Chapter 5

Design Fundamentals

Concepts:
- ▷ Asymptotic analysis and big-O notation
- ▷ Time-space trade-off
- ▷ Back-of-the-envelope estimations
- ▷ Recursion and Induction

We shape clay into a pot,
but it is the emptiness inside
that holds whatever we want.
—Lao Tzu

PROGRAMMERS ARE CRAFTSMEN. Their medium—their programming language—often favors no particular design and pushes for an individual and artistic decision. Given the task of implementing a simple program, any two individuals are likely to make different decisions about their work. Because modern languages allow programmers a great deal of expression, implementations of data structures reflect considerable personal choice.

Some aspects of writing programs are, of course, taught and learned. For example, everyone agrees that commenting code is good. Programmers should write small and easily understood procedures. Other aspects of the design of programs, however, are only appreciated after considerable design experience. In fact, computer science as a whole has only recently developed tools for understanding what it means to say that an algorithm is "implemented nicely," or that a data structure "works efficiently." Since many data structures are quite subtle, it is important to develop a rich set of tools for developing and analyzing their performance.

In this chapter, we consider several important conceptual tools. *Big-O* complexity analysis provides a means of classifying the growth of functions and, therefore, the performance of the structures they describe. The concepts of *recursion* and *self-reference* make it possible to concisely code solutions to complex problems, and *mathematical induction* helps us demonstrate the important properties—including trends in performance—of traditional data structures. Finally, notions of symmetry and friction help us understand how to design data structures so that they have a reasonable look and feel.

5.1 Asymptotic Analysis Tools

We might be satisfied with evaluating the performance or *complexity* of data structures by precisely counting the number of statements executed or objects

referenced. Yet, modern architectures may have execution speeds that vary as much as a factor of 10 or more. The accurate counting of any specific kind of operation alone does not give us much information about the actual running time of a specific implementation. Thus, while detailed accounting can be useful for understanding the fine distinctions between similar implementations, it is not generally necessary to make such detailed analyses of behavior. Distinctions between structures and algorithms, however, can often be identified by observing patterns of performance over a wide variety of problems.

5.1.1 Time and Space Complexity

What concerns the designer most are *trends* suggested by the various performance metrics as the problem size increases. Clearly, an algorithm that takes time proportional to the problem size degrades more slowly than algorithms that decay quadratically. Likewise, it is convenient to agree that any algorithm that takes time bounded by a polynomial in the problem size, is better than one that takes exponential time. Each of these rough characterizations—linear, quadratic, and so on—identifies a class of functions with similar growth behavior. To give us a better grasp on these classifications, we use *asymptotic* or *big-O* analysis to help us to describe and evaluate a function's growth.

Definition 5.1 *A function $f(n)$ is $O(g(n))$ (read "order g" or "big-O of g"), if and only if there exist two positive constants, c and n_0, such that*

$$|f(n)| \leq c \cdot g(n)$$

for all $n \geq n_0$.

In this text, f will usually be a function of problem size that describes the utilization of some precious resource (e.g., time or space). This is a subtle definition (and one that is often stated incorrectly), so we carefully consider why each of the parts of the definition is necessary.

Most importantly, we would like to think of $g(n)$ as being proportional to an upper bound for $f(n)$ (see Figure 5.1). After some point, $f(n)$ does not exceed an "appropriately scaled" $g(n)$. The selection of an appropriate c allows us to enlarge $g(n)$ to the extent necessary to develop an upper bound. So, while $g(n)$ may not directly exceed $f(n)$, it might if it is multiplied by a constant larger than 1. If so, we would be happy to say that $f(n)$ has a trend that is no worse than that of $g(n)$. You will note that if $f(n)$ is $O(g(n))$, it is also $O(10 \cdot g(n))$ and $O(5 + g(n))$. Note, also, that c is positive. Generally we will attempt to bound $f(n)$ by positive functions.

Second, we are looking for long-term behavior. Since the most dramatic growth in functions is most evident for large values, we are happy to ignore "glitches" and anomalous behavior up to a certain point. That point is n_0. We do not care how big n_0 must be, as long as it can be nailed down to some fixed value when relating specific functions f and g.

Nails = proofs.

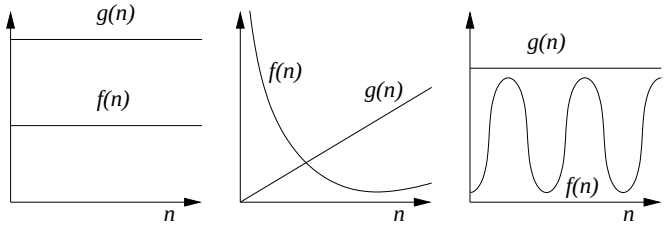


Figure 5.1 Examples of functions, $f(n)$, that are $O(g(n))$.

Third, we are not usually interested in whether the function $f(n)$ is negative or positive; we are just interested in the *magnitude* of its growth. In reality, most of the resources we consider (e.g., time and space) are measured as positive values and larger quantities of the resource are consumed as the problem grows in size; growth is usually positive.

Most functions we encounter fall into one of a few categories. A function that is bounded above by a constant is classified as $O(1)$.[1] The constant factor can be completely accounted for in the value of c in Definition 5.1. These functions measure size-independent characteristics of data structures. For example, the time it takes to assign a value to an arbitrary element of an array of size n is constant.

What's your best guess for the time to assign a value? $\frac{1}{1000}$ second? $\frac{1}{1000000}$ sec.? $\frac{1}{1000000000}$ s.?

When a function grows proportionately to problem size, or *linearly*, we observe it is $O(n)$. Depending on what's being measured, this can be classified as "nice behavior." Summing the values in an n-element array, for example, can be accomplished in linear time. If we double the size of the array, we expect the time of the summation process to grow proportionately. Similarly, the `Vector` takes linear space. Most methods associated with the `Vector` class, if not constant, are linear in time and space. If we develop methods that manipulate the n elements of a `Vector` of numbers in *superlinear* time—faster than linear growth—we're not pleased, as we know it can be accomplished more efficiently.

Other functions grow *polynomially* and are $O(n^c)$, where c is some constant greater than 1. The function $n^2 + n$ is $O(n^2)$ (let $c = 2$ and $n_0 = 1$) and therefore grows as a quadratic. Many simple methods for sorting n elements of an array are quadratic. The space required to store a square matrix of size n takes quadratic space. Usually, we consider functions with polynomial growth to be fairly efficient, though we would like to see c remain small in practice. Because a function n^{c-1} is $O(n \cdot n^{c-1})$ (i.e., $O(n^c)$), we only need consider the growth of the most significant term of a polynomial function. (It is, after all, most significant!) The less significant terms are ultimately outstripped by the leading term.

Grass could be greener.

[1] It is also $O(13)$, but we try to avoid such distractions.

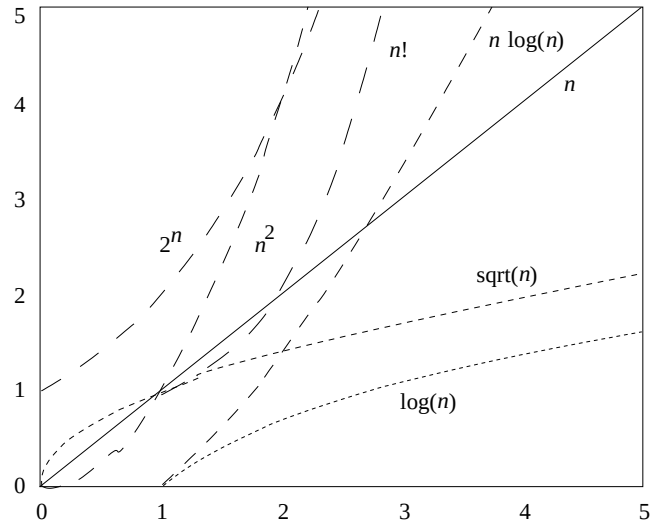


Figure 5.2 Near-origin details of common curves. Compare with Figure 5.3.

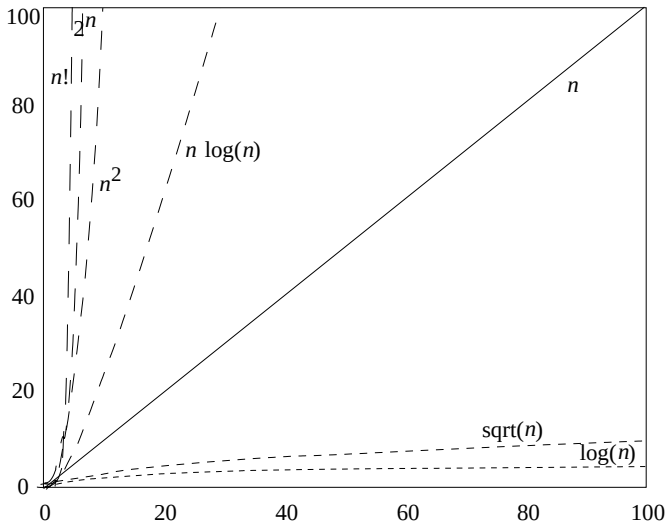


Figure 5.3 Long-range trends of common curves. Compare with Figure 5.2.

