CHAPTER 9

Introduction to Time Complexities

Sometimes, a particular data structure is a part of a well-known algorithm, just as the stack was in the expression evaluation algorithm. In this case, we may choose between the array implementation and the linked-list implementation, but we pretty much know that we are going to have to use a stack. At other times, we are developing an algorithm and we have a choice among two or more different data structures to use for the algorithm—and it is not always the case that one data structure is better than another. Someone who has a lot of experience with data structures eventually realizes that one data structure is better than another in some situations, but not in others. Each data structure seems to have its own situation in which it is more advantageous among the others. Data structures are like the tools in a toolbox: When you need a hammer, you might have a choice between a rubber mallet and a sledgehammer. Both might work for a given task, but one would probably work better; each has a situation in which one works better than the other.

One critical factor in the decision about which data structure to use lies in the speed of the operations (or functions) of each data structure. Other factors can be important, but speed is almost always important. Therefore, to aid in making an informed decision about which data structure to use, we need to have an intelligent way to compare the speeds of the functions involved.

There is more to this comparison than meets the eye, so let's understand what we are dealing with. Consider an array of elements and an algorithm (within the function of a data structure) that does something with the array. The amount of time the function takes to execute may vary with the number of elements in the array. We might expect that when we have more elements in the array, the algorithm will take longer to execute; there are more elements to process, and processing each element takes some

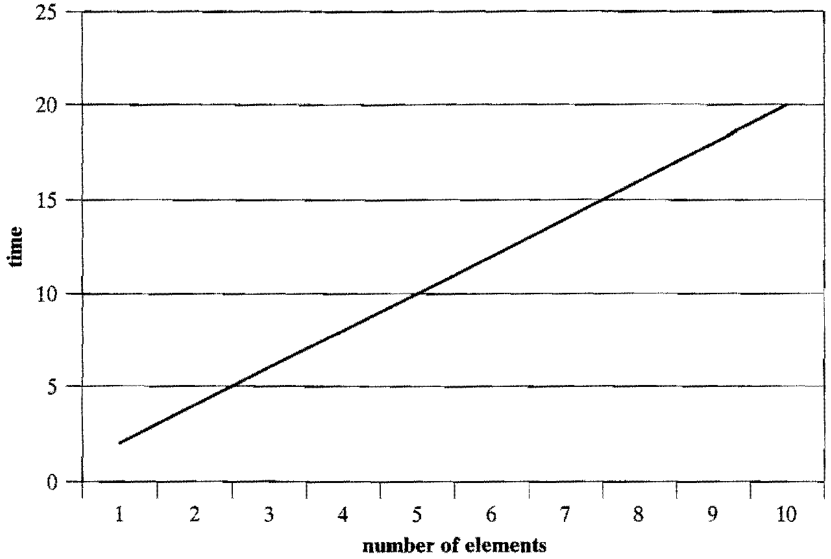

FIGURE 9.1

A line graph of number of elements vs. time.

amount of time (if we are indeed processing each element). Our expectation about the number of elements in the array and the time required for execution might look something like the graph in Figure 9.1.

In the figure, each point on the line gives the number of elements (on the x-axis) and the amount of time the algorithm takes to execute (on the y-axis). Thus, if we take a point higher on the line, the number of elements is greater, and so is the time for execution. Conversely, a point lower on the line gives us a lesser number of elements, but a smaller amount of time for execution.

An algorithm, however, does not always produce the nice, straight line that we see in Figure 9.1: Sometimes the graph is more like that in Figure 9.2 or Figure 9.3. In both of these figures, as the number of elements increases, the time the function takes to execute also increases. However, the *behavior* is different: In Figure 9.2, the time increases dramatically as the number of elements increases; in Figure 9.3, the time increases as the number of elements increases, but it seems about to level off, although it never quite does. If we had to choose which figure has the better behavior, it would be Figure 9.3. In that figure, if we increase the number of elements in the array, we are not going to pay a heavy price in the amount of time the algorithm takes to execute. The ***time complexity*** of an algorithm describes its behavior in such a manner. The different behaviors that we see in Figures 9.1, 9.2, and 9.3 are really different time complexities.

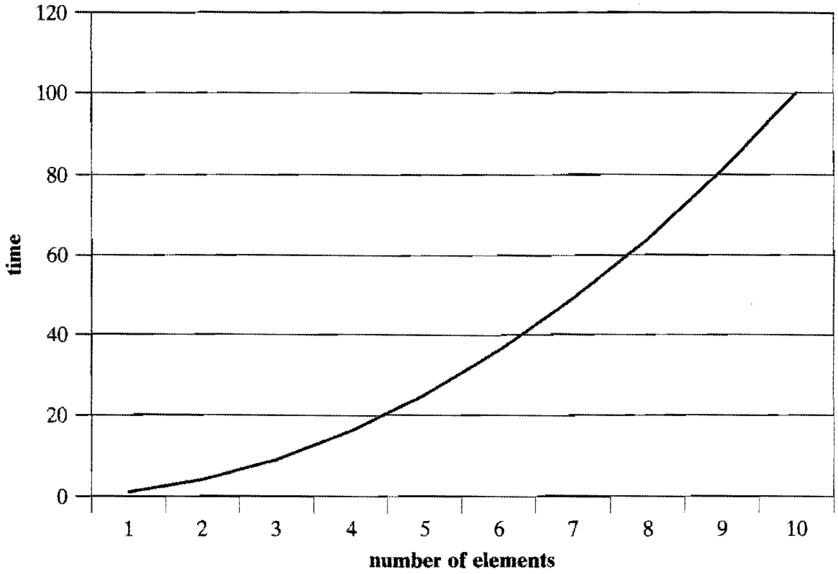

FIGURE 9.2

Another plot of number of elements vs. time. The time scale is higher and increases substantially as the number of elements increases.

