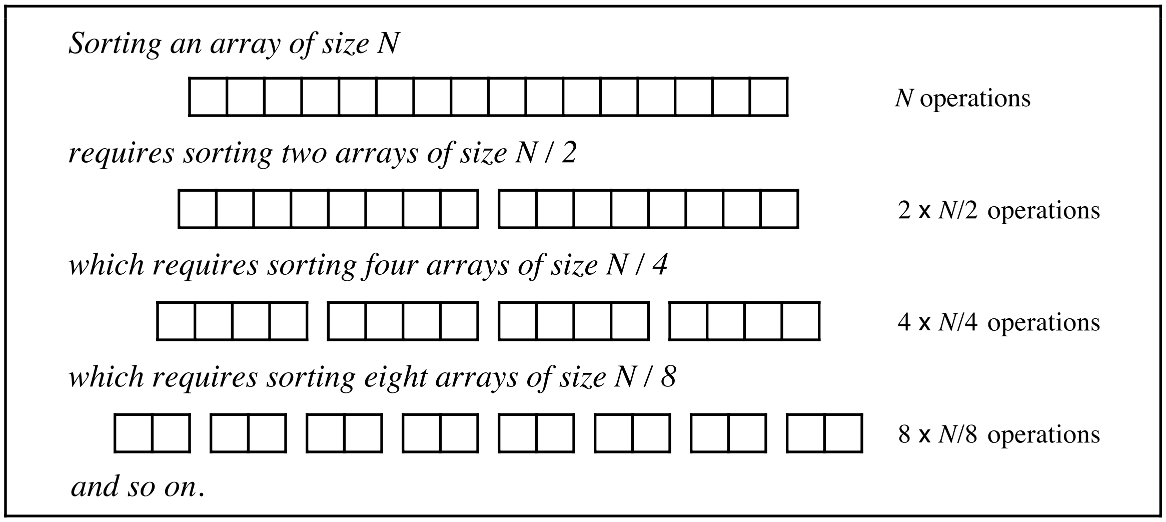
At each level of the hierarchy, the value of N is divided by 2. The total number of levels is therefore equal to the number of times you can divide N by 2 before you get down to 1. Rephrasing this problem in mathematical terms, you need to find a value of k such that

$$N = 2^k$$

Solving the equation for k gives

$$k = \log_2 N$$

Figure 8-3 Recursive decomposition of merge sort


Sorting an array of size N
N operations
requires sorting two arrays of size N / 2
2 x *N*/2 operations
which requires sorting four arrays of size N / 4
4 x *N*/4 operations
which requires sorting eight arrays of size N / 8
8 x *N*/8 operations
and so on.

Because the number of levels is $\log_2 N$ and the amount of work done at each level is proportional to N, the total amount of work is proportional to $N \log_2 N$.

Unlike other scientific disciplines, in which logarithms are expressed in terms of powers of 10 (common logarithms) or the mathematical constant e (natural logarithms), computer science tends to use **binary logarithms,** which are based on powers of 2. Logarithms computed using different bases differ only by a constant factor, and it is therefore traditional to omit the logarithmic base when you talk about computational complexity. Thus, the computational complexity of merge sort is usually written as

$$O(N \log N)$$

Comparing N^2 and $N \log N$ performance

But how good is $O(N \log N)$? You can compare its performance to that of $O(N^2)$ by looking at the values of these functions for different values of N, as follows:

N	N^2	$N \log N$
10	100	33
100	10,000	664
1000	1,000,000	9965
10,000	100,000,000	132,877

The numbers in both columns grow as N becomes larger, but the N^2 column grows much faster than the $N \log N$ column. Sorting algorithms based on an $N \log N$ algorithm are therefore useful over a much larger range of vector sizes.

It is interesting to verify these results in practice. Because big-O notation discards constant factors, it may be that selection sort is more efficient for some problem sizes. Running the selection and merge sort algorithms on vectors of varying sizes and measuring the actual running times results in the timing data shown in Table 8-2. For 10 items, this implementation of merge sort is more than four times slower than selection sort. At 40 items, selection sort is still faster, but not by very much. By the time you get up to 10,000 items, merge sort is almost 100 times faster than selection sort. On my computer, the selection sort algorithm requires almost a minute and a half to sort 10,000 items while merge sort takes a little under a second. For large vectors, merge sort clearly represents a significant improvement.

Table 8-2 Empirical comparison of selection and merge sorts

N	**Selection sort**	**Merge sort**
10	0.12 msec	0.54 msec
20	0.39 msec	1.17 msec
40	1.46 msec	2.54 msec
100	8.72 msec	6.90 msec
200	33.33 msec	14.84 msec
400	135.42 msec	31.25 msec
1000	841.67 msec	84.38 msec
2000	3.35 sec	179.17 msec
4000	13.42 sec	383.33 msec
10,000	83.90 sec	997.67 msec

8.4 Standard complexity classes

In programming, most algorithms fall into one of several common complexity classes. The most important complexity classes are shown in Table 8-3, which gives the common name of the class along with the corresponding big-O expression and a representative algorithm in that class.

The classes in Table 8-3 are presented in strictly increasing order of complexity. If you have a choice between one algorithm that requires $O(\log N)$ time and another that requires $O(N)$ time, the first will always outperform the second as N grows large. For small values of N, terms that are discounted in the big-O calculation may allow a theoretically less efficient algorithm to do better against one that has a lower computational complexity. On the other hand, as N grows larger, there will always be a point at which the theoretical difference in efficiency becomes the deciding factor.