Some functions experience *exponential* growth (see Figures 5.2 and 5.3). The functions are $O(c^n)$, where c is a constant greater than 1. Enumerating all strings of length n or checking topological equivalence of circuits with n devices are classic examples of exponential algorithms. Constructing a list of the n-digit palindromes requires exponential time and space. The demands of an exponential process grow too quickly to make effective use of resources. As a result, we often think of functions with exponential behavior as being *intractable*. In the next section we will see that some recursive solutions to problems are exponential. While these solutions are not directly useful, simple insights can sometimes make these algorithms efficient.

"2002" is a palindrome.

5.1.2 Examples

A "Difference Table"

Suppose we're interested in printing a 10 by 10 table of differences between two integers (`row-col`) values. Each value in the table corresponds to the result of subtracting the row number from the column number:

Analysis

```
0 -1 -2 -3 -4 -5 -6 -7 -8 -9
1 0 -1 -2 -3 -4 -5 -6 -7 -8
2 1 0 -1 -2 -3 -4 -5 -6 -7
3 2 1 0 -1 -2 -3 -4 -5 -6
4 3 2 1 0 -1 -2 -3 -4 -5
5 4 3 2 1 0 -1 -2 -3 -4
6 5 4 3 2 1 0 -1 -2 -3
7 6 5 4 3 2 1 0 -1 -2
8 7 6 5 4 3 2 1 0 -1
9 8 7 6 5 4 3 2 1 0
```

As with most programs that generate two-dimensional output, we consider the use of a nested pair of loops:

```
public static void diffTable(int n)
// pre: n >= 0
// post: print difference table of width n
{
    for (int row = 1; row <= n; row++)       // 1
    {
        for (int col = 1; col <= n; col++)   // 2
        {
            System.out.print(row-col+" ");   // 3
        }
        System.out.println();                // 4
    }
}
```

Each of the loops executes n times. Since printing a value (line 3) takes constant time c_1, the inner loop at line 2 takes $c_1 n$ time. If line 4 takes constant time c_2,

then the outer loop at line 1 takes $n(c_1n+c_2) = c_1n^2+c_2n$ time. This polynomial is clearly $O(n^2)$ (take $c = c_1 + c_2$ and $n_0 = 1$). Doubling the problem size approximately quadruples the running time.

As a rule of thumb, each loop that performs n iterations multiplies the complexity of each iteration by a factor of n. Nested loops multiply the complexity of the most deeply nested code by another power of n. As we have seen, loops doubly nested around a simple statement often consume quadratic time.

Since there are only three variables in our difference method, it takes constant space—an amount of space that is independent of problem size.

A Multiplication Table

Unlike the difference operator, the multiplication operator is commutative, and the multiplication table is symmetric. Therefore, when printing a multiplication table of size n, only the "lower triangular" region is necessary:

```
1
2 4
3 6 9
4 8 12 16
5 10 15 20 25
6 12 18 24 30 36
7 14 21 28 35 42 49
8 16 24 32 40 48 56 64
9 18 27 36 45 54 63 72 81
10 20 30 40 50 60 70 80 90 100
```

Here is a Java method to print the above table:

```
public static void multTable(int n)
// pre: n >= 0
// post: print multiplication table
{
    for (int row = 1; row <= n; row++)       // 1
    {
        for (int col = 1; col <= row; col++) // 2
        {
            System.out.print(row*col+" ");   // 3
        }
        System.out.println();                // 4
    }
}
```

Clearly, this table can be printed at least as fast as our difference table—it has about half the entries—so it is, similarly, $O(n^2)$. Can this limit be improved? If lines 3 and 4 take constant times c_1 and c_2, respectively, then the overall time is approximately

$$(1c_1+c_2)+(2c_1+c_2)+\cdots+(nc_1+c_2) = \frac{c_1n(n+1)}{2}+nc_2 = \frac{c_1}{2}n^2+(c_2+\frac{c_1}{2})n$$

Clearly, no linear function will bound this function above, so the bound of $O(n^2)$ is a good estimate. Notice that we have, essentially, used the fastest growing term of the polynomial—n^2.

Notice that both of these programs print an "area" of values, each of which can be computed in constant time. Thus, the growth rate of the function is the growth rate of the area of the output—which is $O(n^2)$.

Building a Vector of Values

Often, it is useful to build a `Vector` containing specific values. For the purposes of this problem, we will assume the values are integers between 0 and $n - 1$, inclusive. Our first (and best) attempt expands the `Vector` in the natural manner:

```
public static Vector<Integer> buildVector1(int n)
// pre: n >= 0
// post: construct a vector of size n of 1..n
{
    Vector<Integer> v = new Vector<Integer>(n); // 1
    for (int i = 0; i < n; i++)             // 2
    {
        v.add(i);                           // 3
    }
    return v;                               // 4
}
```

We will assume (correctly) that lines 1 and 4 take constant time. The loop at line 2, however, takes n times the length of time it takes to add a single element. Review of that code will demonstrate that the addition of a new element to a `Vector` takes constant time, provided expansion is not necessary. Thus, the total running time is linear, $O(n)$. Notice that the process of building this `Vector` requires space that is linear as well. Clearly, if the method's purpose is to spend time initializing the elements of a `Vector`, it would be difficult for it to consume space at a faster rate than time.

A slightly different approach is demonstrated by the following code:

```
public static Vector<Integer> buildVector2(int n)
// pre: n >= 0
// post: construct a vector of size n of 1..n
{
    Vector<Integer> v = new Vector<Integer>(n);              // 1
    for (int i = 0; i < n; i++)             // 2
    {
        v.add(0,i);            // 3
    }
    return v;                               // 4
}
```

All the assumptions of `buildVector1` hold here, except that the cost of inserting a value at the *beginning* of a `Vector` is proportional to the `Vector`'s current length. On the first insertion, it takes about 1 unit of time, on the second, 2 units, and so on. The analysis of this method, then, is similar to that of the triangular multiplication table. Its running time is $O(n^2)$. Its space utilization, however, remains linear.

Printing a Table of Factors

Suppose we are interested in storing a table of factors of numbers between 1 and n. The beginning of such a table—the factors of values between 1 and 10—includes the following values:

```
1
1 2
1 3
1 2 4
1 5
1 2 3 6
1 7
1 2 4 8
1 3 9
1 2 5 10
```

How much space must be reserved for storing this table? This problem looks a little daunting because the number of factors associated with each line varies, but without any obvious pattern. Here's a program that generates the desired table:

```
public static Vector<Vector<Integer>> factTable(int n)
// pre: n > 0
// post: returns a table of factors of values 1 through n
{
    Vector<Vector<Integer>> table = new Vector<Vector<Integer>>();
    for (int i = 1; i <= n; i++)
    {
        Vector<Integer> factors = new Vector<Integer>();
        for (int f = 1; f <= i; f++)
        {
            if ((i % f) == 0) {
                factors.add(f);
            }
        }
        table.add(factors);
    }
    return table;
}
```

To measure the table size we consider those lines that mention f as a factor. Clearly, f appears on every fth line. Thus, over n lines of the table there are

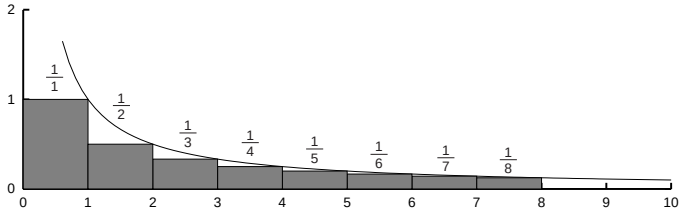


Figure 5.4 Estimating the sum of reciprocal values. Here, $\sum_{x=1}^{8} \frac{1}{x} \approx 2.72$ is no more than $1 + \int_1^8 \frac{1}{x} dx = 1 + \ln 8 \approx 3.08$.

---

no more than $\frac{n}{f}$ lines that include f. Thus, we have as an upper bound on the table size:

$$\frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{n-1} + \frac{n}{n}$$

Factoring out n we have:

$$n\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} + \frac{1}{n}\right)$$

We note that these fractions fall on the curve $\frac{1}{x}$ (see Figure 5.4). We may compute the area of the curve—an upper bound on the sum—as:

$$\begin{aligned} n\sum_{x=1}^{n} \frac{1}{x} &\leq n\left(1 + \int_1^n \frac{1}{x} dx\right) \\ &\leq n(1 + \ln n - \ln 1) \\ &\leq O(n \ln n) \end{aligned}$$

The size of the table grows only a little faster than linearly. The time necessary to create the table, of course, is $O(n^2)$ since we check n factors for number n.

Exercise 5.1 *Slightly modify the method to construct the same table, but in $O(n\sqrt{n})$ time.*

Exercise 5.2 *Rewrite the method to construct the same table, but in $O(n \ln n)$ time.*

Finding a Space in a String

Some problems appear to have behavior that is more variable than the examples we have seen so far. Consider, for example, the code to locate the first space in a string:

```
static int findSpace(String s)
// pre: s is a string, possibly containing a space
// post: returns index of first space, or -1 if none found
{
    int i;
    for (i = 0; i < s.length(); i++)
    {
        if (' ' == s.charAt(i)) return i;
    }
    return -1;
}
```

This simple method checks each of the characters within a string. When one is found to be a space, the loop is terminated and the index is returned. If, of course, there is no space within the string, this must be verified by checking each character. Clearly, the time associated with this method is determined by the number of loops executed by the method. As a result, the time taken is linear in the length of the string.