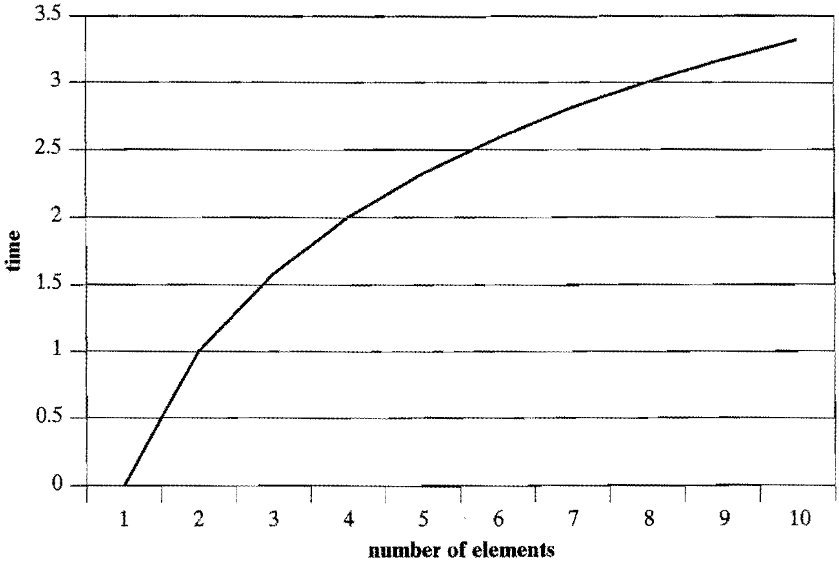

FIGURE 9.3

Another way of plotting number of elements vs. time. The time scale is very small, and an increase in the number of elements results only in a slight increase in time.

9.1 TIME COMPLEXITY BASICS

Suppose we have two different data structures to choose from, one with function A and one with function B. Both functions accomplish the same thing with an array, but they take very different approaches. (There may be only one mountaintop, but there are many ways up the mountain.) Let's suppose that Alan needs to make a decision about which data structure to use. (We need *some* scapegoat.) Now, Alan doesn't know much about computer science, but after carefully weighing all the factors, he realizes that the data structure with the faster function, A or B, should be chosen for the program. He can't really tell which is faster by looking at them, because their approaches to solving a particular problem are radically different. So Alan decides to time them. He uses an array size of 100 and runs each one a million times (automated in a program test, of course). Let's say that function B turned out to be faster. Then its data structure is chosen.

Little does Alan know that if we were to examine the behaviors of functions A and B, they would compare as shown in Figure 9.4. At 100 elements (the array size that Alan used for the test), function B is clearly faster than function A, as shown by the dots. So Alan was right. However, when the program is finally given to the customer, the customer always uses it with a lot of data, filling the array to 500 elements or more; notice that function A would be much faster in this situation, so the customer was shortchanged. The example shows that time complexities would help us to make a more informed decision about which data structure to use.

Whenever we compare two algorithms that have different time complexities, their graphs often intersect as shown in Figure 9.4. The point of intersection is called

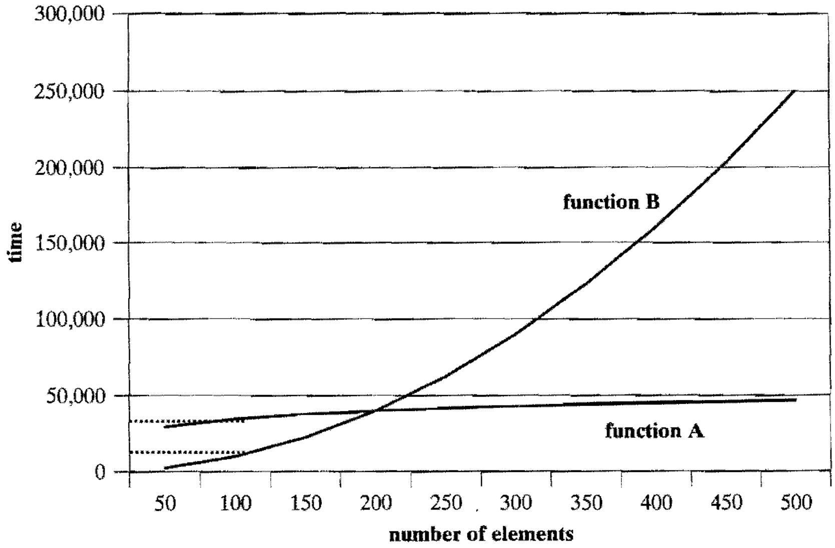

FIGURE 9.4

A comparison of the behaviors of functions A and B for number of elements vs. time.

the ***cross-over*** point, the number of elements at which both algorithms take the same amount of time to execute. In this case, to the left of the cross-over point function B is faster than function A, but to the right of the cross-over point the situation is reversed and function A is faster. So how do we determine which function is better? Computer scientists often ignore what happens to the left of the cross-over point. The reason is that the number of elements is relatively low here, and *any* algorithm that executes with a low number of elements is probably going to be fairly fast. In other words, the user of the algorithm won't really notice the difference in speed. It is when the number of elements is high that computer programs start to take a longer time to execute. Here, the user really *can* notice the difference between the speeds of the algorithms. Therefore, we often base decisions about data structures on what happens to the right of the cross-over point, where the number of elements can be high.

Often, we are not really sure about the number of elements at which a cross-over point occurs between two algorithms with different time complexities. The cross-over point rarely occurs at a high number of elements, but when it does, we should pay attention to the left of the cross-over point, because all practical uses of the program might occur there. In the vast majority of cases, the cross-over point occurs at a relatively small number of elements. Although we frequently are not sure about the precise location of the cross-over point, we know which time complexity is going to eventually be better with an increasing number of elements. As the number of elements increases, the number is really "approaching infinity", so we use this kind of language very often. Sometimes people think of "approaching infinity" as being extremely high, but you can increase the number of elements from 5 to 6 and be approaching infinity. However, "approaching infinity" is eventually going to pass to the right of the cross-over point. We say that we are considering the ***asymptotic running times*** of algorithms when we think about their running times as the number of elements approaches infinity (which is just a fancy way of saying that we are considering numbers to the right of any possible cross-over point).

The main purpose of a time complexity is to compare the amount of time algorithms take to execute when that amount of time can vary. If we never had to make a decision about which algorithm to use or which data structure to use, then time complexities would probably not be discussed in any textbook.

At this point, we would like to examine how we can determine the time complexity of an algorithm. But there is quite a bit to this undertaking, so we will have to lead up to it. First of all, in discussing time complexities, we often use the variable n to represent the number of elements. It is just easier to say n than to keep saying "the number of elements." The variable n can represent the number of elements in an array, but it might also represent the number of elements in something else, such as a linked list. In reality, n is used for anything that has a pronounced effect on the execution time of an algorithm as that thing is varied; it just turns out that, in the vast majority of cases, this happens to be the number of elements that we are working with. We often refer to n as being the *problem size* for the algorithm.

In determining time complexities, the time that an algorithm takes to execute is represented by the number of instructions that are executed in the algorithm. The rationale for adopting this method of representing time is that the more instructions algorithms have, the longer they take to execute. It is not a perfect method, because, in reality, each instruction can take a different amount of time to execute. Therefore, we cannot really think of the execution time of one instruction as being one time unit (although it is often perceived that way during analysis). This method, however, works surprisingly well in determining time complexities.