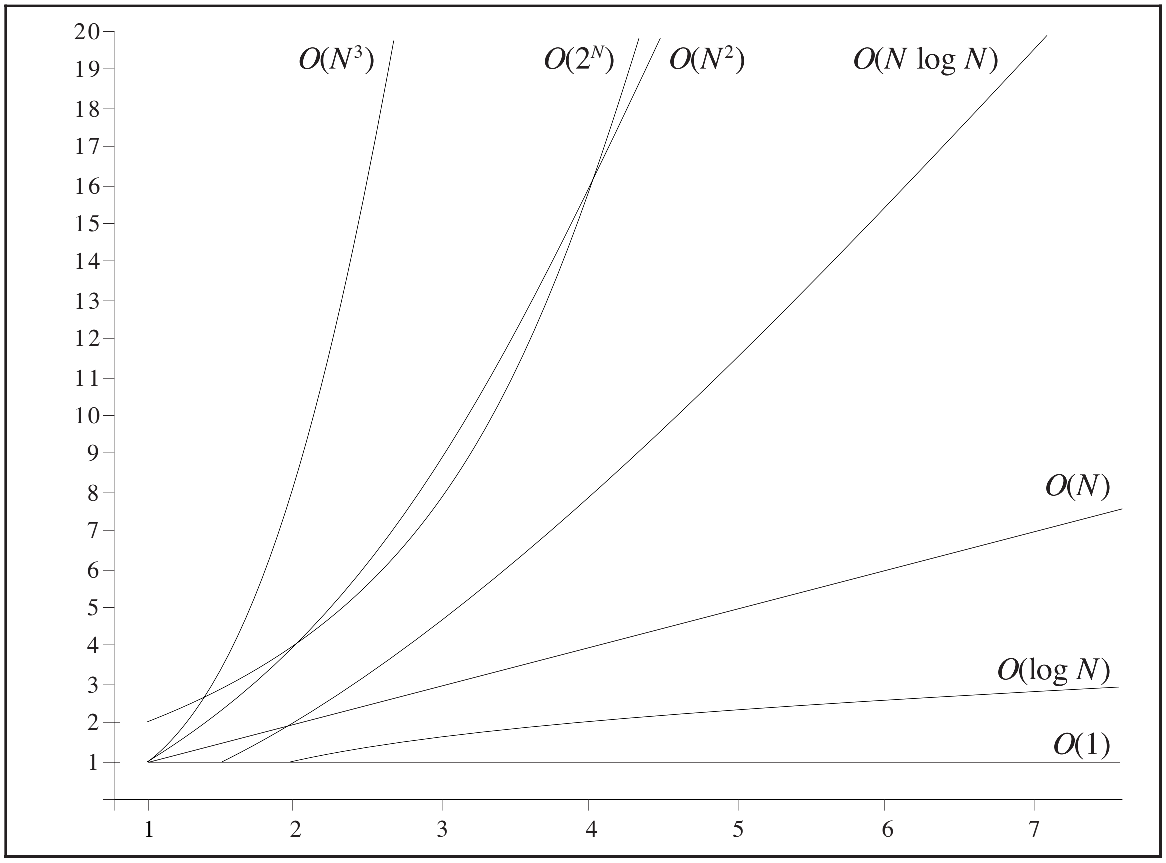
The differences in efficiency between these classes are in fact profound. You can begin to get a sense of how the different complexity functions stand in relation to one another by looking at the graph in Figure 8-4, which plots these complexity functions on a traditional linear scale. Unfortunately, this graph tells an incomplete and somewhat misleading part of the story, because the values of N are all very small. Complexity analysis, after all, is primarily relevant as the values of N become large. Figure 8-5 shows the same data plotted on a logarithmic scale, which gives you a better sense of how these functions grow.

Algorithms that fall into the constant, linear, quadratic, and cubic complexity classes are all part of a more general family called **polynomial algorithms** that execute in time N^k for some constant k. One of the useful properties of the logarithmic plot shown in Figure 8-5 is that the graph of any function N^k always comes out as a straight line whose slope is proportional to k. From looking at the figure, it is immediately clear that the

Table 8-3 Standard complexity classes

constant	$O(1)$	Returning the first element in an array or vector
logarithmic	$O(\log N)$	Binary search in a sorted vector
linear	$O(N)$	Linear search in a vector
$N \log N$	$O(N \log N)$	Merge sort
quadratic	$O(N^2)$	Selection sort
cubic	$O(N^3)$	Conventional algorithms for matrix multiplication
exponential	$O(2^N)$	Tower of Hanoi

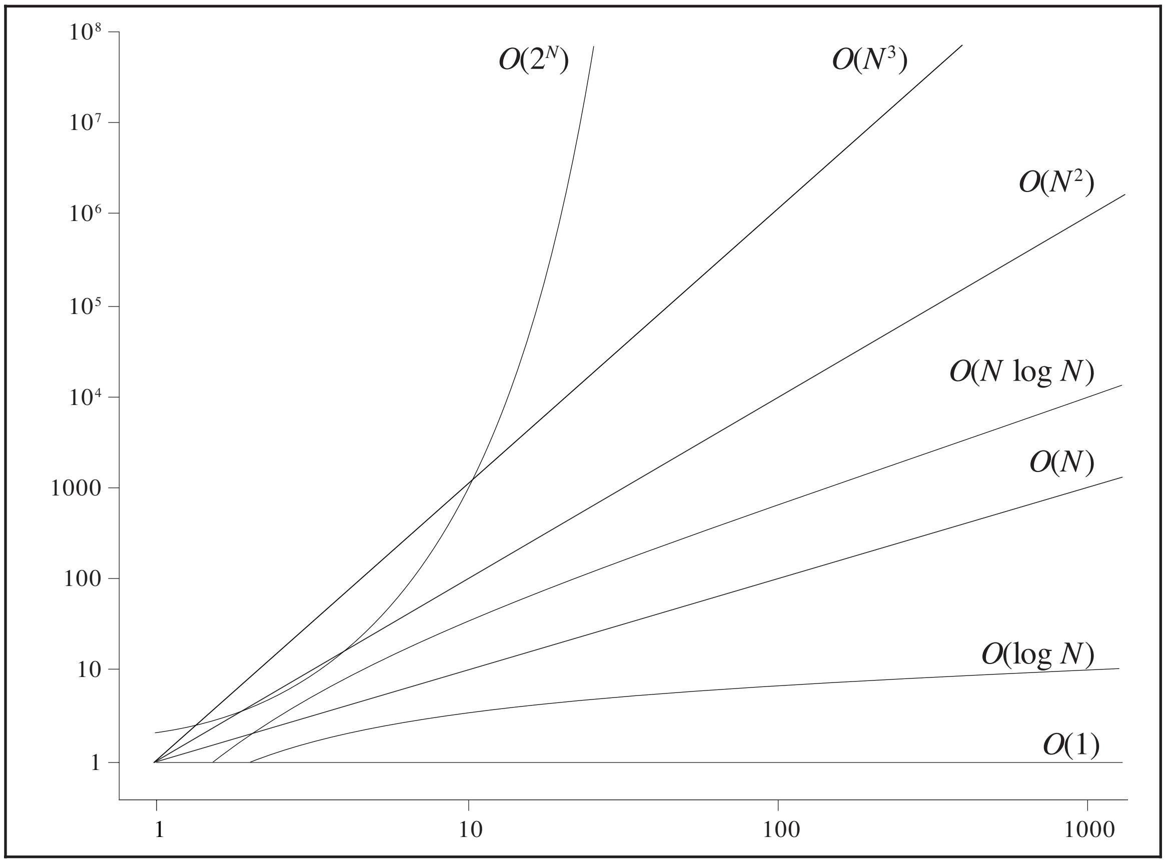
Figure 8-4 Growth characteristics of the standard complexity classes: linear plot