We can, however, be more precise about its behavior using *best-*, *worst-*, and *average-case analyses*:

Best case. The best-case behavior is an upper bound on the shortest time that any problem of size n might take. Usually, best cases are associated with particularly nice arrangements of values—here, perhaps, a string with a space in the first position. In this case, our method takes at most constant time! It is important to note that the best case must be a problem of size n.

Worst case. The worst-case behavior is the longest time that any problem of size n might take. In our string-based procedure, our method will take the longest when there is no space in the string. In that case, the method consumes at most linear time. Unless we specify otherwise, we will use the worst-case consumption of resources to determine the complexity.

Average case. The average-case behavior is the complexity of solving an "average" problem of size n. Analysis involves computing a weighted sum of the cost (in time or space) of problems of size n. The weight of each problem is the probability that the problem would occur. If, in our example, we knew (somehow) that there was exactly one space in the string, and that it appears in any of the n positions with equal probability, we would deduce that, on average,

$$\sum_{i=1}^{n} \frac{1}{n} i = \frac{1}{n} \cdot 1 + \frac{1}{n} \cdot 2 + \cdots + \frac{1}{n} \cdot n = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

iterations would be necessary to locate the space. Our method has linear average-time complexity. If, however, we knew that the string was English prose of length n, the average complexity would be related to the average length of the first word, a value easily bounded above by a constant (say, 10). The weights of the first few terms would be large, while the weights associated with a large number of iterations or more would be zero. The average complexity would be constant. (In this case, the worst case would be constant as well.) Obviously determining the average-case complexity requires some understanding of the desired distributions of data.

German prose may require larger constants.

Best-, worst-, and average-case analyses will be important in helping us evaluate the theoretical complexities of the structures we develop. Some care, however, must be used when determining the growth rates of real Java. It is tempting, for example, to measure the space or time used by a data structure and fit a curve to it in hopes of getting a handle on its long-term growth. This approach should be avoided, if possible, as such statements can rarely be made with much security. Still, such techniques can be fruitfully used to verify that there is no *unexpected* behavior.

5.1.3 The Trading of Time and Space

Two resources coveted by programmers are time and space. When programs are run, the algorithms they incorporate and the data structures they utilize work together to consume time. This time is directly due to executing machine instructions. The fewer instructions executed, the faster the program goes.

Most of us have had an opportunity to return to old code and realize that useless instructions can be removed. For example, when we compute the table factors, we realized that we could speed up the process by checking fewer values for divisibility. Arguably, most programs are susceptible to some of this "instruction weeding," or *optimization*. On the other hand, it is clear that there must be a limit to the extent that an individual program can be improved. For some equivalent program, the removal of any statement causes the program to run incorrectly. This limit, in some sense, is an *information theoretic limit*: given the approach of the algorithm and the design of a data structure, no improvements can be made to the program to make it run faster. To be convinced that there is a firm limit, we would require a formal proof that no operation could be avoided. Such proofs can be difficult, especially without intimate knowledge of the language, its compiler, and the architecture that supports the running code. Nonetheless, the optimization of code is an important feature of making programs run quickly. Engineers put considerable effort into designing compilers to make automated optimization decisions. Most compilers, for example, will not generate instructions for *dead code*—statements that will never be executed. In the following Java code, for example, it is clear that the "then" portion of this code may be removed without fear:

```
if (false)
{
     System.out.println("Man in the moon.");
} else {
     System.out.println("Pie in the sky.");
}
```

After compiler optimizations have been employed, though, there is a limit that can be placed on how fast the code can be made to run. We will assume—whenever we consider a time–space trade-off—that all reasonable efforts have been made to optimize the time and space utilization of a particular approach. Notice, however, that most optimizations performed by a compiler do not significantly affect the *asymptotic* running time of an algorithm. At most, they tend to speed up an algorithm by a constant factor, an amount that is easily absorbed in any theoretical analysis using big-O methods.

Appropriately implemented data structures can, however, yield significant performance improvements. Decisions about data structure design involve weighing—often using results of big-O analysis—the time and space requirements of a structure used to solve a problem. For example, in the `Vector` class, we opted to maintain a field, `elementCount`, that kept track of how many elements within the underlying array are actually being used. This variable became necessary when we realized that as the `Vector` expanded, the constant reallocation of the underlying memory could lead to quadratic time complexity over the life of the `Vector`. By storing a little more information (here, `elementCount`) we reduce the total complexity of expanding the `Vector`—our implementation, recall, requires $O(1)$ data-copying operations as the `Vector` expands. Since `Vectors` are very likely to expand in this way, we find it worthwhile to use this extra space. In other situations we will see that the trade-offs are less obvious and sometimes lead to the development of several implementations of a single data structure designed for various uses by the application designer.

The choice between implementations is sometimes difficult and may require analysis of the application: if `Vector`'s `add` method is to be called relatively infrequently, the time spent resizing the structure is relatively insignificant. On the other hand, if elements are to be added frequently, maintaining `elementCount` saves time. In any case, the careful analysis of trade-off between time and space is an important part of good data structure design.

5.1.4 Back-of-the-Envelope Estimations

A skill that is useful to the designer is the ability to develop good estimates of the time and space necessary to run an algorithm or program. It is one thing to develop a theoretical analysis of an algorithm, but it is quite another to develop a sense of the actual performance of a system. One useful technique is to apply any of a number of back-of-the-envelope approximations to estimating the performance of an algorithm.

The numbers that programmers work with on a day-to-day basis often vary in magnitude so much that it is difficult to develop much of a common sense

for estimating things. It is useful, then, to keep a store of some simple figures that may help you to determine the performance—either in time or space—of a project. Here are some useful rules of thumb:

- Light travels one foot in a *nanosecond* (one billionth of a second).
- Approximately π (≈ 3.15) hundredths of a second is a nanoyear (one billionth of a year).
- It takes between 1 and 10 nanoseconds (ns) to store a value in Java. Basic math operations take a similar length of time.
- An array assignment is approximately twice as slow as a regular assignment.
- A `Vector` assignment is approximately 50 times slower than a regular assignment.
- Modern computers execute 1 billion instructions per second.
- A character is represented by 8 bits (approximately 10).
- An Ethernet network can transmit at 100 million bits per second (expected throughput is nearer 10 million bits).
- Fewer than 100 words made up 50 percent of Shakespeare's writing; they have an average length of π. A core of 3000 words makes up 90 percent of his vocabulary; they have an average of 5.5 letters.

As an informal example of the process, we might attempt to answer the question: How many books can we store on a 10 gigabyte hard drive? First we will assume that 1 byte is used to store a character. Next, assuming that an average word has about 5 characters, and that a typewritten page has about 500 words per typewritten page, we have about 2500 characters per page. Another approximation might suggest 40 lines per page with 60 characters per line, or 2400 characters per page. For computational simplicity, we keep the 2500 character estimate. Next, we assume the average book has, say, 300 pages, so that the result is 0.75 million bytes required to store a text. Call it 1 million. A 10 gigabyte drive contains approximately 10 billion characters; this allows us to store approximately 10 thousand books.

A dictionary is a collection of approximately 250,000 words. How long might it take to compute the average length of words appearing in the dictionary? Assume that the dictionary is stored in memory and that the length of a word can be determined in constant time—perhaps 10 microseconds (μs). The length must be accumulated in a sum, taking an additional microsecond per word—let's ignore that time. The entire summation process takes, then, 2.5 seconds of time. (On the author's machine, it took 3.2 seconds.)

Exercise 5.3 *How many dots can be printed on a single sheet of paper? Assume, for example, your printer prints at 500 dots per inch. If a dot were used to represent a bit of information, how much text could be encoded on one page?*

As you gain experience designing data structures you will also develop a sense of the commitments necessary to support a structure that takes $O(n^2)$ space, or an algorithm that uses $O(n \log n)$ time.

5.2 Self-Reference

One of the most elegant techniques for constructing algorithms, data structures, and proofs is to utilize self-reference in the design. In this section we discuss applications of self-reference in programming—called *recursion*—and in proofs—called *proof by induction*. In both cases the difficulties of solving the problem outright are circumvented by developing a language that is rich enough to support the self-reference. The result is a compact technique for solving complex problems.

5.2.1 Recursion

When faced with a difficult problem of computation or structure, often the best solution can be specified in a *self-referential* or *recursive* manner. Usually, the difficulty of the problem is one of management of the resources that are to be used by the program. Recursion helps us tackle the problem by focusing on reducing the problem to one that is more manageable in size and then building up the answer. Through multiple, nested, progressive applications of the algorithm, a solution is constructed from the solutions of smaller problems.

Summing Integers

We first consider a simple, but classic, problem: suppose we are interested in computing the sum of the numbers from 0 through n.

$$\sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n$$

One approach to the problem is to write a simple loop that over n iterations accumulates the result.