Let's look at an example and count the number of instructions executed:

```
1  sum = 0;
2  i = 0;
3  while ( i < 3 ) {
4      sum += A[ i ];
5      i++;
6  }
```

Lines 1–2 constitute 2 instructions. We'll consider the checking of the condition in the while loop as another instruction, giving us 3 instructions up to this point. Then, we have an iteration of instructions. We repeat line 4, line 5, and the condition check on each iteration, so 3 instructions are involved on each iteration. The loop repeats three times, giving us 9 instructions for all iterations. Adding this on to the other 3 instructions at the beginning gives us a total of 12 instructions. (We are assuming that there are no overloaded operators; otherwise there would be more instructions to count.)

The interesting thing about the number of instructions is that it can often be represented as a function of n. That is, for a value of n put into the function, the function produces a certain number of instructions. To see an example of this, let's take the same code as before, but use a variable numElements instead of 3:

```
sum = 0;
i = 0;
while ( i < numElements ) {
        sum += A[ i ];
        i++;
}
```

We still have the first 3 instructions, counting the initial condition check in the while loop. But this time, the number of iterations will be numElements instead of 3. So the code is adding all the elements in the array A. Realizing that numElements is really n, we know that the number of instructions in all the iterations is $3n$. Therefore, the function for this algorithm would be

$$f(n) = 3n + 3$$

The function $f(n)$ gives us the number of instructions executed in the algorithm. For example, if n is 5, the total number of instructions executed is $f(n) = 3 \times 5 + 3 = 18$. If n is 2, the total number of instructions executed is $f(n) = 3 \times 2 + 3 = 11$. Hence, this is a nice parallel to the number of elements and execution time that we discussed earlier. But the fact that we can represent the number of instructions executed as a function of n is significant, as we will see later. By the way, the function for the first example of this chapter would be

$$f(n) = 12$$

assuming that the algorithm is correctly written (i.e., that the array will always have at least three elements).

Let's take a look at a slightly more complex algorithm:

```
1   sum = 0;
2   i = 0;
3   while ( i < n ) {
4       j = 0;
5       while ( j < n ) {
6               sum += i * j;
7               j++;
8               }
9       i++;
10      }
```

Let's examine lines 4–9 first. We start off with 2 instructions on lines 4–5. Then we iterate n times with 3 instructions in each iteration, giving us $3n + 2$ instructions so far. When line 9 is executed, we have a total of $3n + 3$ instructions for lines 4–9. But all of these instructions are repeated n times in the while loop of line 3, giving us a total of $n(3n + 3)$ instructions. The while loop condition on line 3 is also involved on each iteration, so we have $n(3n + 3) + n$ instructions. Finally, the first 3 instructions, counting the initial condition check, are added in, giving us $n(3n + 3) + n + 3$, which reduces to $3n^2 + 4n + 3$, so our function for this algorithm is

$$f(n) = 3n^2 + 4n + 3$$

As you can probably imagine, for more complex algorithms we could have a lot of fun finding a function for the number of instructions executed. You will see shortly that we don't really have to go through all of this tedious work to determine a time complexity, but it was necessary to show it to you so that you get a feel for what is going on.

If we graph these functions out, we can see their behavior. As you probably know from algebra, the function $f(n) = 3n + 3$ will have the shape of a straight line, as in Figure 9-1. (The figure shows the *shape*, not what the actual line would look like.) The function $f(n) = 3n^2 + 4n + 3$ gives us half of a parabola, like the shape in Figure 9.2. (It won't be a full parabola, because n, the number of elements, can't be negative.) But these functions aren't really time complexities. A time complexity is actually a *set* of functions. So these functions are really *elements* of time complexities.

Now, in order to understand how a time complexity is actually derived, you need to recall from algebra the meanings of the words ***term*** and ***coefficient***. The terms of a function are the parts of it that are added together, as shown in Figure 9.5a. The coefficients of a function are the leading constants of each term, shown in blue in Figure 9.5b.

If the function has at least one term with an n, then, in order to derive the time complexity, we remove the least significant terms from the function, leaving only the most significant term—the term that would be the largest number if n were a very high

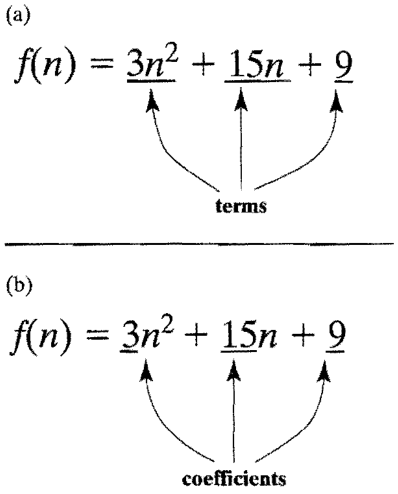

FIGURE 9.5

(a) The terms of a function. (b) The coefficients of a function.

value. Then we remove the coefficient from the most significant term. For the function $f(n) = 3n + 3$, the most significant term is $3n$, and removing its leading coefficient would give us a time complexity of n. For the function $f(n) = 3n^2 + 4n + 3$, the most significant term is $3n^2$, and removing its most significant term would give us a time complexity of n^2. There is a special notation that we use for time complexities. A time complexity of n is written as $\Theta(n)$, and a time complexity of n^2 is really written as $\Theta(n^2)$, pronounced theta-*n* and theta-*n* squared, respectively. This is called ***theta notation***. It is used so that when we see n or n^2, we know that we are talking about, not the number of instructions in an algorithm, but rather, the time complexity.

A time complexity should be thought of as a set. An infinite number of functions would belong to the set $\Theta(n)$. For example, $3n + 4$, $10n + 1, 5n, 2n + 100$, $n, \frac{1}{2}n + 25$, and $n - 2$ are all examples of functions that belong to the $\Theta(n)$ time complexity. The last two examples might surprise you a little. However, the coefficients can be less than 1, as in $\frac{1}{2}n + 25$. In this case, we might go only halfway through an array before stopping a loop. It is also possible to have negative signs in the function, as in $n - 2$. A negative sign in a function would occur, for example, in this algorithm:

```
sum = 0;
i = 0;
while ( i < size – 2 ) {
        sum += sum + A[ i ];
        i++;
}
```

Here, we stop the loop after size −2 iterations. Since size is presumably n, the loop stops after $n - 2$ iterations. There are three instructions on each iteration, so the function for this algorithm would be $3(n - 2) + 3$, which works out to

$$f(n) = 3n - 3$$

If you have a negative sign in the most significant term of a function, however, i is an indication that something went awry. If n is high enough, it would give you a negative number for the number of instructions, which is nonsensical.