function N^k—no matter how big k happens to be—will invariably grow more slowly than the exponential function represented by 2^N, which continues to curve upward as the value of N increases. This property has important implications in terms of finding practical algorithms for real-world problems. Even though the selection sort example makes it clear that quadratic algorithms have substantial performance problems for large values of N, algorithms whose complexity is $O(2^N)$ are considerably worse. As a general rule of thumb, computer scientists classify problems that can be solved using algorithms that run in polynomial time as **tractable,** in the sense that they are amenable to implementation on a computer. Problems for which no polynomial time algorithm exists are regarded as **intractable.**

Unfortunately, there are many commercially important problems for which all known algorithms require exponential time. For example, suppose you are a member of the sales force for a business and need to find a way to start at your home office, visit a list of cities, and return home, in a way that minimizes the cost of travel. This problem has become a classic in computer science and is called the **traveling salesman problem.** As far as anyone knows, it is not possible to solve the traveling salesman problem in polynomial time. The best-known approaches all have exponential performance in the worst case and are equivalent in efficiency to generating all possible routings and comparing the cost. In general, the number of possible routes in a connected network of N cities grows in proportion to 2^N, which gives rise to the exponential behavior. On the other hand, no one has been able to prove conclusively that no polynomial-time algorithm for this problem exists. There might be some clever algorithm that makes this problem tractable. There are many open questions as to what types of problems can actually be solved in polynomial time, which makes this topic an exciting area of active research.

Figure 8-5 Growth characteristics of the standard complexity classes: logarithmic plot



8.5 The Quicksort algorithm

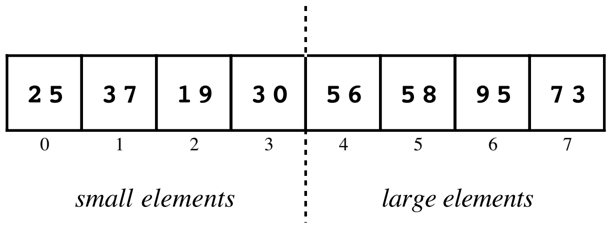
Even though the merge sort algorithm presented earlier in this chapter performs well in theory and has a worst-case complexity of $O(N \log N)$, it is not used much in practice. Instead, most sorting programs in use today are based on an algorithm called Quicksort, developed by the British computer scientist C. A. R. (Tony) Hoare.

Both Quicksort and merge sort employ a divide-and-conquer strategy. In the merge sort algorithm, the original vector is divided into two halves, each of which is sorted independently. The resulting sorted vectors are then merged together to complete the sort operation for the entire vector. Suppose, however, that you took a different approach to dividing up the vector. What would happen if you started the process by making an initial pass through the vector, changing the positions of the elements so that "small" values came at the beginning of the vector and "large" values came at the end, for some appropriate definition of the words *large* and *small?*

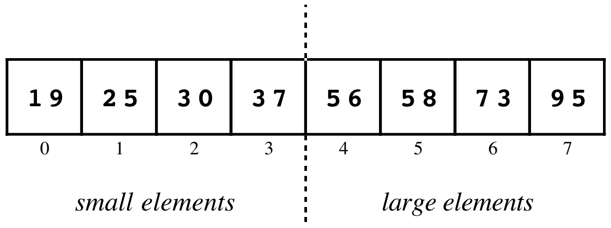
For example, suppose that the original vector you wanted to sort was the following one, presented earlier in the discussion of merge sort:

56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

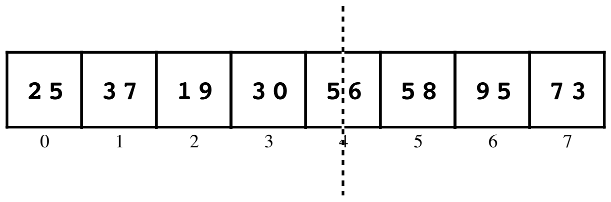
Half of these elements are larger than 50 and half are smaller, so it might make sense to define *small* in this case as being less than 50 and *large* as being 50 or more. If you could then find a way to rearrange the elements so that all the small elements came at the beginning and all the large ones at the end, you would wind up with an vector that looks something like the following diagram, which shows one of many possible orderings that fit the definition:



When the elements are divided into parts in this fashion, all that remains to be done is to sort each of the parts, using a recursive call to the function that does the sorting. Since all the elements on the left side of the boundary line are smaller than all those on the right, the final result will be a completely sorted vector:



If you could always choose the optimal boundary between the small and large elements on each cycle, this algorithm would divide the vector in half each time and end up demonstrating the same qualitative characteristics as merge sort. In practice, the Quicksort algorithm selects some existing element in the vector and uses that value as the dividing line between the small and large elements. For example, a common approach is to pick the first element, which was 56 in the original vector, and use it as the demarcation point between small and large elements. When the vector is reordered, the boundary is therefore at a particular index position rather than between two positions, as follows:



From this point, the recursive calls must sort the vector between positions 0 and 3 and the vector between positions 5 and 7, leaving index position 4 right where it is.

As in merge sort, the simple case of the Quicksort algorithm is a vector of size 0 or 1, which must already be sorted. The recursive part of the Quicksort algorithm consists of the following steps:

1. *Choose an element to serve as the boundary between the small and large elements.* This element is traditionally called the **pivot.** For the moment, it is sufficient to choose any element for this purpose, and the simplest strategy is to select the first element in the vector.
2. *Rearrange the elements in the vector so that large elements are moved toward the end of the vector and small elements toward the beginning.* More formally, the goal of

this step is to divide the elements around a boundary position so that all elements to the left of the boundary are less than the pivot and all elements to the right are greater than or possibly equal to the pivot.[2] This processing is called **partitioning** the vector and is discussed in detail in the next section.