Recursion

```
public static int sum1(int n)
// pre: n >= 0
// post: compute the sum of 0..n
{
    int result = 0;
    for (int i = 1; i <= n; i++)
    {
```

```
            result = result + i;
        }
        return result;
    }
```

The method starts by setting a partial sum to 0. If n is a value that is less than 1, then the loop will never execute. The result (0) is what we expect if $n = 0$. If n is greater than 0, then the loop executes and the initial portion of the partial sum is computed. After $n-1$ loops, the sum of the first $n-1$ terms is computed. The nth iteration simply adds in n. When the loop is finished, `result` holds the sum of values 1 through n. We see, then, that this method works as advertised in the postcondition.

Suppose, now, that a second programmer is to solve the same problem. If the programmer is particularly lazy and has access to the `sum1` solution the following code also solves the problem:

```
public static int sum2(int n)
// pre: n >= 0
// post: compute the sum of 0..n
{
    if (n < 1) return 0;
    else return sum1(n-1) + n;
}
```

For the most trivial problem (any number less than 1), we return 0. For all other values of n, the programmer turns to `sum1` to solve the next simplest problem (the sum of integers 0 through $n-1$) and then adds n. Of course, this algorithm works as advertised in the postcondition, because it depends on `sum1` for all but the last step, and it then adds in the correct final addend, n.

Actually, if `sum2` calls *any* method that is able to compute the sum of numbers 0 through $n-1$, `sum2` works correctly. But, wait! The sum of integers is precisely what `sum2` is supposed to be computing! We use this observation to derive, then, the following *self-referential* method:

```
public static int sum3(int n)
// pre: n >= 0
// post: compute the sum of 0..n
{
    if (n < 1) return 0;        // base case
    else return sum3(n-1) + n;  // reduction, progress, solution
}
```

This code requires careful inspection (Figure 5.5). First, in the simplest or *base* cases (for $n < 1$), `sum3` returns 0. The second line is only executed when $n \geq 1$. It reduces the problem to a simpler problem—the sum of integers between 0 and $n-1$. As with all recursive programs, this requires a little work (a subtraction) to reduce the problem to one that is closer to the base case. Considering the problem $n+1$ would have been fatal because it doesn't make suitable *progress*

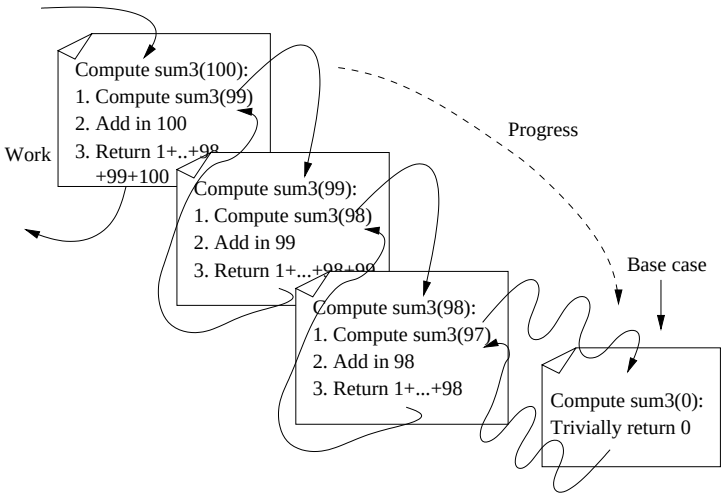


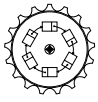
Figure 5.5 The "unrolling" of a procedure to recursively sum integers. Rightward arrows break the problem down; leftward arrows build up the solution.

toward the base case. The subproblem is passed off to *another invocation* of `sum3`. Once that procedure computes its result (either immediately or, if necessary, through further recursion), a little more work is necessary to convert the solution of the problem of size $n-1$ into a solution for a problem of size n. Here, we have simply added in n. Notice the operation involved in building the answer (addition) opposes the operation used to reduce the problem (subtraction). This is common in recursive procedures.

Principle 8 *Recursive structures must make "progress" toward a "base case."*

We cast this principle in terms of "structures" because much of what we say about self-referential execution of code can be applied to self-referential structuring of data. Most difficulties with recursive structures (including recursive methods) stem from either incorrectly stating the base case or failing to make proper progress.

Inserting a Value into a Vector



`Vector`

Recursion is a natural method for accomplishing many complicated tasks on `Vectors` and arrays. For example, the `add(index,object)` method of the `Vector` class discussed on page 51 can be written as a recursive procedure. The essential concept is to insert the value into the `Vector` only after having moved the previous value out of the way. That value is inserted at the next larger location. This leads us to the following alternative to the standard `Vector` method:

```
public void add(int index, E value)
// pre: 0 <= index <= size()
// post: inserts new value in vector with desired index
//   moving elements from index to size()-1 to right
{
    if (index >= size()) {
        add(value); // base case: add at end
    } else {
        E previous = get(index); // work
        add(index+1,previous);  // progress through recursion
        set(index,value);  // work
    }
}
```

Note that the base case is identified through the need to apply a *trivial operation* rather than, say, the size of the index. Indeed, progress is determined by how close the index gets to the size of the `Vector`. Again, this is a linear or $O(n)$ process.

Printing a Vector of Values

In the previous example, the recursive routine was suitable for direct use by the user. Often, though, recursion demands additional parameters that encode, in some way, progress made toward the solution. These parameters can be confusing to users who, after all, are probably unaware of the details of the recursion. To avoid this confusion, we "wrap" the call to a protected recursive method in a public method. This hides the details of the initial recursive method call. Here, we investigate a printing extension to the `Vector` class:

```
public void print()
// post: print the elements of the vector
{
    printFrom(0);
}

protected void printFrom(int index)
// pre: index <= size()
// post: print elements indexed between index and size()
{
    if (index < size()) {
        System.out.println(get(index));
        printFrom(index+1);
    }
}
```

The `print` method wraps or hides the call to the recursive `printFrom` method. The recursive method accepts a single parameter that indicates the index of the first element that should be printed out. As progress is made, the initial

index increases, leading to linear performance. To print the entire `Vector`, the recursive method is called with a value of zero.

It would appear that the base case is missing. In fact, it is indicated by the *failure* of the `if` statement. Even though the base case is to "do nothing," the `if` statement is absolutely necessary. Every terminating recursive method should have some conditional statement.

`PrintFrom` is an example of a *tail recursive* method. Any recursion happens just before exiting from the method. Tail recursive methods are particularly nice because good compilers can translate them into loops. Each iteration of the loop simulates the computation and return of another of the nested recursive procedure calls. Since there is one call for each of the n values, and the procedure performs a constant amount of work, the entire process takes $O(n)$ time.

Exercise 5.4 *Write a recursive method to print out the characters of a string with spaces between characters. Make sure your method does not print a leading or tailing space, unless it is a leading or trailing character of the original string.*

Computing Change in Postage Stamps

Suppose, when receiving change at the post office, you wished to be paid your change in various (useful) stamps. For example, at current rates, you might be interested in receiving either 39 cent stamps, 24 cent postcards, or penny stamps (just in case of a postage increase). For a particular amount, what is *the smallest number of stamps necessary* to make the change?

This problem is fairly complex because, after all, the minimum number of stamps needed to make 50 cents involves 4 stamps—two postcard stamps and two penny stamps—and *not* 12—a 39 cent stamp and 11 penny stamps. (The latter solution might be suggested by postal clerks used to dealing with U.S. coinage, which is fairly easily minimized.) We will initially approach this problem using recursion. Our solution will only report the minimum number of stamps returned. We leave it as an exercise to report the number of each type of stamp (consider Problem 5.22); that solution does not greatly change the approach of the problem.

If no change is required—a base case—the solution is simple: hand the customer zero stamps. If the change is anything more, we'll have to do some work. Consider the 70 cent problem. We know that some stamps will have to be given to the customer, but not the variety. We *do* know that the last stamp handed to the customer will either be a penny stamp, a 26 cent step, or a 41 cent stamp. If we could only solve three smaller minimization problems—the 69 cent problem, the 34 cent problem, and the 29 cent problem—then our answer would be one stamp more than the minimum of the answers to those three problems. (The answers to the three problems are 4, 9, and 4, respectively, so our answer should be 5.) Of course, we should ignore meaningless reduced problems: the -6 cent problem results from considering handing a 26 cent stamp over to solve the 20 cent problem.

`Recursive-Postage`

Here is the `stampCount` method that computes the solution:

```
public final static int LETTER=41;
public final static int CARD=26;
public final static int PENNY=1;

public static int stampCount(int amount)
// pre: amount >= 0
// post: return *number* of stamps needed to make change
//       (only use letter, card, and penny stamps)
{
    int minStamps;
    Assert.pre(amount >= 0,"Reasonable amount of change.");
    if (amount == 0) return 0;
    // consider use of a penny stamp
    minStamps = 1+stampCount(amount-PENNY);
    // consider use of a post card stamp
    if (amount >= CARD) {
        int possible = 1+stampCount(amount-CARD);
        if (minStamps > possible) minStamps = possible;
    }
    // consider use of a letter stamp
    if (amount >= LETTER) {
        int possible = 1+stampCount(amount-LETTER);
        if (minStamps > possible) minStamps = possible;
    }
    return minStamps;
}
```

For the nontrivial cases, the variable `minStamps` keeps track of the minimum number of stamps returned by any of these three subproblems. Since each method call potentially results in several recursive calls, the method is not tail recursive. While it is possible to solve this problem using iteration, recursion presents a very natural solution.