When one gains experience in determining time complexities, one realizes that i is not really necessary to count every single instruction in an algorithm. All that is necessary is that you have an understanding of how the time complexity is derived, so you know what you can ignore and what you can't ignore. Let's take a look at the second algorithm again:

```
sum = 0;
i = 0;
while ( i < numElements ) {
        sum += A[ i ];
        i++;
}
```

Once we notice that the while loop executes n times, we have the time complexity. We don't really need to count the number of instructions in the while loop if it is a constant number, because it will just end up being the coefficient of n that we throw out. The first three instructions will be an insignificant term that we throw out.

In the example

```
sum = 0;
i = 0;
while ( i < n ) {
        j = 0;
        while ( j < n ) {
                sum += i * j;
                j++;
                }
        i++;
        }
```

the nested loops, with each loop iterating n times, tells us that the time complexity will be $\Theta(n^2)$. If a loop iterates n times, and all those iterations are repeated n times, it's going to give us an n^2 term. There is nothing here to give us an n^3 term or higher. Once we realize that the n^2 term will be the most significant term, we have the time complexity.

Sometimes, we'll have one loop followed by another loop in an algorithm. (The second loop starts after the first finishes executing.) In sequential cases like this, we would *add* the time complexities of the individual components that are in sequence For example, suppose that the first loop, by itself, has a time complexity of $\Theta(n)$. Once the first loop finishes executing, the second loop starts, and it has a time complexity of say, $\Theta(n^2)$ (possibly, it contains a nested loop). Adding the time complexities of these components together gives us $\Theta(n) + \Theta(n^2)$. Remember that we can add these time complexities because one finishes executing before the other one starts. If one looks at this sum, one realizes that, in the instruction function, the highest term would be the n^2 term. Therefore, the overall time complexity would be $\Theta(n^2)$. We often use this technique to come up with an overall time complexity—the highest time complexity of those components which are in a sequence.

We rarely have a need to count up the exact number of instructions, as a function of n, in an algorithm. On occasion, however, we are interested in where the cross-over point occurs between two algorithms. The cross-over point between two algorithms can often be approximated by the functions that represent the number of instructions. For example, let's consider the functions of two different algorithms:

$$f(n) = n^2 + n + 2$$
$$g(n) = 4n + 2$$

The cross-over point would occur where the number of elements and the number of instructions are exactly the same for both functions. Since the number of instructions would be the same, and each function gives us the number of instructions, we can equate the two functions:

$$n^2 + n + 2 = 4n + 2$$

When we solve for an n that makes this equation true, we have the point at which the number of elements and the number of instructions is exactly the same. Subtracting the right side from both sides of the equation yields

$$n^2 - 3n = 0$$

which factors to

$$n(n - 3) = 0$$

There are two solutions: $n = 0$ and $n = 3$. But $n = 0$ is a nonsensical answer, since the number of elements is 0. Thus, at $n = 3$, we have a cross-over point between the two algorithms that we are interested in. We can check that the number of instructions is the same by using $n = 3$ in both functions. This yields 14 instructions for each function. (Such a calculation should be considered only a ballpark approximation of a cross-over point.) At $n = 2$, we would presume that $f(n)$ is faster than $g(n)$, since $f(n) = 8$ instructions while $g(n) = 10$ instructions. But at $n = 4$, we would presume that $g(n)$ is faster, since $g(n) = 18$ instructions while $f(n) = 22$ instructions. We can see that, for larger values of n, $g(n)$ will continue to be faster.

However, $g(n)$ will do more than continue to be faster. Since $g(n)$ and $f(n)$ belong to different time complexities, the larger n gets, the faster $g(n)$ will get over $f(n)$. This situation, which always occurs when two algorithms have different time complexities, is illustrated in Table 9.1. At $n = 10$, $g(n)$ is 2.7 times faster than $f(n)$. At $n = 100$, $g(n)$ is 25.1 times faster, and at $n = 1000$, $g(n)$ is 250.1 times faster. The greater the number of elements, the more benefit that we get out of an algorithm that has a better time complexity. However, don't read more into this than you see. If one algorithm has more speedup over another as n increases, it does not mean that the algorithms belong to different

TABLE 9.1 Amount of speedup of $g(n)$ over $f(n)$ as n increases.

n	$f(n) = n^2 + n + 2$ (number of instructions)	$g(n) = 4n + 2$ (number of instructions)	How many times faster $g(n)$ is over $f(n) : [f(n)/g(n)]$
10	112	42	2.7
100	10102	402	25.1
1000	1001002	4002	250.1

time complexities (consider $3n^2 + 5n$ and $3n^2$). But if two algorithms have different time complexities, then one will have more speedup over another as n increases.

Let's consider two functions: $f(n) = 3n$ and $g(n) = 9n$. These functions belong to the same time complexity, $\Theta(n)$. As Table 9.2 shows, $f(n)$ is always three times faster than $g(n)$ as n increases. But we don't need a table for that; we can see that it's true just by looking at the functions, right? When the speedup of one function over another remains the same as n increases, the two functions belong to the same time complexity. Again, however, don't read more into this than you see: If the speedup of one function over another increases as n increases, it doesn't mean that the algorithms *don't* belong to the same time complexity. (Consider, once more, $3n^2 + 5n$ and $3n^2$.) However, if two algorithms belong to the same time complexity, then any speedup of one over another as n increases won't be significant.

These principles apply to any two time complexities. In theory, there are an infinite number of time complexities—for example, $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^3)$, $\Theta(n^4)$, etc. However, the only time complexities that we will need to deal with in this textbook are listed in Table 9.3, which also gives the names that are sometimes used for these time complexities. It may be surprise you that the time complexities in Table 9.3 are really quite common. We will be discussing these other time complexities a little later.

For now, we are still building on the notion of what a time complexity is. Let's consider why the function $f(n) = 4n^2 + 6n$ belongs to the $\Theta(n^2)$ time complexity and not some other time complexity. In the light of the discussion we just had, it might seem that it should belong to another time complexity. However, let's consider the function $g(n) = 4n^2$. We can see clearly that $g(n)$ is faster than $f(n)$ for all $n > 0$. That is,

$$4n^2 < 4n^2 + 6n$$

Let us also consider the function $h(n) = 12n^2$. The relationship

$$4n^2 + 6n \le 4n^2 + 6n^2 = 10n^2 < 12n^2$$

TABLE 9.2 The amount of speedup of $f(n)$ over $g(n)$ stays the same as n increases. (This can also be seen just by looking at the functions: $g(n) =3f(n)$, so $g(n)$ always has three times as many instructions as $f(n)$.)

n	$f(n) = 3n$ (number of instructions)	$g(n) = 9n$ (number of instructions)	How many times faster $f(n)$ is over $g(n) : [g(n)/f(n)]$
10	30	90	3
100	300	900	3
1000	3000	9000	3

TABLE 9.3 Time complexities used in this book and their common names.