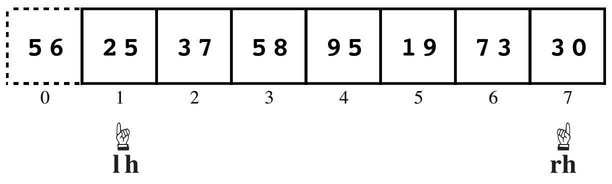
3. *Sort the elements in each of the partial vectors.* Because all elements to the left of the pivot boundary are strictly less than all those to the right, sorting each of the vectors must leave the entire vector in sorted order. Moreover, since the algorithm uses a divide-and-conquer strategy, these smaller vectors can be sorted using a recursive application of Quicksort.

Partitioning the vector

In the partition step of the Quicksort algorithm, the goal is to rearrange the elements so that they are divided into three classes: those that are smaller than the pivot; the pivot element itself, which is situated at the boundary position; and those elements that are at least as large as the pivot. The tricky part about partition is to rearrange the elements without using any extra storage, which is typically done by swapping pairs of elements.

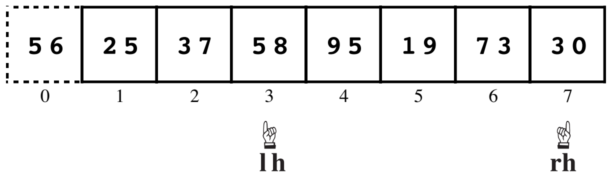
Tony Hoare's original approach to partitioning is fairly easy to explain in English. Because the pivot value has already been selected when you start the partitioning phase of the algorithm, you can tell immediately whether a value is large or small relative to that pivot. To make things easier, let's assume that the pivot value is stored in the initial element position. Hoare's partitioning algorithm proceeds as follows:

1. For the moment, ignore the pivot element at index position 0 and concentrate on the remaining elements. Use two index values, **`lh`** and **`rh`**, to record the index positions of the first and last elements in the rest of the vector, as shown:



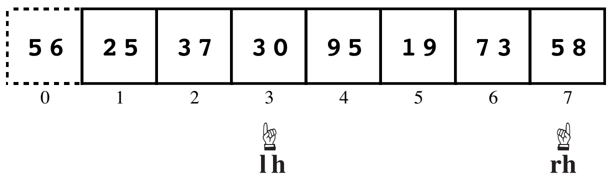
2. Move the **`rh`** index to the left until it either coincides with **`lh`** or points to an element containing a value that is small with respect to the pivot. In this example, the value 30 in position 7 is already a small value, so the **`rh`** index does not need to move.

3. Move the **`lh`** index to the right until it coincides with **`rh`** or points to an element containing a value that is larger than or equal to the pivot. In this example, the **`lh`** index must move to the right until it points to an element larger than 56, which leads to the following configuration:

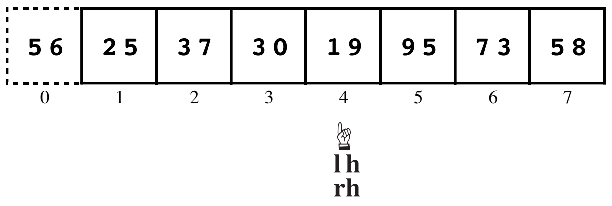


4. If the **`lh`** and **`rh`** index values have not yet reached the same position, exchange the elements in those positions in the vector, which leaves it looking like this:

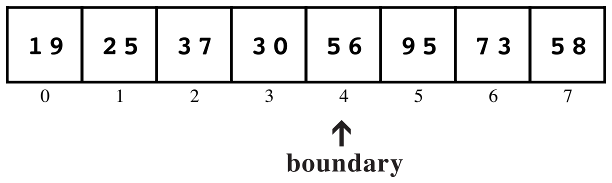
[2] The subarray to the right of the boundary will contain values equal to the pivot only if the pivot value appears more than once in the array.



5. Repeat steps 2 through 4 until the **lh** and **rh** positions coincide. On the next pass, for example, the exchange operation in step 4 swaps the 19 and the 95. As soon as that happens, the next execution of step 2 moves the **rh** index to the left, where it ends up matching the **lh**, as follows:



6. In most cases, the element at the point where the **lh** and **rh** index positions coincide will be the small value that is furthest to the right in the vector. If that position in fact contains a large value, the pivot must have been the smallest element in the entire vector, so the boundary is at position 0. If that position contains a small value, as it usually does, the only remaining step is to exchange that value in this position with the pivot element at the beginning of the vector, as shown:



   Note that this configuration meets the requirements of the partitioning step. The pivot value is at the marked boundary position, with every element to the left being smaller and every element to the right being at least that large.

A simple implementation of **Sort** using the Quicksort algorithm is shown in Figure 8-6.

Figure 8-6 Implementation of Hoare's Quicksort algorithm

```
/*
 * Function: Sort
 * --------------
 * This function sorts the elements of the vector into
 * increasing numerical order using the Quicksort algorithm.
 * In this implementation, Sort is a wrapper function that
 * calls Quicksort to do all the work.
 */

void Sort(Vector<int> & vec) {
   Quicksort(vec, 0, vec.size() - 1);
}
```

```
/*
 * Function: Quicksort
 * -------------------
 * Sorts the elements in the vector between index positions
 * start and finish, inclusive.  The Quicksort algorithm begins
 * by "partitioning" the vector so that all elements smaller
 * than a designated pivot element appear to the left of a
 * boundary and all equal or larger values appear to the right.
 * Sorting the subsidiary vectors to the left and right of the
 * boundary ensures that the entire vector is sorted.
 */

void Quicksort(Vector<int> & vec, int start, int finish) {
    if (start >= finish) return;
    int boundary = Partition(vec, start, finish);
    Quicksort(vec, start, boundary - 1);
    Quicksort(vec, boundary + 1, finish);
}

/*
 * Function: Partition
 * -------------------
 * This function rearranges the elements of the vector so that the
 * small elements are grouped at the left end of the vector and the
 * large elements are grouped at the right end.  The distinction
 * between small and large is made by comparing each element to the
 * pivot value, which is initially taken from vec[start].  When the
 * partitioning is done, the function returns a boundary index such
 * that vec[i] < pivot for all i < boundary, vec[i] == pivot
 * for i == boundary, and vec[i] >= pivot for all i > boundary.
 */

int Partition(Vector<int> & vec, int start, int finish) {
    int pivot = vec[start];
    int lh = start + 1;
    int rh = finish;
    while (true) {
        while (lh < rh && vec[rh] >= pivot) rh--;
        while (lh < rh && vec[lh] < pivot) lh++;
        if (lh == rh) break;
        int temp = vec[lh];
        vec[lh] = vec[rh];
        vec[rh] = temp;
    }
    if (vec[lh] >= pivot) return start;
    vec[start] = vec[lh];
    vec[lh] = pivot;
    return lh;
}
```

Analyzing the performance of Quicksort

As you can see from Table 8-4, this implementation of Quicksort tends to run several times faster than the implementation of merge sort given in Figure 8-2, which is one of the reasons why programmers use it more frequently in practice. Moreover, the running times for both algorithms appear to grow in roughly the same way.