An Efficient Solution to the Postage Stamp Problem

Making currency is illegal. Making change is not!

If the same procedure were used to compute the minimum number of stamps to make 70 cents change, the `stampCount` procedure would be called 2941 times. This number increases exponentially as the size of the problem increases (it is $O(3^n)$). Because 2941 is greater than 70—the number of distinct subproblems—some subproblems are recomputed many times. For example, the 2 cent problem must be re-solved by every larger problem.

To reduce the number of calls, we can incorporate an array into the method. Each location `n` of the array stores either 0 or the answer to the problem of size `n`. If, when looking for an answer, the entry is 0, we invest time in computing the answer and *cache* it in the array for future use. This technique is called *dynamic programming* and yields an efficient linear algorithm. Here is our modified solution:

`FullPostage`

```
public static final int LETTER = 41; // letter rate
public static final int CARD = 26;   // post card rate
public static final int PENNY = 1;   // penny stamp

public static int stampCount(int amount)
// pre: amount >= 0
// post: return *number* of stamps needed to make change
//       (only use letter, post card, and penny stamps)
{
    return stampCount(amount, new int[amount+1]);
}

protected static int stampCount(int amount, int answer[])
// pre: amount >= 0; answer array has length >= amount
// post: return *number* of stamps needed to make change
//       (only use letter, post card, and penny stamps)
{
    int minStamps;
    Assert.pre(amount >= 0,"Reasonable amount of change.");
    if (amount == 0) return 0;
    if (answer[amount] != 0) return answer[amount];
    // consider use of a penny stamp
    minStamps = 1+stampCount(amount-1,answer);
    // consider use of a post card stamp
    if (amount >= CARD) {
        int possible = 1+stampCount(amount-CARD,answer);
        if (minStamps > possible) minStamps = possible;
    }
    // consider use of a letter stamp
    if (amount >= LETTER) {
        int possible = 1+stampCount(amount-LETTER,answer);
        if (minStamps > possible) minStamps = possible;
    }
    answer[amount] = minStamps;
    return minStamps;
}
```

When we call the method for the first time, we allocate an array of sufficient size (`amount+1` because arrays are indexed beginning at zero) and pass it as `answer` in the protected two-parameter version of the method. If the answer is not found in the array, it is computed using up to three recursive calls that pass the array of previously computed answers. Just before returning, the newly computed answer is placed in the appropriate slot. In this way, when solutions are sought for this problem again, they can be retrieved without the overhead of redundant computation.

When we seek the solution to the 70 cent problem, 146 calls are made to the procedure. Only a few of these get past the first few statements to potentially make recursive calls. The combination of the power recursion and the efficiency of dynamic programming yields elegant solutions to many seemingly difficult

problems.

Exercise 5.5 *Explain why the dynamic programming approach to the problem runs in linear time.*

In the next section, we consider *induction*, a recursive proof technique. Induction is as elegant a means of proving theorems as recursion is for writing programs.

5.2.2 Mathematical Induction

The accurate analysis of data structures often requires mathematical proof. An effective proof technique that designers may apply to many computer science problems is *mathematical induction*. The technique is, essentially, the construction of a recursive proof. Just as we can solve some problems elegantly using recursion, some properties may be elegantly verified using induction.

A common template for proving statements by mathematical induction is as follows:

1. Begin your proof with "We will prove this using induction on the size of the problem." This informs the reader of your approach.
2. Directly prove whatever base cases are necessary. Strive, whenever possible to keep the number of cases small and the proofs as simple as possible.
3. State the assumption that the observation holds for all values from the base case, up to but not including the nth case. Sometimes this assumption can be relaxed in simple inductive proofs.
4. Prove, from simpler cases, that the nth case also holds.
5. Claim that, by mathematical induction on n, the observation is true for all cases more complex than the base case.

Individual proofs, of course, can deviate from this pattern, but most follow the given outline.

As an initial example, we construct a formula for computing the sum of integers between 0 and $n \geq 0$ inclusively. Recall that this result was used in Section 3.5 when we considered the cost of extending `Vectors`, and earlier, in Section 5.1.2, when we analyzed `buildVector2`. Proof of this statement also yields a constant-time method for implementing `sum3`.

Observation 5.1 $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

Proof: We prove this by induction. First, consider the simplest case, or *base case*. If $n = 0$, then the sum is 0. The formula gives us $\frac{0(0+1)}{2} = 0$. The observation appears to hold for the base case.

Now, suppose we know—for some reason—that our closed-form formula holds for all values between 0 (our base case) and $n - 1$. This knowledge may

help us solve a more complex problem, namely, the sum of integers between 0 and n. The sum

$$0 + 1 + 2 + \cdots + (n-1) + n$$

conveniently contains the sum of the first $n-1$ integers, so we rewrite it as

$$[0 + 1 + 2 + \cdots + (n-1)] + n$$

Because we have assumed that the sum of the natural numbers to $n-1$ can be computed by the formula, we may rewrite the sum as

$$\left[\frac{(n-1)n}{2}\right] + n$$

The terms of this expression may be simplified and reorganized:

$$\frac{(n-1)n + 2n}{2} = \frac{n(n+1)}{2}$$

Thus given only the knowledge that the formula worked for $n-1$, we have been able to extend it to n. It is not hard to convince yourself, then, that the observation holds for any nonnegative value of n. Our base case was for $n = 0$, so it must hold as well for $n = 1$. Since it holds for $n = 1$, it must hold for $n = 2$. In fact, it holds for any value of $n \geq 0$ by simply proving it holds for values $0, 1, 2, \ldots, n-1$ and then observing it can be extended to n.◇

99 cases left to prove! Take one down, pass it around, 98 cases left to prove! . . .

The induction can be viewed as a recursively constructed proof (consider Figure 5.6). Suppose we wish to see if our observation holds for $n = 100$. Our method requires us to show it holds for $n = 99$. Given that, it is a simple matter to extend the result to 100. Proving the result for $n = 99$, however, is *almost*[2] as difficult as it is for $n = 100$. We need to prove it for $n = 98$, and extend *that* result. This process of developing the proof for 100 eventually unravels into a recursive construction of a (very long) proof that demonstrates that the observation holds for values 0 through 99, and then 100.

The whole process, like recursion, depends critically on the proof of appropriate base cases. In our proof of Observation 5.1, for example, we proved that the observation held for $n = 0$. If we do not prove this simple case, then our recursive construction of the proof for any value of $n \geq 0$ does not terminate: when we try to prove it holds for $n = 0$, we have no base case, and therefore must prove it holds for $n = -1$, and in proving that, we prove that it holds for $-2, -3, \ldots$, *ad infinitum*. The proof construction never terminates!

Our next example of proof by induction is a *correctness proof*. Our intent is to show that a piece of code runs as advertised. In this case, we reinvestigate `sum3` from page 95:

Recursion

[2] It is important, of course, to base your inductive step on simpler problems—problems that take you closer to your base case. If you avoid basing it on simpler cases, then the recursive proof will never be completely constructed, and the induction will fail.

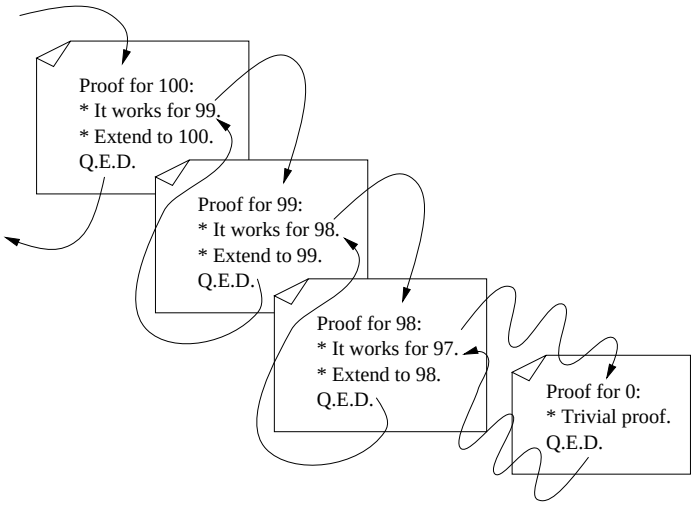


Figure 5.6 The process of proof by induction simulates the recursive construction of a proof. Compare with Figure 5.5.

```
public static int sum3(int n)
// pre: n >= 0
// post: compute the sum of 0..n
{
    if (n < 1) return 0;        // 1
    else return                 // 2
                sum3(           // 3
                     n-1        // 4
                        ) + n;  // 5
}
```

(The code has been reformatted to allow discussion of portions of the computation.) As with our mathematical proofs, we state our result formally:

Observation 5.2 *Given that* $n \geq 0$*, the method* `sum3` *computes the sum of the integers 0 through* n*, inclusive.*

Proof: Our proof is by induction, based on the parameter `n`. First, consider the action of `sum3` when it is passed the parameter 0. The `if` statement of line 1 is true, and the program returns 0, the desired result.