Time complexity	Common name
$\Theta(1)$	constant
$\Theta(\lg n)$	logarithmic
$\Theta(n)$	linear
$\Theta(n\ lg\ n)$	n-log-n
$\Theta(n^2)$	quadratic

makes it clear that $f(n)$ is faster than $h(n)$ for all $n > 0$. Thus, $4n^2 + 6n < 12n^2$ for all $n > 0$. So now we have a situation in which $g(n) = 4n^2$ is always faster than $f(n) = 4n^2 + 6n$, but $f(n)$ is always faster then $h(n) = 12n^2$. That is, $f(n)$ is always sandwiched in between the two as n approaches infinity. But if you look closely at the "bread slices," $g(n)$ and $h(n)$, you will see that they definitely belong to the same time complexity and, further, that $g(n)$ will always be three times faster than $h(n)$. We can see that in the function $f(n) = 4n^2 + 6n$, the $6n$ term is really not significant at all; we say that it is ***absorbed*** into the time complexity of the most significant term.

It can further be shown that *any* term which is not the most significant term of a function can be absorbed into the time complexity of the most significant term, although some of the proofs are quite involved. But that is the reason we always remove the terms of lesser significance from the function in deriving the time complexity. It can also be shown that a function which belongs to one time complexity cannot be sandwiched in between the functions of another time complexity. You will see such proofs at the beginning of a course or textbook on algorithm analysis.

Once we have removed all terms of lesser significance, the coefficient of the remaining most significant term is also not a part of the time complexity. Recall that the speedup between this term and a term of equal significance will be the same, regardless of the coefficient, as was illustrated in Table 9.2. When the amount of speedup does not change as n increases, the functions belong to the same time complexity. So the coefficient is not a part of the time complexity either.

If we are comparing two algorithms and we find that they have two different time complexities, we can tell quickly which will be faster to the right of the cross-over point (if there is one). It is obvious, for example, that $\Theta(n)$ is a better time complexity than $\Theta(n^2)$. After all, when we look at n, it is really giving us an indication about the number of instructions that need to be executed. The shapes of the graphs of these two time complexities also tell you which will be better. In the $\Theta(n^2)$ algorithm, we pay a heavy price in execution time as the number of elements is increased, as shown in Figure 9.2.

Sometimes a time complexity of an algorithm is called the running time of the algorithm, but this is really a misnomer. We can tell nothing about how long the algorithm will actually take to run simply by looking at its time complexity. If an algorithm has a $\Theta(n)$ time complexity, we often say that the algorithm runs in $\Theta(n)$ time, language that seems to be a bit more palatable.

There is a long, formal definition of time complexity that relates closely to what we have been talking about, but there is no need to examine it. As long as you have a basic idea of what a time complexity is, and you are familiar with the time complexities that we need to use, that is all that is required for this textbook. So, let's summarize the notion of a time complexity:

1. Time complexities describe the pattern of how an algorithm's execution time changes as some other factor or factors, usually the number of elements, change.
2. Time complexities are useful in comparing algorithms, because the execution time of an algorithm will sometimes differ depending upon some other factor or factors; thus, time complexities are used as an aid in making an informed decision about which algorithm or data structure to use.
3. When two algorithms have different time complexities, one will have an increasingly significant speedup over another as n increases. If two algorithms have the same time complexity, there will be no significant difference in the speedup of one over another as n increases.

In determining which algorithm will be better as far as speed goes, we always look at their time complexities first. If their time complexities are the same, then we sometimes time them to see which one is better.

9.2 THE CONSTANT TIME COMPLEXITY

A $\Theta(1)$ (pronounced theta-1) time complexity is the best possible time complexity. In this time complexity, an increase in n does not affect how long the algorithm will take to execute. An example of a $\Theta(1)$ time complexity is the time complexity of the algorithm that dequeues an element from a queue implemented with a linked list. It does not matter if the queue has a hundred elements, a thousand elements, or even a billion elements: The dequeue operation works just with the front of the linked list and takes the same amount of time.

Sometimes a $\Theta(1)$ time complexity is called *constant time*. This, however, is a misnomer. Time is hardly ever constant for such algorithms, and many factors including the architecture of the computer, can affect the time one way or another.

The only thing that is true about a $\Theta(1)$ time complexity is that it has an upper limit to the number of instructions, regardless of how high n is increased. Take, for example, the following code:

```
if ( a > b ) {
        x = A[ 0 ] + 2;
        z = x * 3;
        }
x++;
y = x + 1;
```

This is code that has a $\Theta(1)$ time complexity. It does not matter how many elements are in array A, because we access only the first element. However, the number of instructions can be different, depending on whether a $>$ b. If a $>$ b, then five instructions are executed, counting the condition. If a $<=$ b, then only three instructions are executed.

However, the number of instructions in this code can be no more than five. Therefore, five is an upper limit on the number of instructions. When code has a constant upper limit on the number of instructions, the code has a $\Theta(1)$ time complexity. You will sometimes hear the expression that an algorithm is ***bounded*** by a constant. This means the same thing as having a constant upper limit in instructions.

It is considered improper to use notations such as $\Theta(5)$ for a constant time complexity. No matter how high the constant is that bounds the algorithm, the correct notation is always $\Theta(1)$.

9.3 BIG-OH NOTATION

Sometimes, not only the number of elements used in an algorithm, but even the time complexity itself, can vary with the situation. For example, let's look at the templated code presented next, which searches for an element in a ***sorted*** array. Recall that a sorted array is an array in which the elements are placed in order. If the elements are objects, then they are ordered according to some data member within the objects.

```
i = 0;
found = false;
while ( ( i < size ) && itemToFind > A[ i ] )
    i++;

if ( itemToFind == A[ i ] )
    found = true;
```

We are assuming templated code here, so that itemToFind and the elements within A[i] can be of any DataType. The operators > and == are overloaded if DataType is an object.

We know that the array A is in sorted order, so as long as itemToFind is greater than A[i], we should keep searching. If itemToFind is no longer greater than A[i], then one of two things has occurred: (1) We have found the item in A, and they are equal, which causes us to exit the while loop, or (2) the item is not in A; then, since A is in order, itemToFind will be less than the rest of the elements in A, and we should stop searching. The last two lines of the code test for the situation that has occurred and set found to true if appropriate. It is also possible that we will search through the whole array and never find itemToFind, because itemToFind is greater than every element in A. In this case, i will eventually reach the last element. When i becomes equal to size, we also exit the while loop. Short-circuiting is used here: If the condition i < size is false, the condition itemToFind < A[i] won't be checked; if short-circuiting were not used, we could get a runtime error.