Table 8-4 Empirical comparison of merge sort and Quicksort

N	**Merge sort**	**Quicksort**
10	0.54 msec	0.10 msec
20	1.17 msec	0.26 msec
40	2.54 msec	0.52 msec
100	6.90 msec	1.76 msec
200	14.84 msec	4.04 msec
400	31.25 msec	8.85 msec
1000	84.38 msec	26.04 msec
2000	179.17 msec	56.25 msec
4000	383.33 msec	129.17 msec
10,000	997.67 msec	341.67 msec

The empirical results presented in Table 8-4, however, obscure an important point. As long as the Quicksort algorithm chooses a pivot that is close to the median value in the vector, the partition step will divide the vector into roughly equal parts. If the algorithm chooses its pivot value poorly, one of the two partial vectors may be much larger than the other, which defeats the purpose of the divide-and-conquer strategy. In a vector with randomly chosen elements, Quicksort tends to perform well, with an average-case complexity of $O(N \log N)$. In the worst case—which paradoxically consists of a vector that is already sorted—the performance degenerates to $O(N^2)$. Despite this inferior behavior in the worst case, Quicksort is so much faster in practice than most other algorithms that it has become the standard choice for general sorting procedures.

There are several strategies you can use to increase the likelihood that the pivot is in fact close to the median value in the vector. One simple approach is to have the Quicksort implementation choose the pivot element at random. Although it is still possible that the random process will choose a poor pivot value, it is unlikely that it would make the same mistake repeatedly at each level of the recursive decomposition. Moreover, there is no distribution of the original vector that is always bad. Given any input, choosing the pivot randomly ensures that the average-case performance for that vector would be $O(N \log N)$. Another possibility, which is explored in more detail in exercise 7, is to select a few values, typically three or five, from the vector and choose the median of those values as the pivot.

You do have to be somewhat careful as you try to improve the algorithm in this way. Picking a good pivot improves performance, but also costs some time. If the algorithm spends more time choosing the pivot than it gets back from making a good choice, you will end up slowing down the implementation rather than speeding it up.

8.6 Mathematical induction

Earlier in the chapter, I asked you to rely on the fact that the sum

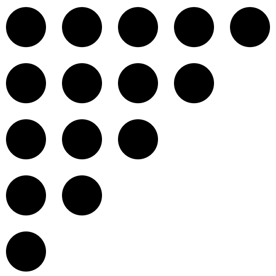
$$N + N{-}1 + N{-}2 + \,.\,.\,.\, + 3 + 2 + 1$$

could be simplified to the more manageable formula

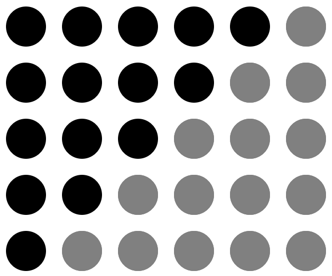
$$\frac{N\,(N+1)}{2}$$

If you were skeptical about this simplification, how would you go about proving that the simplified formula is indeed correct?

There are, in fact, several different proof techniques you could try. One possibility is to represent the original extended sum in a geometric form. Suppose, for example, that N is 5. If you then represent each term in the summation with a row of dots, those dots form the following triangle:



If you make a copy of this triangle and flip it upside down, the two triangles fit together to form a rectangle, shown here with the lower triangle in gray:



Since the pattern is now rectangular, the total number of dots—both black and gray—is easy to compute. In this picture, there are five rows of six dots each, so the total collection of dots, counting both colors, is 5 x 6, or 30. Since the two triangles are identical, exactly half of these dots are black; thus the number of black dots is 30 / 2, or 15. In the more general case, there are N rows containing N+1 dots each, and the number of black dots from the original triangle is therefore

$$\frac{N\,(N+1)}{2}$$

Proving that a formula is correct in this fashion, however, has some potential drawbacks. For one thing, geometrical arguments presented in this style are not as formal as many computer scientists would like. More to the point, constructing this type of argument requires that you come up with the right geometrical insight, which is different for each problem. It would be better to adopt a more general proof strategy that would apply to many different problems.

The technique that computer scientists generally use to prove propositions like

$$N + N{-}1 + N{-}2 + \ . \ . \ . + 3 + 2 + 1 \ = \ \frac{N\,(N+1)}{2}$$

is called **mathematical induction.** Mathematical induction applies when you want to show that a proposition is true for all values of an integer N beginning at some initial starting point. This starting point is called the **basis** of the induction and is typically 0 or 1. The process consists of the following steps:

- *Prove the base case.* The first step is to establish that the proposition holds true when N has the basis value. In most cases, this step is a simple matter of plugging the basis value into a formula and showing that the desired relationship holds.
- *Prove the inductive case.* The second step is to demonstrate that, if you assume the proposition to be true for N, it must also be true for N+1.

As an example, here is how you can use mathematical induction to prove the proposition that

$$N + N-1 + N-2 + \ldots + 3 + 2 + 1 = \frac{N(N+1)}{2}$$

is indeed true for all N greater than or equal to 1. The first step is to prove the base case, when N is equal to 1. That part is easy. All you have to do is substitute 1 for N in both halves of the formula to determine that

$$1 = \frac{1 \times (1+1)}{2} = \frac{2}{2} = 1$$

To prove the inductive case, you begin by assuming that the proposition

$$N + N-1 + N-2 + \ldots + 3 + 2 + 1 = \frac{N(N+1)}{2}$$

is indeed true for N. This assumption is called the **inductive hypothesis.** Your goal is now to verify that the same relationship holds for N+1. In other words, what you need to do to establish the truth of the current formula is to show that

$$N+1 + N + N-1 + N-2 + \ldots + 3 + 2 + 1 = \frac{(N+1)(N+2)}{2}$$

If you look at the left side of the equation, you should notice that the sequence of terms beginning with N is exactly the same as the left side of your inductive hypothesis. Since you have assumed that the inductive hypothesis is true, you can substitute the equivalent closed-form expression, so that the left side of the proposition you're trying to prove looks like this:

$$N+1 + \frac{N(N+1)}{2}$$

From here on, the rest of the proof is simple algebra:

$$N+1 + \frac{N(N+1)}{2}$$

$$= \frac{2N+2}{2} + \frac{N^2+N}{2}$$

$$= \frac{N^2+3N+2}{2}$$

$$= \frac{(N+1)(N+2)}{2}$$

The last line in this derivation is precisely the result you were looking for and therefore completes the proof.