We now consider `n>0` and assume that the method computes the correct result for all values less that `n`. We extend our proof of correctness to the parameter value of `n`. Since `n` is greater than 0, the `if` of line 1 fails, and the `else` beginning on line 2 is considered. On line 4, the parameter is decremented, and on line 3, the recursion takes place. By our assumption, this recursive

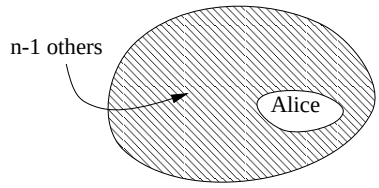


Figure 5.7 A group of n computer scientists composed of Alice and $n-1$ others.

call returns the correct result—the sum of values between 0 and `n-1`, inclusive. Line 5 adds in the final value, and the entire result is returned. The program works correctly for a parameter `n` greater than 0. By induction on `n`, the method computes the correct result for all `n>=0`.◇

Proofs of correctness are important steps in the process of verifying that code works as desired. Clearly, since induction and recursion have similar forms, the application of inductive proof techniques to recursive methods often leads to straightforward proofs. Even when iteration is used, however, induction can be used to demonstrate assumptions made about loops, no matter the number of iterations.

We state here an important result that gives us a closed-form expression for computing the sum of powers of 2.

Observation 5.3 $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.

Exercise 5.6 *Prove Observation 5.3.*

There are, of course, ways that the inductive proof can go awry. Not proving the appropriate base cases is the most common mistake and can lead to some interesting results. Here we prove what few have suspected all along:

Observation 5.4 *All computer scientists are good programmers.*

Warning: bad proof!

Proof: We prove the observation is true, using mathematical induction. First, we use traditional techniques (examinations, etc.) to demonstrate that Alice is a good programmer.

Now, assume that our observation is true of any group of fewer than n computer scientists. Let's extend our result: select n computer scientists, including Alice (see Figure 5.7). Clearly, the subgroup consisting of all computer scientists that are "not Alice" is a group of $n-1$ computer scientists. Our assumption states that this group of $n-1$ computer scientists is made up of good programmers. So Alice and all the other computer scientists are good programmers. By induction on n, we have demonstrated that all computer scientists are good programmers.◇

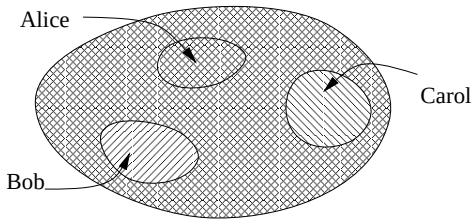


Figure 5.8 A group of n computer scientists, including Alice, Bob, and Carol.

This is a very interesting result, especially since it is not true. (Among other things, some computer scientists do not program computers!) How, then, were we successful in proving it? If you look carefully, our base case is Alice. The assumption, on the other hand, is based on *any* group of $n - 1$ programmers. Unfortunately, since our only solid proof of quality programming is Alice, and non-Alice programmers cannot be reduced to cases involving Alice, our proof is fatally flawed.

Still, a slight reworking of the logic might make the proof of this observation possible. Since Alice is a computer scientist, we can attempt to prove the observation by induction on groups of computer scientists *that include Alice:*

Warning: bad proof, take 2!

Proof: We prove the observation by induction. First, as our base case, consider Alice. Alice is well known for being a good programmer. Now, assume that for any group of fewer than n computer scientists that includes Alice, the members are excellent programmers. Take n computer scientists, including Alice (see Figure 5.8). Select a non-Alice programmer. Call him Bob. If we consider all non-Bob computer scientists, we have a group of $n - 1$ computer scientists—including Alice. By our assumption, they must all be good. What about Bob? Select another non-Alice, non-Bob computer scientist from the group of n. Call her Carol. Carol must be a good programmer, because she was a member of the $n - 1$ non-Bob programmers. If we consider the $n - 1$ non-Carol programmers, the group includes both Alice and Bob. Because it includes Alice, the non-Carol programmers must all be good. Since Carol is a good programmer, then all n must program well. By induction on n, all groups of computer scientists that include Alice must be good programmers. Since the group of all computer scientists is finite, and it includes Alice, the entire population must program well. The observation holds!$\diamond$

This proof looks pretty solid—until you consider that in order for it to work, you must be able to distinguish between Alice, Bob, and Carol. There are three people. The proof of the three-person case depends directly on the observation holding for just two people. But we have not considered the two-person case! In fact, *that* is the hole in the argument. If we know of a bad programmer, Ted, we can say nothing about the group consisting of Alice and Ted (see Figure 5.9).

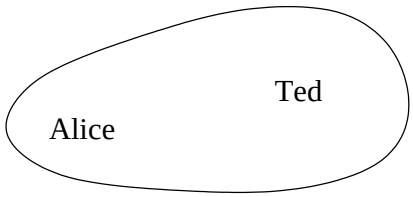


Figure 5.9 The proof does not hold for the simplest nontrivial case: Alice and any bad programmer.

As a result, we have a worrisome hole in the proof of the group consisting of Alice, Bob, and Ted. In the end, the attempt at a complete proof unravels.

Lesson: it's hard to find good programmers.

What have we learned from this discussion? For an inductive proof, the base cases must be carefully enumerated and proved. When proving the inductive step, the step must be made upon a proved foundation. If not, the entire statement collapses. The subtlety of this difficulty should put us on alert: even the most thoughtful proofs can go awry if the base case is not well considered.

We can now make a similar statement about recursion: it is important to identify and correctly code the base cases you need. If you don't, you run the risk that your method will fail to stop or will compute the wrong answer. One of the most difficult debugging situations occurs when multiple base cases are to be considered and only a few are actually programmed.

Our final investigation considers the implementation of a Java method to compute the following sequence of values:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

These values are the first of the sequence of *Fibonacci numbers*. Each value is the sum of the two values that fall before it. We should be careful—especially given our last discussion—that we have the base cases carefully considered. In this particular case, we must specify *two* initial values: 0 and 1.

This sequence may be familiar to you. If it is, you may have seen the definition of F_n, the nth value of the sequence as

$$F_n = \begin{cases} n & n = 0 \text{ or } n = 1 \\ F_{n-2} + F_{n-1} & n > 1 \end{cases}$$

The translation of this type of equation into Java is fairly straightforward. We make the following attempt:

Fibo

```
static public int fibo(int n)
// pre: n is a nonnegative integer
// post: result is the ith term from the sequence
//     0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .
```

```
{
    Assert.pre(n >= 0, "Index is nonnegative.");
    // when n < 2, return n
    if (n == 0) return 0;                   // line 1
    else if (n == 1) return 1;              // line 2
    // complex, self-referential case:
    else return fibo(n-2)+fibo(n-1);        // line 3
}
```

We now seek to prove that the recursive method computes and returns the nth member of the sequence.

Proof: First, suppose $n = 0$: the method returns 0, on line 1. Next, suppose that $n = 1$: the method returns 1, on line 2. So, for the two very simplest cases, the method computes the correct values. Now, suppose that $n > 1$, and furthermore, assume that `fibo` returns the correct value for all terms with index less than n. Since $n > 1$, lines 1 and 2 have no effect. Instead, the method resorts to using line 3 to compute the value. Since the method works for all values less than n, it specifically computes the two previous terms—F_{n-2} and F_{n-1}—correctly. The sum of these two values (F_n) is therefore computed and immediately returned on line 3. We have, then, by mathematical induction on n proved that $fibo(n)$ computes F_n for all $n \geq 0$.$\diamond$

Another approach to computing Fibonacci numbers, of course, would be to use an iterative method:

```
static public int fibo2(int n)
// pre: n is a nonnegative integer
// post: result is the ith term from the sequence
//     0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .
{
    Assert.pre(n >= 0, "Index is nonnegative.");
    int a = 0;
    int b = 1;
    if (n == 0) return a;       // line 1
    if (n == 1) return b;       // line 2
    // for large values of n, iteratively compute sequence
    int i=2,F;
    do
    {
        // Assertion: b is the i-1st member of the sequence
        //            a is the i-2nd member
        F = a + b;              // line 3
        // Assertion: F is the ith member
        // update previous two values:
        a = b;                  // line 4
        b = F;                  // line 5
        i++;                    // line 6
    } while (i <= n);           // line 7
    return F;                   // line 8
}
```

To demonstrate that such a program works, we perform a step-wise analysis of the method as it computes F_n.

Proof: Suppose $n = 0$. The condition in the `if` statement on line 1 is true and the value `a` (0) is returned. If $n = 1$, the condition on line 1 is false, but the `if` statement on line 2 is true and `b` (1) is returned. In both of these cases the correct value is returned.

We now investigate the loop. We notice that when the loop starts (it is a `do` loop, it must execute at least once if $n > 1$), `a` and `b` contain the values F_0 and F_1, and `i` is 2. Thus, the loop invariant before line 3 holds on the first iteration.