In the best case, the algorithm finds itemToFind right away, in the first element of the array. So the best case is bounded by a constant number of instructions, six in this case (counting the two conditions in the while loop heading as two instructions). Therefore, the best case runs in $\Theta(1)$ time.

In the worst case, we would go through all of the elements of the array without finding itemToFind. Since we would have n iterations, the worst case runs in $\Theta(\mathrm{n})$ time. So, in this example, the time complexity can vary with the situation.

A type of time complexity notation that we can use for this situation is $O(n)$, which is pronounced big-oh-of-n. When we use big-oh notation, we are saying that the time complexity might not be exactly what we have written, but it will be no worse. So $O(n)$ means that the time complexity won't be any worse than $\Theta(n)$. The $O(n)$ set of algorithms would include all of those algorithms whose time complexity is either $\Theta(n)$ or better than $\Theta(n)$. Thus, if we have a $\Theta(1)$ algorithm that executes only, say, six instructions, regardless of the size of n, then this function belongs to $\mathrm{O}(n)$ as well as to $\Theta(1)$. Such an algorithm would also belong to $\mathrm{O}(n^2)$.

The $O(n)$ time complexity applies to the previous search algorithm, since its time complexity, depending on the situation, might not be $\Theta(n)$, but can be no worse than $\Theta(n)$. We would also be correct in using the $O(n^2)$ notation for this search algorithm, but it would be misleading. When using big-oh notation, we should always use the best time complexity that applies correctly to the situation. We should always use theta notation when the time complexity cannot vary and we know exactly what it is. Sometimes it is considered to be an abuse of big-oh notation when we use it while knowing the best-case, average-case, and worst-case time complexities for an algorithm, all in theta notation. The reason is that if the best- and average-case time complexities are much better than the worst-case time complexity, big-oh notation can be misleading.

There are three other time complexity notations, but there is no need to cover them here. You will see them in a course or textbook on algorithm analysis.

9.4 THE LOGARITHMIC TIME COMPLEXITY

The last time complexity we will discuss is the logarithmic time complexity, $\Theta(\lg n)$.

Logarithms are really quite simple if you look at them in the right way. A logarithmic equation is just another way of rewriting an equation that has an exponent. Figure 9.6 shows two equivalent equations and how they translate. The equations mean exactly the same thing; what you see are just two different ways of writing it. And it is really not too hard to remember how these equations translate. You just have to remember two things:

1. The base of the logarithm in one equation is also the base for the exponent in the other equation. In Figure 9.6, the base is 3 in both equations.
2. The result of applying a logarithm is an exponent.

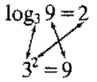

FIGURE 9.6

Conversion from a logarithmic equation to an exponential equation and vice versa. The two equations have exactly the same meaning, but are written in different forms.

These are the only two things you have to remember. If you are translating from an equation with an exponent to an equation with a logarithm, or vice versa, once you know where the base and the exponent go, there is only one other place for the third number to go.

In computer science, the abbreviation lg is often used for a logarithm of base 2. So let's look at an example. What is the result of lg 16? First, think about the equation with the exponent. If you diligently memorized the two points just presented, you know that the result (let's call it x) is an exponent. The base is 2, so we have

$$2^x$$

There is a 16 left and there is only one place for it to go:

$$2^x = 16$$

We know that when x is 4, this equation is true. So 4 is the result of lg 16. Try your hand at translating $\log_3 9$. Once you get the hang of this, you can do it in your head by visualizing the equation.

The logarithmic time complexity $\Theta(\lg n)$ is a highly desirable time complexity. Although it is not as good as $\Theta(1)$, it is the next best thing among the most common time complexities. It is much better than $\Theta(n)$. Its shape is shown in the graph of Figure 9.3.

The nice thing about the logarithmic time complexity is that n can be very large, but the number of instructions executed, represented by $\lg n$, is still rather small. Let's suppose we have an algorithm whose function for the number of instructions is exactly $\lg n$; that is $f(n) = \lg n$. We know that

$$2^{10} = 1024$$

so we may write it as the equivalent logarithm

$$\lg 1024 = 10.$$

Since our function is $f(n) = \lg n$, if 1024 elements were used by this algorithm, only 10 instructions would be executed. It gets better: If n is 1,048,576, then $\lg n$ is only 20 instructions. What if n is the number of atoms in the known universe?

The number of atoms in the known universe is estimated to be 10^{80}. Since 10 is approximately $2^{3.32}$, $10^{80} \approx (2^{3.32})^{80}$. Multiplying the exponents together gives us approximately 2^{266} for the number of atoms in the known universe. So what is the result of $\lg 2^{266}$? Well, if we set it equal to x, we have

$$\lg 2^{266} = x$$

Remembering that lg is another way to write $\log_2$, we know that the base is 2 and x is an exponent, giving us

$$2^x$$

There is only one place for the large number 2^{266} to go:

$$2^x = 2^{266}$$

This means that $x = 266$. (Actually, we should have known that. Why?) But since our function is $f(n) = \lg n$, if n (the number of elements used by our algorithm) is equal to the number of atoms in the known universe, 2^{266}, then the number of instructions executed in the algorithm is only 266. (You've now got to have some respect for the $\Theta(\lg n)$ time complexity!)

Now that we've talked about how tantalizing the $\Theta(\lg n)$ time complexity is, let's discuss how an algorithm can achieve such a time complexity in its worst case. Basically, the idea is to somehow set up a loop that reduces the size of the problem by half on each iteration. That is, on the first iteration of the loop, the size of the problem is n; on the second iteration, the size is $\frac{1}{2}n$; on the third iteration, the size is $\frac{1}{4}n$, and so on, until the size of the problem becomes 1 element. Let's take a look at how many iterations would be required in such a loop. Suppose we have t iterations and we want to solve for t. Since each iteration reduces the size of the problem by half until we reach 1 element, we need to figure out how many times we need to multiply $\frac{1}{2}$ by n in order to reach 1 element. On the first iteration, we have n elements. On the second iteration, we have

$$\tfrac{1}{2}n$$

elements. On the third iteration, we have

$$\tfrac{1}{2} \times \tfrac{1}{2}n$$

elements. We keep multiplying by $\frac{1}{2}$ until the size of the problem is equal to 1:

$$\tfrac{1}{2} \times \tfrac{1}{2} \times \tfrac{1}{2} \times \ldots \times \tfrac{1}{2}n = 1$$

Since each iteration reduces the size of the problem by half, and we have t iterations, we obtain

$$(\tfrac{1}{2})^t n = 1$$

which is the same as

$$\left(\frac{1^t}{2^t}\right)n = 1$$

This equation reduces to

$$\left(\frac{1}{2^t}\right)n = 1$$

Multiplying both sides by 2^t yields

$$n = 2^t$$

When we convert this formula into its equivalent logarithm form, we get

$$\lg n = t$$

Since t is the number of iterations of the loop, we have $\lg n$ iterations of the loop, thereby achieving the $\Theta(\lg n)$ time complexity (assuming that the number of instructions inside the loop is bounded by a constant).