Many students need time to get used to the idea of mathematical induction. At first glance, the inductive hypothesis seems to be "cheating" in some sense; after all, you get to assume precisely the proposition that you are trying to prove. In fact, the process of mathematical induction is nothing more than an infinite family of proofs, each of which proceeds by the same logic. The base case in a typical example establishes that the proposition is true for $N = 1$. Once you have proved the base case, you can adopt the following chain of reasoning:

> Now that I know the proposition is true for $N = 1$, I can prove it is true for $N = 2$.
> Now that I know the proposition is true for $N = 2$, I can prove it is true for $N = 3$.
> Now that I know the proposition is true for $N = 3$, I can prove it is true for $N = 4$.
> Now that I know the proposition is true for $N = 4$, I can prove it is true for $N = 5$.
> And so on. . . .

At each step in this process, you could write out a complete proof by applying the logic you used to establish the inductive case. The power of mathematical induction comes from the fact that you don't actually need to write out the details of each step individually.

In a way, the process of mathematical induction is like the process of recursion viewed from the opposite direction. If you try to explain a typical recursive decomposition in detail, the process usually sounds something like this:

> To calculate this function for $N = 5$, I need to know its value for $N = 4$.
> To calculate this function for $N = 4$, I need to know its value for $N = 3$.
> To calculate this function for $N = 3$, I need to know its value for $N = 2$.
> To calculate this function for $N = 2$, I need to know its value for $N = 1$.
> The value $N = 1$ represents a simple case, so I can return the result immediately.

Both induction and recursion require you to make a leap of faith. When you write a recursive function, this leap consists of believing that all simpler instances of the function call will work without your paying any attention to the details. Making the inductive hypothesis requires much the same mental discipline. In both cases, you have to restrict your thinking to one level of the solution and not get sidetracked trying to follow the details all the way to the end.

Summary

The most valuable concept to take with you from this chapter is that algorithms for solving a problem can vary widely in their performance characteristics. Choosing an algorithm that has better computational properties can often reduce the time required to solve a problem by many orders of magnitude. The difference in behavior is illustrated dramatically by the tables presented in this chapter that give the actual running times for various sorting algorithms. When sorting an vector of 10,000 integers, for example, the Quicksort algorithm outperforms selection sort by a factor of almost 250; as the vector sizes get larger, the difference in efficiency between these algorithms will become even more pronounced.

Other important points in this chapter include:

- Most algorithmic problems can be characterized by an integer N that represents the size of the problem. For algorithms that operate on large integers, the size of the integer provides an effective measure of problem size; for algorithms that operate on arrays or vectors, it usually makes sense to define the problem size as the number of elements.

- The most useful qualitative measure of efficiency is *computational complexity,* which is defined as the relationship between problem size and algorithmic performance as the problem size becomes large.
- *Big-O notation* provides an intuitive way of expressing computational complexity because it allows you to highlight the most important aspects of the complexity relationship in the simplest possible form.
- When you use big-O notation, you can simplify the formula by eliminating any term in the formula that becomes insignificant as N becomes large, along with any constant factors.
- You can often predict the computational complexity of a program by looking at the nesting structure of the loops it contains.
- Two useful measures of complexity are *worst-case* and *average-case* analysis. Average-case analysis is usually much more difficult to conduct.
- Divide-and-conquer strategies make it possible to reduce the complexity of sorting algorithms from $O(N^2)$ to $O(N \log N)$, which is a significant reduction.
- Most algorithms fall into one of several common *complexity classes,* which include the *constant, logarithmic, linear, N log N, quadratic, cubic,* and *exponential* classes. Algorithms whose complexity class appears earlier in this list are more efficient than those that come afterward when the problems being considered are sufficiently large.
- Problems that can be solved in *polynomial time,* which is defined to be $O(N^k)$ for some constant value k, are considered to be *tractable*. Problems for which no polynomial-time algorithm exists are considered *intractable* because solving such problems requires prohibitive amounts of time even for problems of relatively modest size.
- Because it tends to perform extremely well in practice, most sorting programs are based on the *Quicksort algorithm,* developed by Tony Hoare, even though its worst-case complexity is $O(N^2)$.
- Mathematical induction provides a general technique for proving that a property holds for all values of N greater than or equal to some *base* value. To apply this technique, your first step is to demonstrate that the property holds in the base case. In the second step, you must prove that, if the formula holds for a specific value N, then it must also hold for $N+1$.

Review questions

1. The simplest recursive implementation of the Fibonacci function is considerably less efficient than the iterative version. Does this fact allow you to make any general conclusions about the relative efficiency of recursive and iterative solutions?

2. What is the sorting problem?

3. The implementation of **`Sort`** shown in Figure 8-1 runs through the code to exchange the values at positions **`lh`** and **`rh`** even if these values happen to be the same. If you change the program so that it checks to make sure **`lh`** and **`rh`** are different before making the exchange, it is likely to run more slowly than the original algorithm. Why might this be so?

4. Suppose that you are using the selection sort algorithm to sort a vector of 250 values and find that it takes 50 milliseconds to complete the operation. What would you

expect the running time to be if you used the same algorithm to sort a vector of 1000 values on the same machine?

5. What is the closed-form expression that computes the sum of the series

$$N + N{-}1 + N{-}2 + \;.\;.\;. + 3 + 2 + 1$$

6. In your own words, define the concept of computational complexity.

7. True or false: Big-O notation was invented as a means to express computational complexity.

8. What are the two rules presented in this chapter for simplifying big-O notation?

9. Is it technically correct to say that selection sort runs in

$$O\left(\frac{N^2 + N}{2}\right)$$

time? What, if anything, is wrong with doing so?

10. Is it technically correct to say that selection sort runs in $O(N^3)$ time? Again, what, if anything, is wrong with doing so?