Now, assume that $i \geq 2$ and the loop invariant before line 3 holds. The effect of line 3 is to compute F_i from F_{i-1} and F_{i-2}. The result is placed in F, and the loop invariant after line 3 is met. The remaining statements, on lines 4 through 6 result in F_{i-2} in `a` and F_{i-1} in `b`. If the condition on line 7 should be true, we meet the loop invariant for the next iteration.

If the condition on line 7 should be false, then we note that this value of `i` is the first that is greater than `n`, so $F = F_{i-1} = F_n$, and the result returned is the correct result. $\diamond$

It is interesting to note that the initial values of the sequence are rather arbitrary, and that different natural phenomena related to Fibonacci numbers can be modeled by sequences that begin with different initial values.

5.3 Properties of Design

This section is dedicated to two informal properties of design that are referenced elsewhere within this text. The property of *symmetry* describes the predictability of a design, while *friction* describes the difficulty of moving a data structure from one state to another. Both terms extend the vocabulary of implementors when discussing design decisions.

5.3.1 Symmetry

For the most part, our instruction of computers occurs through programs. As a result, programs can be nonintuitive and hard to understand if they are not designed with human-readability in mind. On the other hand, a well-designed program can be used by a novice without a significant learning curve. Systems that are easy to use tend to survive longer.

The programmer, as a designer of a data structure, is responsible for delivering a usable implementation of an abstract data structure. For an implementation to be usable, it should provide access to the structure with methods that are predictable and easy to use. The notion of predictability is particularly difficult for designers of data structures to understand, and it is something often overlooked by novice programmers.

When designing a system (here, a program or data structure) a useful principle is to make its interface *symmetric*. What is symmetry? Symmetry allows one

to view a system from different points of view and see similarities. Mathematicians would say that a system exhibits a symmetry if "it looks like itself under a nontrivial transformation." Given this, programmers consider asymmetries in transformed programs to be early warning signs of errors in logic.

Consider the following method (you will see this as part of the `swap` procedure of page 120). It exchanges two object references—`data[i]` and `data[j]`.

```
int temp;
temp = data[i];
data[i] = data[j];
data[j] = temp;
```

Close inspection of this code demonstrates that it does what it claims to do. Even if we stand back, not thinking so much about the actual workings of the code, we can see that the code is pretty symmetric. For example, if we squint our eyes and look at the code from the standpoint of variable `data[i]`, we see it as:

```
... = data[i];
data[i] = ...;
```

Here, `data[i]` is assigned to a variable, and a value is assigned to `data[i]`. We see a similar pattern with `data[j]`:

```
... = data[j];
data[j] = ...;
```

While this is not direct proof that the code works, it is an indication that the code is, in some way, "symmetric," and that helps make the argument that it is well designed.

Not everything we do is symmetric. If we consider the `Association` class, for example, the key and value components of the `Association` are different. The value, of course, has two associated methods, `getValue` and `setValue`. The first of the methods reads and returns a value, while the second method consumes and sets a value. Everything is in balance, and so we are hopeful that the design of the structure is correct. On the other hand, the key can only be read: while there is a `getKey` method, there is no `setKey`. We have suggested good reasons for doing this. As long as you can make a good argument for asymmetry in design, the breaking of symmetry can be useful. Unreasoned asymmetry, however, is a sign of poor and unpredictable design.

Here are various ways that one can look at a system to evaluate it for symmetry:

1. Compare methods that extend the structure with methods that trim the structure. Do they have similar approaches? Are they similar in number?

2. Consider methods that read and write values. Can the input methods read what is written by the output methods? Can the writing methods write all values that can be read?

3. Are procedures that consume parameters matched by functions that deliver values?
4. Can points of potential garbage collection be equally balanced by `new` invocations?
5. In linked structures, does unlinking a value from the structure appear to be the reverse of linking a new value into the structure?

When asymmetries are found, it is important to consider why they occur. Arguments such as "I can't imagine that anyone would need an opposite method!" are usually unconvincing. Many methods are added to the structures, not because they are obviously necessary, but because there is no good argument against them. Sometimes, of course, the language or underlying system forces an asymmetry. In Java, for example, every `Object` has a `toString` method that converts an internal representation of an object to a human-readable form, but there's no `fromString` required method that reads the value of an `Object` from a `String`. There *should be*, but there isn't.

Should ≠ will.

5.3.2 Friction

One of the obvious benefits of a data structure is that it provides a means of storing information. The ease with which the structure accepts and provides information about its contents can often be determined by its interface. Likewise, the difficulty of moving a data structure from one state to another determines, in some way, its "stiffness" or the amount of *friction* the structure provides when the state of the structure is to be modified.

One way that we might measure friction is to determine a sequence of logical states for the structure, and then determine the number of operations that are necessary to move the structure from each state to the next. If the number of operations is high, we imagine a certain degree of friction; if the operation count is low, the structure moves forward with relative ease.

Often we see that the less space provided to the structure, the more friction appears to be inherent in its structure. This friction can be good—it may make it less possible to get our structure into states that are inconsistent with the definition of the structure, or it may be bad—it may make it difficult to get something done.

5.4 Conclusions

Several formal concepts play an important role in modern data structure design—the use of big-O analysis to support claims of efficiency, the use of recursion to develop concise but powerful structures, and the use of induction to prove statements made about both data structures and algorithms. Mastery of these concepts improves one's approach to solving problems of data structure design.

The purpose of big-O analysis is to demonstrate upper bounds on the growth of functions that describe behavior of the structures we use. Since these are upper bounds, the tightest bounds provide the most information. Still, it is often not very difficult to identify the fastest-growing component of a function—analysis of that component is likely to lead to fairly tight bounds and useful results.

Self-reference is a powerful concept. When used to develop methods, we call this recursion. Recursion allows us to break down large problems into smaller problems whose solutions can be brought together to solve the original problem. Interestingly, recursion is often a suitable substitute for loops as a means of progressing through the problem solution, but compilers can often convert tail recursive code back into loops, for better performance. All terminating recursive methods involve at least one test that distinguishes the base case from the recursive, and every recursive program must eventually make progress toward a base case to construct a solution.

Mathematical induction provides a means of recursively generating proofs. Perhaps more than most other proof mechanisms, mathematical induction is a useful method for demonstrating a bound on a function, or the correct termination of a method. Since computers are not (yet) able to verify everyday inductive proofs, it is important that they be constructed with appropriate care. Knowing how to correctly base induction on special cases can be tricky and, as we have recently seen, difficult to verify.

In all these areas, practice makes perfect.

Self Check Problems

Solutions to these problems begin on page 444.

5.1 Suppose $f(x) = x$. What is its best growth rate, in big-O notation?

5.2 Suppose $f(x) = 3x$. What is its growth rate?

5.3 What is the growth rate of $f(x) = x + 900$?

5.4 How fast does $f(x)$ grow if $f(x) = x$ for odd integers and $f(x) = 900$ for even integers?

5.5 Evaluate and order the functions $\log_2 x$, $\sqrt{x}$, x, $30x$, x^2, 2^x, and $x!$ at $x = 2$, 4, 16, and 64. For each value of x, which is largest?

5.6 What are three features of recursive programs?

5.7 The latest Harry Potter book may be read by as much as 75 percent of the reading child population in the United States. Approximately how many child-years of reading time does this represent?

5.8 Given an infinite supply of 37 cent stamps, 21 cent stamps, and penny stamps a postmaster returns a minimum number of stamps composed of $c_{37}(x)$, $c_{21}(x)$, and $c_1(x)$ stamps for x dollars in change. What are the growth rates of these functions?

Problems

Solutions to the odd-numbered problems begin on page 462.

5.1 What is the time complexity associated with accessing a single value in an array? The `Vector` class is clearly more complex than the array. What is the time complexity of accessing an element with the `get` method?

5.2 What is the worst-case time complexity of the index-based `remove` code in the `Vector` class? What is the best-case time complexity? (You may assume the `Vector` does not get resized during this operation.)

5.3 What is the running time of the following method?

```
public static int reduce(int n)
{
    int result = 0;
    while (n > 1)
    {
        n = n/2;
        result = result+1;
    }
    return result;
}
```

5.4 What is the time complexity of determining the length of an n-character `null`-terminated string? What is the time complexity of determining the length of an n-character counted string?

5.5 What is the running time of the following matrix multiplication method?

```
// square matrix multiplication
// m1, m2, and result are n by n arrays
for (int row = 0; row < n; row++)
{
    for (int col = 0; col < n; col++)
    {
        int sum = 0;
        for (int entry = 0; entry < n; entry++)
        {
            sum = sum + m1[row][entry]*m2[entry][col];
        }
        result[row][col] = sum;
    }
}
```

5.6 In Definition 5.1 we see what it means for a function to be an upper bound. An alternative definition provides a *lower bound* for a function:

Definition 5.2 *A function $f(n)$ is $\Omega(g(n))$ (read "big-omega of g" or "at least order g"), if and only if there exist two positive constants, c and n_0, such that*

$$f(n) \geq c \cdot g(n)$$

for all $n \geq n_0$.

What is a lower bound on the time it takes to remove a value from a `Vector` by index?