Sometimes, multiplying by $^1/_2$ continuously does not give us exactly 1. But if we need to multiply by $^1/_2$ an extra time to get less than 1 element, it does not affect the time complexity. That is, if we have $(\lg n) + 1$ iterations, we would end up throwing out the least significant term to get the time complexity. So, in the end, the time complexity is still $\Theta(\lg n)$.

9.5 THE BINARY SEARCH ALGORITHM

Let's take a look at an algorithm called the *binary search algorithm*, which runs in $O(\lg n)$ time (i.e., its time complexity can vary, but is no worse than $\Theta(\lg n)$). Let's call the previous search algorithm we examined the linear search algorithm. The linear search algorithm is often slower, running in $O(n)$ time. In both algorithms, we work with a sorted array. (In fact, the binary search algorithm will not work if the array is *not* sorted.)

The operation of the binary search algorithm is shown in Figure 9.7, which illustrates searching for a value of 33 in an array of integers. We first get the middle index of the array by adding the first and last indexes and dividing their sum by 2. If the result

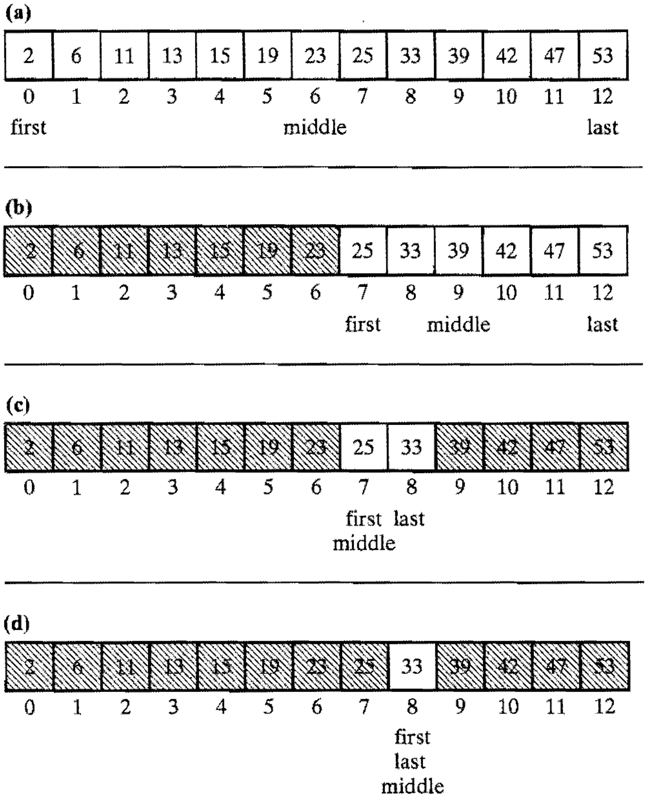

FIGURE 9.7

(a) A sorted array that we want to search for the key value 33, using the binary search algorithm; we start by looking at the middle location. (b) After comparing 33 with the value at the middle location, we need not consider the keys 23 and lower anymore. (Locations are darkened.) We look at the middle location of the remaining section. (c)–(d) The search narrows until the key value 33 is found.

does not work out to an integer, the next lower integer is used for the middle index. We compare the value at that middle index, A [middle], with 33. Since the array is sorted, if A[middle] <33, then we must search in the right half of the array, and if A[middle] >33, then we must search in the left half of the array. This feature is significant, because we are eliminating about half the array from the search. When we restrict ourselves to one-half of the array for the search, the size of the problem essentially reduces to that half. We then take the middle index of that half and compare it with 33. We can then reduce the size of *this* problem by half, using the same technique. Each time we eliminate half of what remains, we are going through an iteration of the binary search algorithm. Eventually, we will get down to 1 element. If it isn't the number 33, then 33 is not in the array.

The templated binary search algorithm is as follows:

```
1   int first = 0;
2   int last = size – 1;
3   int middle;
4   bool found = false;
5
6   while ( first <= last && !found ) {
7       middle = ( first + last ) >> 1;
8       if ( itemToFind == A[ middle ] )
9               found = true;
10      else if ( itemToFind < A[ middle ] )
11              last = middle – 1;
12      else // itemToFind must be greater than A[ middle ]
13              first = middle + 1;
14      }
```

The itemToFind and the elements of array A might be objects. If so, the == and < operators will be overloaded. The first two lines show the index variables used as place markers for the first and last elements of any array section we might be dealing with. Initially, first is set to the first index of the entire array and last is set to the last index of the entire array. We also declare an index for the middle of the array on line 3. We use a found variable, which is set to true if the value is found. When found is set to true, the while loop condition will be false on line 6 and the program will exit the loop.

Line 7 calculates the middle index with the use of the current first and last indexes by taking their average. The average, in this case, gives the midpoint, so the current first and last indexes are added together and divided by 2. (See the discussion of the shift operator in Section 8.2.) The division is integer division, so if the division is not even, an integer result is still given (by truncating the right of the decimal point).

Then, if itemToFind is equal to A[middle] on line 8, we have found the value, so found is set to true on line 9. On line 10, if itemToFind is less than A[middle], we know that we need to search the left half of this array section. The current first index is still the first index in the left half; we just need to reset the last index. The new last index will be the next position to the left of the middle. (We don't need to include the current middle again, since we know it is not the value we seek.) The new last index is set on line 11.

If we execute line 13, we must have had the case that itemToFind was greater than A[middle]. So we know that we need to search the right half of this array section. In this case, the current last index will still be the last index in the right half; we just need to reset the first index. The new first index will be the next position to the right of the middle. The new first index is set on line 13.

If itemToFind still has not been found, then, when we continue into the next iteration of the loop, we will do so with the first and last indexes set appropriately to the section of the array that we need to search. The middle index of that section is then recalculated on line 7, and we compare itemToFind with the value at this index on line 8. Hence, the process repeats itself, narrowing the search down by resetting the first and last indexes as appropriate.

The first and last indexes will be approaching the same value as they get closer and closer together. If itemToFind is not found, eventually the first index will be higher than the last index, causing the condition on line 6 to be false, and the while loop is exited. For example, if the first and last indexes become equal, the middle index will be calculated at that same index. Then, if itemToFind is not equal to A[middle], it is not in the array. Notice, however, that if itemToFind is less than A[middle], the last index is set to middle −1 on line 11, which would make it less than the first index. By contrast, if itemToFind is greater than A[middle], the first index is set to middle +1, which would make it greater than the last index. Either situation will cause an exit from the while loop, since we have the condition "first <= last" in the while loop heading on line 6.

We may not have to go through all lg n iterations of the binary search algorithm to find the element we are looking for. If we are lucky, we will find it on the first attempt, at the first A[middle]. So its best-case time complexity is $\Theta(1)$. This is the reason we say that the binary search algorithm runs in O(lg n) time.

You can probably win a $10 bet with one of your friends. Tell them to think of a number between 1 and 1 billion, and tell them that you can get it within 30 attempts, as long as they tell you whether each guess you make is higher or lower than the actual number. Then, just use the binary search algorithm. Since 2^{30} is greater than a billion, you'll be able to find it within the 30 attempts. (On the first attempt, you eliminate 500 million numbers right off the bat!) Just don't make a mistake! Figure 9.8 shows that starting with a billion elements and eliminating half the elements on each step will get you down to 1 element after 29 attempts. Note that when the number of elements remaining is odd, the binary search algorithm eliminates half the elements +0.5, an integer, if the element is not found at the middle index. When the number of remaining elements is even, the binary search will eliminate either half the elements or half the elements +1 (the worst case is assumed in Figure 9.8). The binary search algorithm is something to keep in mind when you're short on cash. Your friends will learn not to mess with a computer scientist!

Attempt Number	Elements Remaining
1	500,000,000
2	250,000,000
3	125,000,000
4	62,500,000
5	31,250,000
6	15,625,000
7	7,812,500
8	3,906,250
9	1,953,125
10	976,562
11	488,281
12	244,140
13	122,070
14	61,035
15	30,517
16	15,258
17	7,629
18	3,814
19	1,907
20	953
21	476
22	238
23	119
24	59
25	29
26	14
27	7
28	3
29	1
30	Give me my $10.

FIGURE 9.8

A number between 1 and 1 billion can be guessed within 30 attempts with the use of the binary search algorithm.

9.6 COMPUTER SPEED: WHERE DOES IT REALLY COME FROM?

The contents of the while loop of the binary search algorithm are bounded by a constant. We may have to check up to two conditions inside the loop, and the while loop heading has two conditions to check. Add these to the instructions on line 9 and line 11 (or line 13), and we have a bound of six instructions on each iteration of the binary search. In the linear search algorithm, we had a bound of three instructions in the loop. Thus, each iteration of the linear search can be about twice as fast as each iteration of the binary search algorithm. But the most important thing is the number of iterations.

Here is a problem to think about: If we have a million elements in an array, then which algorithm would be faster, a binary search on a 500-MHz computer or a linear search on a faster 5-GHz computer? Assume that each instruction on the 5-GHz computer is 10 times faster than each instruction on the 500-MHz computer and that each iteration of a linear search will be twice as fast as each iteration of a binary search, as we learned in the previous paragraph.

Well, first of all, let's think about the number of iterations we might have. A binary search of 1 million elements is only 20 iterations at the most. (Let's assume this worst case.) Using a linear search on a million elements, we might assume, on average, that we would go halfway through the array before we find the element. That would be 500,000 iterations. If each iteration took the same amount of time, the binary search would be 500,000/20 = 25,000 times faster than the linear search. However, as we've said, an iteration of a linear search is assumed to be about twice as fast as an iteration of a binary search *on the same computer*. So if we were on the same computer, the binary search would be 25,000/2 = 12,500 times faster.

The binary search is working on the slower, 500-MHz computer, while the linear search is working with the faster, 5-GHz computer. Therefore, each instruction of the binary search will be 10 times slower. On these different computers, the binary search will be 12,500/10 = 1,250 times faster. So, when we have a million elements, a binary search on a 500-MHz computer is 1,250 times faster than a linear search on a 5-GHz computer.

This example was brought up for an important reason: to show that algorithmic improvements can make a computer much faster than typical hardware improvements can. As computer scientists, we have a tremendous impact on computer speed whenever we develop algorithms. Through our own improvements in software, we have the capability of making computers run much faster even without any hardware improvements at all. So whenever you study time complexities, don't take them with a grain of salt.

Our ingenuity in improving computer speed doesn't apply just to algorithms like the binary search algorithm; it applies to the data structures that use the algorithms, too. Consider the linked-list implementation of the queue in Chapter 8. With most linked lists, the only thing we are concerned about is having a pointer to the front of the list. However, if we just had a pointer to the front of the linked list in a queue, as is traditional, then whenever we needed to enqueue, we would first have to travel through the entire linked list, node by node, until we found the last node with a pointer to NULL. Then we would enqueue the new element at the end of the linked list. The problem is that the procedure would require a loop to find the last node, and the time complexity would be $\Theta(n)$. Ingeniously, however, the standard implementation of a queue has a back pointer to the last node added into the data structure, so that we would not have to travel node by node through the linked list to find the last node. Thus, the enqueue operation is turned into a $\Theta(1)$ algorithm. This is nothing to scoff at: If a linked list has a thousand nodes, this little idea of having a back pointer might make a run through the algorithm about a thousand times faster.

9.7 TIME COMPLEXITIES OF DATA STRUCTURE FUNCTIONS

If we take a look at most of the stack and queue functions that we have talked about in Chapter 8, we see that most of them turn out to have $\Theta(1)$ algorithms. In the linked-list implementations, the exceptions turn out to be the copy constructors, overloaded assignment operators, destructors, and makeEmpty functions. The copy constructors and overloaded assignment operators must make deep copies of the linked lists; in so doing, they process the list node by node and are thus $\Theta(n)$ algorithms. The destructor and the makeEmpty function must free the entire linked list node by node; therefore, these also run in $\Theta(n)$ time.

In the array implementations, the copy constructors and overloaded assignment operators must copy an array element by element, too. There is no way to do it all at once if we are making a deep copy. This causes them to run in $\Theta(n)$ time. The makeEmpty functions and destructors in the array implementations, however, run in $\Theta(1)$ time, because the dynamic array does not have to be freed element by element.

In the array implementation of a stack, the push and pop algorithms run in $\Theta(1)$ time if the dynamic array does not need to be expanded or contracted. If the dynamic