11. Why is it customary to omit the base of the logarithm in big-O expressions such as $O(N \log N)$?

12. What is the computational complexity of the following function:

```
int Mystery1(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            sum += i * j;
        }
    }
    return sum;
}
```

13. What is the computational complexity of this function:

```
int Mystery2(int n) {
    int sum = 0;
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < i; j++) {
            sum += j * n;
        }
    }
    return sum;
}
```

14. Explain the difference between worst-case and average-case complexity. In general, which of these measures is harder to compute?

15. State the formal definition of big-O.

16. In your own words, explain why the **Merge** function runs in linear time.

17. The last two lines of the **Merge** function are

```
while (p1 < n1) vec.add(v1[p1++]);
while (p2 < n2) vec.add(v2[p2++]);
```

Would it matter if these two lines were reversed? Why or why not?

18. What are the seven complexity classes identified in this chapter as the most common classes encountered in practice?

19. What does the term *polynomial algorithm* mean?

20. What criterion do computer scientists use to differentiate tractable and intractable problems?

21. In the Quicksort algorithm, what conditions must be true at the conclusion of the partitioning step?

22. What are the worst- and average-case complexities for Quicksort?

23. Describe the two steps involved in a proof by mathematical induction.

24. In your own words, describe the relationship between recursion and mathematical induction.

Programming exercises

1. It is easy to write a recursive function

```
double RaiseToPower(double x, int n)
```

that calculates x^n, by relying on the recursive insight that

$$x^n = x \times x^{n-1}$$

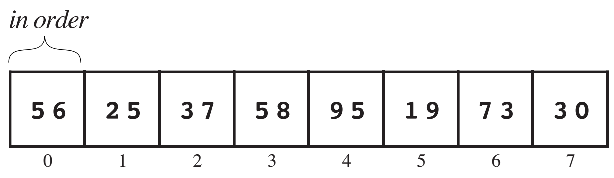
Such a strategy leads to an implementation that runs in linear time. You can, however, adopt a recursive divide-and-conquer strategy which takes advantage of the fact that

$$x^{2n} = x^n \times x^n$$

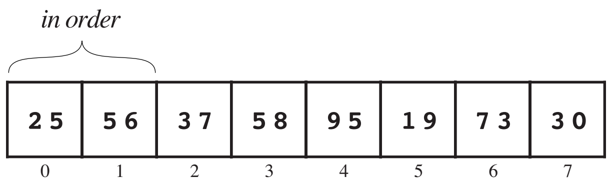
Use this fact to write a recursive version of **RaiseToPower** that runs in $O(\log N)$ time.

2. There are several other sorting algorithms that exhibit the $O(N^2)$ behavior of selection sort. Of these, one of the most important is **insertion sort,** which operates as follows. You go through each element in the vector in turn, as with the selection sort algorithm. At each step in the process, however, the goal is not to find the smallest remaining value and switch it into its correct position, but rather to ensure that the values considered so far are correctly ordered with respect to each other. Although those values may shift as more elements are processed, they form an ordered sequence in and of themselves.

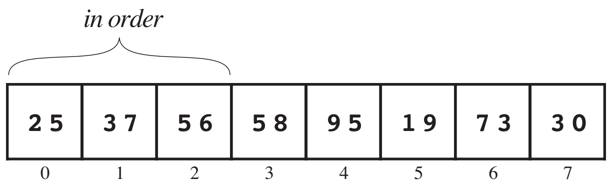
   For example, if you consider again the data used in the sorting examples from this chapter, the first cycle of the insertion sort algorithm requires no work, because an vector of one element is always sorted:



On the next cycle, you need to put 25 into the correct position with respect to the elements you have already seen, which means that you need to exchange the 56 and 25 to reach the following configuration:



On the third cycle, you need to find where the value 37 should go. To do so, you need to move backward through the earlier elements—which you know are in order with respect to each other—looking for the position where 37 belongs. As you go, you need to shift each of the larger elements one position to the right, which eventually makes room for the value you're trying to insert. In this case, the 56 gets shifted by one position, and the 37 winds up in position 1. Thus, the configuration after the third cycle looks like this:



After each cycle, the initial portion of the vector is always sorted, which implies that cycling through all the positions in this way will sort the entire vector.

The insertion sort algorithm is important in practice because it runs in linear time if the vector is already more or less in the correct order. It therefore makes sense to use insertion sort to restore order to a large vector in which only a few elements are out of sequence.

Write an implementation of **`Sort`** that uses the insertion sort algorithm. Construct an informal argument to show that the worst-case behavior of insertion sort is $O(N^2)$.

3. Suppose you know that all the values in an integer array fall into the range 0 to 9999. Show that it is possible to write a $O(N)$ algorithm to sort arrays with this restriction. Implement your algorithm and evaluate its performance by taking empirical measurements using the strategy outlined in exercise 3. Explain why the performance of the algorithm is so bad for small values of N.

4. Write a function that keeps track of the elapsed time as it executes the **`Sort`** procedure on a randomly chosen vector. Use that function to write a program that produces a table of the observed running times for a predefined set of sizes, as shown in the following sample run:

```
     N | Time (msec)
-------+------------
    10 |       0.54
    20 |       1.17
    40 |       2.54
   100 |       6.90
   200 |      14.84
   400 |      31.25
  1000 |      84.38
  2000 |     179.17
  4000 |     383.33
 10000 |     997.67
```

The best way to measure elapsed system time for programs of this sort is to use the ANSI **clock** function, which is exported by the **ctime** interface. The **clock** function takes no arguments and returns the amount of time the processing unit of the computer has used in the execution of the program. The unit of measurement and even the type used to store the result of **clock** differ depending on the type of machine, but you can always convert the system-dependent clock units into seconds by using the following expression:

```
double(clock()) / CLOCKS_PER_SEC
```

If you record the starting and finishing times in the variables **start** and **finish**, you can use the following code to compute the time required by a calculation:

```
double start, finish, elapsed;

start = double(clock()) / CLOCKS_PER_SEC;
. . . Perform some calculation . . .
finish = double(clock()) / CLOCKS_PER_SEC;
elapsed = finish - start;
```

Unfortunately, calculating the time requirements for a program that runs quickly requires some subtlety because there is no guarantee that the system clock unit is precise enough to measure the elapsed time. For example, if you used this strategy to time the process of sorting 10 integers, the odds are good that the time value of **elapsed** at the end of the code fragment would be 0. The reason is that the processing unit on most machines can execute many instructions in the space of a single clock tick—almost certainly enough to get the entire sorting process done for a vector of 10 elements. Because the system's internal clock may not tick in the interim, the values recorded for **start** and **finish** are likely to be the same.

The best way to get around this problem is to repeat the calculation many times between the two calls to the **clock** function. For example, if you want to determine how long it takes to sort 10 numbers, you can perform the sort-10-numbers experiment 1000 times in a row and then divide the total elapsed time by 1000. This strategy gives you a timing measurement that is much more accurate.

5. The implementations of the various sort algorithms in this chapter are written to sort a **Vector<int>**, because the **Vector** type is safer and more convenient than arrays. Existing software libraries, however, are more likely to define these functions so that they work with arrays, which means that the prototype for the **Sort** function is

```
void Sort(int array[], int n)
```

where **n** is the effective size of the array.

Revise the implementation of the three sorting algorithms presented in this chapter so that the use arrays rather than the collection classes from Chapter 4.

6. Write a program that generates a table which compares the performance of two algorithms—linear and binary search—when used to find a randomly chosen integer key in a sorted **Vector<int>**. The linear search algorithm simply goes through each element of the vector in turn until it finds the desired one or determines that the key does not appear. The binary search algorithm, which is implemented for string arrays in Figure 5-5, uses a divide-and-conquer strategy by checking the middle element of the vector and then deciding which half of the remaining elements to search.

   The table you generate in this problem, rather than computing the time as in exercise 3, should instead calculate the number of comparisons made against elements of the vector. To ensure that the results are not completely random, your program should average the results over several independent trials. A sample run of the program might look like this:

```
      N |  Linear   |  Binary
-------+----------+----------
     10 |      5.7  |      2.8
     20 |      8.4  |      3.7
     40 |     18.0  |      4.5
    100 |     49.3  |      5.3
    200 |     93.6  |      6.6
    400 |    193.2  |      7.9
   1000 |    455.7  |      8.9
   2000 |    924.1  |     10.0
   4000 |   2364.2  |     11.2
  10000 |   5078.1  |     12.4
```

7. Change the implementation of the Quicksort algorithm so that, instead of picking the first element in the vector as the pivot, the **Partition** function chooses the median of the first, middle, and last elements.

8. Although $O(N \log N)$ sorting algorithms are clearly more efficient than $O(N^2)$ algorithms for large vectors, the simplicity of quadratic algorithms like selection sort often means that they perform better for small values of N. This fact raises the possibility of developing a strategy that combines the two algorithms, using Quicksort for large vectors but selection sort whenever the vectors become less than some threshold called the **crossover point.** Approaches that combine two different algorithms to exploit the best features of each are called **hybrid strategies.**

   Reimplement **Sort** using a hybrid of the Quicksort and selection sort strategies. Experiment with different values of the crossover point below which the implementation chooses to use selection sort, and determine what value gives the best performance. The value of the crossover point depends on the specific timing characteristics of your computer and will change from system to system.

9. Another interesting hybrid strategy for the sorting problem is to start with a recursive Quicksort that simply returns when the size of the vector falls below a certain threshold. When this function returns, the vector is not sorted, but all the elements are relatively close to their final positions. At this point, you can use the insertion

sort algorithm presented in exercise 2 on the entire vector to fix any remaining problems. Because insertion sort runs in linear time on vectors that are mostly sorted, you may be able to save some time using this approach.

10. Suppose you have two functions, f and g, for which $f(N)$ is less than $g(N)$ for all values of N. Use the formal definition of big-O to prove that

$$15f(N) + 6g(N)$$

is $O(g(N))$.

11. Use the formal definition of big-O to prove that N^2 is $O(2^N)$.

12. Use mathematical induction to prove that the following properties hold for all positive values of N.

a) $1 + 3 + 5 + 7 + \ldots + 2N\text{–}1 = N^2$

b) $1^2 + 2^2 + 3^2 + 4^2 + \ldots + N^2 = \dfrac{N(N+1)(2N+1)}{6}$

c) $1^3 + 2^3 + 3^3 + 4^3 + \ldots + N^3 = (1 + 2 + 3 + 4 + \ldots + N)^2$

d) $2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^N = 2^{N+1} - 1$

13. Exercise 1 shows that it is possible to compute x^n in $O(\log N)$ time. This fact in turn makes it possible to write an implementation of the function `Fib(n)` that also runs in $O(\log N)$ time, which is much faster than the traditional iterative version. To do so, you need to rely on the somewhat surprising fact that the Fibonacci function is closely related to a value called the **golden ratio,** which has been known since the days of Greek mathematics. The golden ratio, which is usually designated by the Greek letter ϕ, is defined to be the value that satisfies the equation

$$\phi^2 - \phi - 1 = 0$$

Because this is a quadratic equation, it actually has two roots. If you apply the quadratic formula, you will discover that these roots are

$$\phi = \frac{1 + \sqrt{5}}{2}$$

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2}$$

In 1718, the French mathematician Abraham de Moivre discovered that the nth Fibonacci number can be represented in closed form as

$$\frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$$

Moreover, since $\hat{\phi}^n$ is always very small, the formula can be simplified to

$$\frac{\phi^n}{\sqrt{5}}$$

rounded to the nearest integer.

Use this formula and the **`RaiseToPower`** function from exercise 1 to write an implementation of **`Fib(n)`** that runs in $O(\log N)$ time. Once you have verified empirically that the formula seems to work for the first several terms in the sequence, use mathematical induction to prove that the formula

$$\frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$$

actually computes the n^{th} Fibonacci number.

14. If you're ready for a real algorithmic challenge, write the function

```
int MajorityElement(Vector<int> & vec);
```

that takes a vector of nonnegative integers and returns the **majority element**, which is defined to be a value that occurs in an absolute majority (at least 50 percent plus one) of the element positions. If no majority element exists, the function should return –1 to signal that fact. Your function must also meet the following conditions:

- It must run in $O(N)$ time.
- It must use $O(1)$ additional space. In other words, it may use individual temporary variables but may not allocate any additional array or vector storage. Moreover, this condition rules out recursive solutions, because the space required to store the stack frames would grow with the depth of the recursion.
- It may not change any of the values in the vector.

The hard part about this problem is coming up with the algorithm, not implementing it. Play with some sample vectors and see if you can come up with the right strategy.