5.7 What is a lower bound on adding a value to the end of a `Vector`? Does it matter that sometimes we may have to spend time doubling the size of the underlying array?

5.8 When discussing symmetry, we investigated a procedure that swapped two values within an array. Is it possible to write a routine that swaps two integer values? If so, provide the code; if not, indicate why.

5.9 For subtle reasons `String` objects cannot be modified. Instead, `Strings` are used as parameters to functions that build new `Strings`. Suppose that `a` is an n-character `String`. What is the time complexity of performing `a=a+"!"`?

5.10 Read Problem 5.9. Suppose that `a` and `b` are n-character `Strings`. What is the complexity of performing `a=a+b`?

5.11 What is the rate of growth (using big-O analysis) of the function $f(n) = n + \log n$? Justify your answer.

5.12 In this text, logarithms are assumed to be in base 2. Does it make a difference, from a complexity viewpoint?

5.13 What is the rate of growth of the function $\frac{1}{n} + 12$? Justify your answer.

5.14 What is the rate of growth of the function $\frac{\sin n}{n}$? Justify your answer.

5.15 Trick question: What is the rate of growth of $\tan n$?

5.16 Suppose n integers between 1 and 366 are presented as input, and you want to know if there are any duplicates. How would you solve this problem? What is the rate of growth of the function $T(n)$, describing the time it takes for you to determine if there are duplicates? (Hint: Pick an appropriate n_0.)

5.17 The first element of a *Syracuse sequence* is a positive integer s_0. The value s_i (for $i > 0$) is defined to be $s_{i-1}/2$ if s_{i-1} is even, or $3s_{i-1} + 1$ if s_{i-1} is odd. The sequence is finished when a 1 is encountered. Write a procedure to print the Syracuse sequence for any integer s_0. (It is not immediately obvious that this method should always terminate.)

5.18 Rewrite the `sqrt` function of Section 2.1 as a recursive procedure.

5.19 Write a recursive procedure to draw a line segment between (x_0, y_0) and (x_1, y_1) on a screen of pixels with integer coordinates. (Hint: The pixel closest to the midpoint is not far off the line segment.)

5.20 Rewrite the `reduce` method of Problem 5.3 as a recursive method.

5.21 One day you notice that integer multiplication no longer works. Write a recursive procedure to multiply two values a and b using only addition. What is the complexity of this function?

5.22 Modify the "stamp change" problem of Section 5.2.1 to report the number of each type of stamp to be found in the minimum stamp change.

5.23 Prove that $5^n - 4n - 1$ is divisible by 16 for all $n \geq 0$.

5.24 Prove Observation 5.3, that $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ for $n \geq 0$.

5.25 Prove that a function n^c is $O(n^d)$ for any $d \geq c$.

5.26 Prove that $\sum_{i=1}^{n} 2i = n(n+1)$.

5.27 Prove that $\sum_{i=1}^{n} (2i-1) = n^2$.

5.28 Show that for $c \geq 2$ and $n \geq 0$, $\sum_{i=0}^{n} c^i = \frac{c^{n+1}+(c-2)}{c-1} - 1$.

5.29 Prove that $\sum_{i=1}^{n} \log i \leq n \log n$.

5.30 Some artists seek asymmetry. Physicists tell us the universe doesn't always appear symmetric. Why are we unfazed?

5.31 With a colleague, implement a fresh version of `Lists`. First, agree on the types and names of private fields. Then, going down the list of methods required by the `List` interface, split methods to be implemented between you by assigning every other method to your colleague. Bring the code together and compile it. What types of bugs occur? Did you depend on your colleague's code?

5.32 Consider the implementation of a `Ratio` data type. How does symmetry appear in this implementation?

5.33 In the `Vector` class, we extend by doubling, but we never discuss reducing by a similar technique. What is a good strategy?

5.34 Consider the following Java method:

```
static public int fido(int n)
// pre: n is a positive integer
// post: result is the nth term from the sequence
//     1, 3, 7, 15, 31, 63, 127, ...
{
    int result = 1;
    if (n > 1) result = 1+fido(n-1)+fido(n-1);
    // assertion: the above if condition was tested
    //    fido(n) times while computing result
    return result;
}
```

a. What does it compute?

b. Prove or disprove the informal assertion following the `if` statement.

c. What is the time complexity of the method?

d. Why is `fido` an appropriate name for this method?

5.5 Laboratory: How Fast Is Java?

Objective. To develop an appreciation for the speed of basic Java operations including assignment of value to variables, arrays, and `Vectors`.

Discussion. How long does it take to add two integers in Java? How long does it take to assign a value to an entry in an array? The answers to these questions depend heavily on the type of environment a programmer is using and yet play an important role in evaluating the trade-offs in performance between different implementations of data structures.

If we are interested in estimating the time associated with an operation, it is difficult to measure it accurately with clocks available on most modern machines. If an operation takes 100 ns (*nanoseconds*, or billionths of a second), 10,000 of these operations can be performed within a single millisecond clock tick. It is unlikely that we would see a change in the millisecond clock while the operation is being performed.

One approach is to measure, say, the time it takes to perform a million of these operations, and divide that portion of the time associated with the operation by a million. The result *can* be a very accurate measurement of the time it takes to perform the operation. Several important things must be kept in mind:

- Different runs of the experiment can generate different times. This variation is unlikely to be due to significant differences in the speed of the operation, but instead to various interruptions that regularly occur when a program is running. Instead of computing the *average* of the running times, it is best to compute the *minimum* of the experiment's elapsed times. It's unlikely that this is much of an underestimate!

- Never perform input or output while you are timing an experiment. These operations are very expensive and variable. When reading or writing, make sure these operations appear before or after the experiment being timed.

- On modern systems there are many things happening concurrently with your program. Clocks tick forward, printer queues manage printers, network cards are accepting viruses. If you can keep your total experiment time below, say, a tenth of a second, it is likely that you will eliminate many of these distractions.

- The process of repeating an operation takes time. One of our tasks will be to measure the time it takes to execute an empty `for` loop. The loop, of course, is not really empty: it performs a test at the top of the loop and an increment at the bottom. Failing to account for the overhead of a `for` loop makes it impossible to measure any operation that is significantly faster.

- Good compilers can recognize certain operations that can be performed more efficiently in a different way. For example, traditional computers

can assign a value of 0 much faster than the assignment of a value of 42. If an experiment yields an unexpectedly short operation time, change the Java to obscure any easy optimizations that may be performed. Don't forget to subtract the overhead of these obscuring operations!

Keeping a mindful eye on your experimental data will allow you to effectively measure very, very short events accurate to nanoseconds. In one nanosecond, light travels 11.80 inches!

Procedure. The ultimate goal of this experiment is a formally written lab report presenting your results. Carefully design your experiment, and be prepared to defend your approach. The data you collect here is experimental, and necessarily involves error. To reduce the errors described above, perform multiple runs of each experiment, and carefully document your findings. Your report should include results from the following experiments:

1. A description of the machine you are using. Make sure you use this machine for all of your experiments.

2. Write a short program to measure the time that elapses, say, when an empty `for` loop counts to one million. Print out the elapsed time, as well as the per-iteration elapsed time. Adjust the number of loops so that the total elapsed time falls between, say, one-hundredth and one-tenth of a second.

   Recall that we can measure times in nanoseconds (as accurately as possible, given your machine) using `System.nanoTime()`:

   ```
   int i, loops;
   double speed;
   loops = 10000000;
   long start,stop,duration;

   start = System.nanoTime();
   for (i = 0; i < loops; i++)
   {
       // code to be timed goes here
   }
   stop = System.nanoTime();

   duration = stop-start;
   System.out.println("# Elapsed time: "+duration+"ns");
   System.out.println("# Mean time: "+
                      (((double)duration)/loops)+
                      "nanoseconds");
   ```

3. Measure the time it takes to do a single integer assignment (e.g., `i=42;`). Do not forget to subtract the time associated with executing the `for` loop.

4. Measure the time it takes to assign an integer to an array entry. Make sure that the array has been allocated *before* starting the timing loop.

5. Measure the time it takes to assign a `String` reference to an array.

6. Measure the length of time it takes to assign a `String` to a `Vector`. (Note that it is not possible to directly assign an `int` to a `Vector` class.)

7. Copy one `Vector` to another, manually, using `set`. Carefully watch the elapsed time and do not include the time it takes to construct the two `Vectors`! Measure the time it takes to perform the copy for `Vectors` of different lengths. Does this appear to grow linearly?

Formally present your results in a write-up of your experiments.

Thought Questions. Consider the following questions as you complete the lab:

1. Your Java compiler and environment may have several switches that affect the performance of your program. For example, some environments allow the use of *just-in-time* (jit) compilers, that compile frequently used pieces of code (like your timing loop) into machine-specific instructions that are likely to execute faster. How does this affect your experiment?

2. How might you automatically guarantee that your total experiment time lasts between, say, 10 and 100 milliseconds?

3. It is, of course, possible for a timer to *underestimate* the running time of an instruction. For example, if you time a single assignment, it is certainly possible to get an elapsed time of 0—an impossibility. To what extent would a timing underestimate affect your results?

Notes: