CHAPTER 3

The Running Time of Programs

In Chapter 2, we saw two radically different algorithms for sorting: selection sort and merge sort. There are, in fact, scores of algorithms for sorting. This situation is typical: every problem that can be solved at all can be solved by more than one algorithm.

How, then, should we choose an algorithm to solve a given problem? As a general rule, we should always pick an algorithm that is easy to understand, implement, and document. When performance is important, as it often is, we also need to choose an algorithm that runs quickly and uses the available computing resources efficiently. We are thus led to consider the often subtle matter of how we can measure the running time of a program or an algorithm, and what steps we can take to make a program run faster.

3.1 What This Chapter Is About

In this chapter we shall cover the following topics:

- The important performance measures for programs
- Methods for evaluating program performance
- "Big-oh" notation
- Estimating the running time of programs using the big-oh notation
- Using recurrence relations to evaluate the running time of recursive programs

The big-oh notation introduced in Sections 3.4 and 3.5 simplifies the process of estimating the running time of programs by allowing us to avoid dealing with constants that are almost impossible to determine, such as the number of machine instructions that will be generated by a typical C compiler for a given source program.

We introduce the techniques needed to estimate the running time of programs in stages. In Sections 3.6 and 3.7 we present the methods used to analyze programs with no function calls. Section 3.8 extends our capability to programs with calls to nonrecursive functions. Then in Sections 3.9 and 3.10 we show how to deal with recursive functions. Finally, Section 3.11 discusses solutions to recurrence relations, which are inductive definitions of functions that arise when we analyze the running time of recursive functions.

3.2 Choosing an Algorithm

If you need to write a program that will be used once on small amounts of data and then discarded, then you should select the easiest-to-implement algorithm you know, get the program written and debugged, and move on to something else. However, when you need to write a program that is to be used and maintained by many people over a long period of time, other issues arise. One is the understandability, or simplicity, of the underlying algorithm. Simple algorithms are desirable for several reasons. Perhaps most important, a simple algorithm is easier to implement correctly than a complex one. The resulting program is also less likely to have subtle bugs that get exposed when the program encounters an unexpected input after it has been in use for a substantial period of time.

Simplicity

Clarity

Programs should be written clearly and documented carefully so that they can be maintained by others. If an algorithm is simple and understandable, it is easier to describe. With good documentation, modifications to the original program can readily be done by someone other than the original writer (who frequently will not be available to do them), or even by the original writer if the program was done some time earlier. There are numerous stories of programmers who wrote efficient and clever algorithms, then left the company, only to have their algorithms ripped out and replaced by something slower but more understandable by subsequent maintainers of the code.

Efficiency

When a program is to be run repeatedly, its efficiency and that of its underlying algorithm become important. Generally, we associate efficiency with the time it takes a program to run, although there are other resources that a program sometimes must conserve, such as

1. The amount of storage space taken by its variables.
2. The amount of traffic it generates on a network of computers.
3. The amount of data that must be moved to and from disks.

For large problems, however, it is the running time that determines whether a given program can be used, and running time is the main topic of this chapter. We shall, in fact, take the efficiency of a program to mean the amount of time it takes, measured as a function of the size of its input.

Often, understandability and efficiency are conflicting aims. For example, the reader who compares the selection sort program of Fig. 2.3 with the merge sort program of Fig. 2.32 will surely agree that the latter is not only longer, but quite a bit harder to understand. That would still be true even if we summarized the explanation given in Sections 2.2 and 2.8 by placing well-thought-out comments in the programs. As we shall learn, however, merge sort is much more efficient than selection sort, as long as the number of elements to be sorted is a hundred or more. Unfortunately, this situation is quite typical: algorithms that are efficient for large amounts of data tend to be more complex to write and understand than are the relatively inefficient algorithms.

The understandability, or simplicity, of an algorithm is somewhat subjective. We can overcome lack of simplicity in an algorithm, to a certain extent, by explaining the algorithm well in comments and program documentation. The documentor should always consider the person who reads the code and its comments: Is a reasonably intelligent person likely to understand what is being said, or are further explanation, details, definitions, and examples needed?

On the other hand, program efficiency is an objective matter: a program takes what time it takes, and there is no room for dispute. Unfortunately, we cannot run the program on all possible inputs — which are typically infinite in number. Thus, we are forced to make measures of the running time of a program that summarize the program's performance on all inputs, usually as a single expression such as "n^2." How we can do so is the subject matter of the balance of this chapter.

3.3 Measuring Running Time

Once we have agreed that we can evaluate a program by measuring its running time, we face the problem of determining what the running time actually is. The two principal approaches to summarizing the running time are

1. Benchmarking
2. Analysis

We shall consider each in turn, but the primary emphasis of this chapter is on the techniques for analyzing a program or an algorithm.

Benchmarking

When comparing two or more programs designed to do the same set of tasks, it is customary to develop a small collection of typical inputs that can serve as *benchmarks.* That is, we agree to accept the benchmark inputs as representative of the job mix; a program that performs well on the benchmark inputs is assumed to perform well on all inputs.

For example, a benchmark to evaluate sorting programs might contain one small set of numbers, such as the first 20 digits of π; one medium set, such as the set of zip codes in Texas; and one large set, such as the set of phone numbers in the Brooklyn telephone directory. We might also want to check that a program works efficiently (and correctly) when given an empty set of elements to sort, a singleton set, and a list that is already in sorted order. Interestingly, some sorting algorithms perform poorly when given a list of elements that is already sorted.[1]

[1] Neither selection sort nor merge sort is among these; they take approximately the same time on a sorted list as they would on any other list of the same length.

The 90-10 Rule

In conjunction with benchmarking, it is often useful to determine where the program under consideration is spending its time. This method of evaluating program performance is called *profiling* and most programming environments have tools called *profilers* that associate with each statement of a program a number that represents the fraction of the total time taken executing that particular statement. A related utility, called a *statement counter,* is a tool that determines for each statement of a source program the number of times that statement is executed on a given set of inputs.

Profiling

Many programs exhibit the property that most of their running time is spent in a small fraction of the source code. There is an informal rule that states 90% of the running time is spent in 10% of the code. While the exact percentage varies from program to program, the "90-10 rule" says that most programs exhibit significant locality in where the running time is spent. One of the easiest ways to speed up a program is to profile it and then apply code improvements to its "hot spots," which are the portions of the program in which most of the time is spent. For example, we mentioned in Chapter 2 that one might speed up a program by replacing a recursive function with an equivalent iterative one. However, it makes sense to do so only if the recursive function occurs in those parts of the program where most of the time is being spent.

Locality and hot spots

As an extreme case, even if we reduce to zero the time taken by the 90% of the code in which only 10% of the time is spent, we will have reduced the overall running time of the program by only 10%. On the other hand, cutting in half the running time of the 10% of the program where 90% of the time is spent reduces the overall running time by 45%.

Analysis of a Program

To analyze a program, we begin by grouping inputs according to size. What we choose to call the size of an input can vary from program to program, as we discussed in Section 2.9 in connection with proving properties of recursive programs. For a sorting program, a good measure of the size is the number of elements to be sorted. For a program that solves n linear equations in n unknowns, it is normal to take n to be the size of the problem. Other programs might use the value of some particular input, or the length of a list that is an input to the program, or the size of an array that is an input, or some combination of quantities such as these.

Running Time

It is convenient to use a function $T(n)$ to represent the number of units of time taken by a program or an algorithm on any input of size n. We shall call $T(n)$ the *running time* of the program. For example, a program may have a running time $T(n) = cn$, where c is some constant. Put another way, the running time of this program is linearly proportional to the size of the input on which it is run. Such a program or algorithm is said to be *linear time,* or just *linear.*

Linear-time algorithm

We can think of the running time $T(n)$ as the number of C statements executed by the program or as the length of time taken to run the program on some standard computer. Most of the time we shall leave the units of $T(n)$ unspecified. In fact, as we shall see in the next section, it makes sense to talk of the running time of a program only as some (unknown) constant factor times $T(n)$.

Quite often, the running time of a program depends on a particular input, not just on the size of the input. In these cases, we define $T(n)$ to be the *worst-case* running time, that is, the maximum running time on any input among all inputs of size n.

Worst and average-case running time

Another common performance measure is $T_{avg}(n)$, the average running time of the program over all inputs of size n. The average running time is sometimes a more realistic measure of what performance one will see in practice, but it is often much harder to compute than the worst-case running time. The notion of an "average" running time also implies that all inputs of size n are equally likely, which may or may not be true in a given situation.

✦ **Example 3.1.** Let us estimate the running time of the `SelectionSort` fragment shown in Fig. 3.1. The statements have the original line numbers from Fig. 2.2. The purpose of the code is to set `small` to the index of the smallest of the elements found in the portion of the array `A` from `A[i]` through `A[n-1]`.

```
(2)         small = i;
(3)         for(j = i+1; j < n; j++)
(4)             if (A[j] < A[small])
(5)                 small = j;
```

Fig. 3.1. Inner loop of selection sort.

To begin, we need to develop a simple notion of time units. We shall examine the issue in detail later, but for the moment, the following simple scheme is sufficient. We shall count one time unit each time we execute an assignment statement. At line (3), we count one unit for initializing `j` at the beginning of the for-loop, one unit for testing whether $j < n$, and one unit for incrementing `j`, each time we go around the loop. Finally, we charge one unit each time we perform the test of line (4).

First, let us consider the body of the inner loop, lines (4) and (5). The test of line (4) is always executed, but the assignment at line (5) is executed only if the test succeeds. Thus, the body takes either 1 or 2 time units, depending on the data in array `A`. If we want to take the worst case, then we can assume that the body takes 2 units. We go around the for-loop $n - i - 1$ times, and each time around we execute the body (2 units), then increment `j` and test whether $j < n$ (another 2 units). Thus, the number of time units spent going around the loop is $4(n - i - 1)$. To this number, we must add 1 for initializing `small` at line (2), 1 for initializing `j` at line (3), and 1 for the first test $j < n$ at line (3), which is not associated with the end of any iteration of the loop. Hence, the total time taken by the program fragment in Fig. 3.1 is $4(n - i) - 1$.

It is natural to regard the "size" m of the data on which Fig. 3.1 operates as $m = n - i$, since that is the length of the array `A[i..n-1]` on which it operates. Then the running time, which is $4(n - i) - 1$, equals $4m - 1$. Thus, the running time $T(m)$ for Fig. 3.1 is $4m - 1$. ✦

Comparing Different Running Times

Suppose that for some problem we have the choice of using a linear-time program A whose running time is $T_A(n) = 100n$ and a quadratic-time program B whose running time is $T_B(n) = 2n^2$. Let us suppose that both these running times are the number of milliseconds taken on a particular computer on an input of size n.[2] The graphs of the running times are shown in Fig. 3.2.

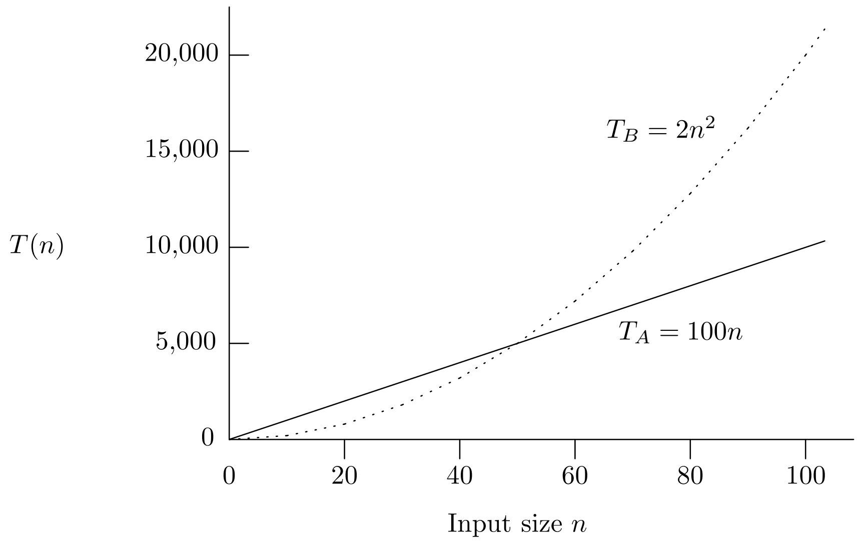


Fig. 3.2. Running times of a linear and a quadratic program.

From Fig. 3.2 we see that for inputs of size less than 50, program B is faster than program A. When the input becomes larger than 50, program A becomes faster, and from that point on, the larger the input, the bigger the advantage A has over B. For inputs of size 100, A is twice as fast as B, and for inputs of size 1000, A is 20 times as fast.

The functional form of a program's running time ultimately determines how big a problem we can solve with that program. As the speed of computers increases, we get bigger improvements in the sizes of problems that we can solve with programs whose running times grow slowly than with programs whose running times rise rapidly.

Again, assuming that the running times of the programs shown in Fig. 3.2 are in milliseconds, the table in Fig. 3.3 indicates how large a problem we can solve with each program on the same computer in various amounts of time given in seconds. For example, suppose we can afford 100 seconds of computer time. If computers become 10 times as fast, then in 100 seconds we can handle problems of the size that used to require 1000 seconds. With algorithm A, we can now solve problems 10 times as large, but with algorithm B we can only solve problems about 3 times as large. Thus, as computers continue to get faster, we gain an even more significant advantage by using algorithms and programs with lower growth rates.

[2] This situation is not too dissimilar to the situation where algorithm A is merge sort and algorithm B is selection sort. However, the running time of merge sort grows as $n \log n$, as we shall see in Section 3.10.

Never Mind Algorithm Efficiency; Just Wait a Few Years

Frequently, one hears the argument that there is no need to improve the running time of algorithms or to select efficient algorithms, because computer speeds are doubling every few years and it will not be long before any algorithm, however inefficient, will take so little time that one will not care. People have made this claim for many decades, yet there is no limit in sight to the demand for computational resources. Thus, we generally reject the view that hardware improvements will make the study of efficient algorithms superfluous.

There are situations, however, when we need not be overly concerned with efficiency. For example, a school may, at the end of each term, transcribe grades reported on electronically readable grade sheets to student transcripts, all of which are stored in a computer. The time this operation takes is probably linear in the number of grades reported, like the hypothetical algorithm A. If the school replaces its computer by one 10 times as fast, it can do the job in one-tenth the time. It is very unlikely, however, that the school will therefore enroll 10 times as many students, or require each student to take 10 times as many classes. The computer speedup will not affect the size of the input to the transcript program, because that size is limited by other factors.

On the other hand, there are some problems that we are beginning to find approachable with emerging computing resources, but whose "size" is too great to handle with existing technology. Some of these problems are natural language understanding, computer vision (understanding of digitized pictures), and "intelligent" interaction between computers and humans in all sorts of endeavors. Speedups, either through improved algorithms or from machine improvements, will enhance our ability to deal with these problems in the coming years. Moreover, when they become "simple" problems, a new generation of challenges, which we can now only barely imagine, will take their place on the frontier of what it is possible to do with computers.

Time sec.	Maximum Problem Size Solvable with Program A	Maximum Problem Size Solvable with Program B
1	10	22
10	100	70
100	1000	223
1000	10000	707

Fig. 3.3. Problem size as a function of available time.

EXERCISES

3.3.1: Consider the factorial program fragment in Fig. 2.13, and let the input size be the value of n that is read. Counting one time unit for each assignment, read, and write statement, and one unit each time the condition of the while-statement is tested, compute the running time of the program.

3.3.2: For the program fragments of (a) Exercise 2.5.1 and (b) Fig. 2.14, give an appropriate size for the input. Using the counting rules of Exercise 3.3.1, determine the running times of the programs.

3.3.3: Suppose program A takes $2^n/1000$ units of time and program B takes $1000n^2$ units. For what values of n does program A take less time than program B?

3.3.4: For each of the programs of Exercise 3.3.3, how large a problem can be solved in (a) 10^6 time units, (b) 10^9 time units, and (c) 10^{12} time units?

3.3.5: Repeat Exercises 3.3.3 and 3.3.4 if program A takes $1000n^4$ time units and program B takes n^{10} time units.

3.4 Big-Oh and Approximate Running Time

Suppose we have written a C program and have selected the particular input on which we would like it to run. The running time of the program on this input still depends on two factors:

1. The computer on which the program is run. Some computers execute instructions more rapidly than others; the ratio between the speeds of the fastest supercomputers and the slowest personal computers is well over 1000 to 1.

2. The particular C compiler used to generate a program for the computer to execute. Different programs can take different amounts of time to execute on the same machine, even though the programs have the same effect.

As a result, we cannot look at a C program and its input and say, "This task will take 3.21 seconds," unless we know which machine and which compiler will be used. Moreover, even if we know the program, the input, the machine, and the compiler, it is usually far too complex a task to predict exactly the number of machine instructions that will be executed.

For these reasons, we usually express the running time of a program using "big-oh" notation, which is designed to let us hide constant factors such as

1. The average number of machine instructions a particular compiler generates.

2. The average number of machine instructions a particular machine executes per second.

For example, instead of saying, as we did in Example 3.1, that the `SelectionSort` fragment we studied takes time $4m - 1$ on an array of length m, we would say that it takes $O(m)$ time, which is read "big-oh of m" or just "oh of m," and which informally means "some constant times m."

The notion of "some constant times m" not only allows us to ignore unknown constants associated with the compiler and the machine, but also allows us to make some simplifying assumptions. In Example 3.1, for instance, we assumed that all assignment statements take the same amount of time, and that this amount of time was also taken by the test for termination in the for-loop, the incrementation of `j` around the loop, the initialization, and so on. Since none of these assumptions is valid in practice, the constants 4 and -1 in the running-time formula $T(m) = 4m-1$ are at best approximations to the truth. It would be more appropriate to describe $T(m)$ as "some constant times m, plus or minus another constant" or even as "at most proportional to m." The notation $O(m)$ enables us to make these statements without getting involved in unknowable or meaningless constants.

On the other hand, representing the running time of the fragment as $O(m)$ does tell us something very important. It says that the time to execute the fragment on progressively larger arrays grows linearly, like the hypothetical Program A of Figs. 3.2 and 3.3 discussed at the end of Section 3.3. Thus, the algorithm embodied by this fragment will be superior to competing algorithms whose running time grows faster, such as the hypothetical Program B of that discussion.

Definition of Big-Oh

We shall now give a formal definition of the notion of one function being "big-oh" of another. Let $T(n)$ be a function, which typically is the running time of some program, measured as a function of the input size n. As befits a function that measures the running time of a program, we shall assume that

1. The argument n is restricted to be a nonnegative integer, and

2. The value $T(n)$ is nonnegative for all arguments n.

Let $f(n)$ be some function defined on the nonnegative integers n. We say that

"$T(n)$ is $O\big(f(n)\big)$"

if $T(n)$ is at most a constant times $f(n)$, except possibly for some small values of n. Formally, we say that $T(n)$ is $O\big(f(n)\big)$ if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$, we have $T(n) \leq cf(n)$.

Witnesses

We call the pair n_0 and c *witnesses* to the fact that $T(n)$ is $O\big(f(n)\big)$. The witnesses "testify" to the big-oh relationship of $T(n)$ and $f(n)$ in a form of proof that we shall next demonstrate.

Proving Big-Oh Relationships

We can apply the definition of "big-oh" to prove that $T(n)$ is $O\big(f(n)\big)$ for particular functions T and f. We do so by exhibiting a particular choice of witnesses n_0 and c and then proving that $T(n) \leq cf(n)$. The proof must assume only that n is a nonnegative integer and that n is at least as large as our chosen n_0. Usually, the proof involves algebra and manipulation of inequalities.

Quadratic running time

✦ **Example 3.2.** Suppose we have a program whose running time is $T(0) = 1$, $T(1) = 4$, $T(2) = 9$, and in general $T(n) = (n+1)^2$. We can say that $T(n)$ is $O(n^2)$, or that $T(n)$ is *quadratic*, because we can choose witnesses $n_0 = 1$ and $c = 4$. We then need to prove that $(n+1)^2 \leq 4n^2$, provided $n \geq 1$. In proof, expand $(n+1)^2$ as $n^2 + 2n + 1$. As long as $n \geq 1$, we know that $n \leq n^2$ and $1 \leq n^2$. Thus

$$n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2$$

Template for Big-Oh Proofs

Remember: all big-oh proofs follow essentially the same form. Only the algebraic manipulation varies. We need to do only two things to have a proof that $T(n)$ is $O\big(f(n)\big)$.

1. State the witnesses n_0 and c. These witnesses must be specific constants, e.g., $n_0 = 47$ and $c = 12.5$. Also, n_0 must be a nonnegative integer, and c must be a positive real number.
2. By appropriate algebraic manipulation, show that if $n \geq n_0$ then $T(n) \leq cf(n)$, for the particular witnesses n_0 and c chosen.

Alternatively, we could pick witnesses $n_0 = 3$ and $c = 2$, because, as the reader may check, $(n+1)^2 \leq 2n^2$, for all $n \geq 3$.

However, we cannot pick $n_0 = 0$ with any c, because with $n = 0$, we would have to show that $(0+1)^2 \leq c0^2$, that is, that 1 is less than or equal to c times 0. Since $c \times 0 = 0$ no matter what c we pick, and $1 \leq 0$ is false, we are doomed if we pick $n_0 = 0$. That doesn't matter, however, because in order to show that $(n+1)^2$ is $O(n^2)$, we had only to find one choice of witnesses n_0 and c that works. ✦

It may seem odd that although $(n+1)^2$ is larger than n^2, we can still say that $(n+1)^2$ is $O(n^2)$. In fact, we can also say that $(n+1)^2$ is big-oh of any fraction of n^2, for example, $O(n^2/100)$. To see why, choose witnesses $n_0 = 1$ and $c = 400$. Then if $n \geq 1$, we know that

$$(n+1)^2 \leq 400(n^2/100) = 4n^2$$

by the same reasoning as was used in Example 3.2. The general principles underlying these observations are that

1. *Constant factors don't matter.* For any positive constant d and any function $T(n)$, $T(n)$ is $O\big(dT(n)\big)$, regardless of whether d is a large number or a very small fraction, as long as $d > 0$. To see why, choose witnesses $n_0 = 0$ and $c = 1/d$.[3] Then $T(n) \leq c\big(dT(n)\big)$, since $cd = 1$. Likewise, if we know that $T(n)$ is $O\big(f(n)\big)$, then we also know that $T(n)$ is $O\big(df(n)\big)$ for any $d > 0$, even a very small d. The reason is that we know that $T(n) \leq c_1 f(n)$ for some constant c_1 and all $n \geq n_0$. If we choose $c = c_1/d$, we can see that $T(n) \leq c\big(df(n)\big)$ for $n \geq n_0$.
2. *Low-order terms don't matter.* Suppose $T(n)$ is a polynomial of the form

   $$a_k n^k + a_{k-1}n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

   where the leading coefficient, a_k, is positive. Then we can throw away all terms but the first (the term with the highest exponent, k) and, by rule (1), ignore the constant a_k, replacing it by 1. That is, we can conclude $T(n)$ is $O(n^k)$. In proof, let $n_0 = 1$, and let c be the sum of all the positive coefficients among the a_i's, $0 \leq i \leq k$. If a coefficient a_j is 0 or negative, then surely $a_j n^j \leq 0$. If

[3] Note that although we are required to choose constants as witnesses, not functions, there is nothing wrong with choosing $c = 1/d$, because d itself is some constant.

Fallacious Arguments About Big-Oh

The definition of "big-oh" is tricky, in that it requires us, after examining $T(n)$ and $f(n)$, to pick witnesses n_0 and c once and for all, and then to show that $T(n) \leq cf(n)$ for all $n \geq n_0$. It is not permitted to pick c and/or n_0 anew for each value of n. For example, one occasionally sees the following fallacious "proof" that n^2 is $O(n)$. "Pick $n_0 = 0$, and for each n, pick $c = n$. Then $n^2 \leq cn$." This argument is invalid, because we are required to pick c once and for all, without knowing n.

a_j is positive, then $a_j n^j \leq a_j n^k$, for all $j < k$, as long as $n \geq 1$. Thus, $T(n)$ is no greater than n^k times the sum of the positive coefficients, or cn^k.

✦ **Example 3.3.** As an example of rule (1) ("constants don't matter"), we can see that $2n^3$ is $O(.001n^3)$. Let $n_0 = 0$ and $c = 2/.001 = 2000$. Then clearly $2n^3 \leq 2000(.001n^3) = 2n^3$, for all $n \geq 0$.

As an example of rule (2) ("low order terms don't matter"), consider the polynomial

$$T(n) = 3n^5 + 10n^4 - 4n^3 + n + 1$$

The highest-order term is n^5, and we claim that $T(n)$ is $O(n^5)$. To check the claim, let $n_0 = 1$ and let c be the sum of the positive coefficients. The terms with positive coefficients are those with exponents 5, 4, 1, and 0, whose coefficients are, respectively, 3, 10, 1, and 1. Thus, we let $c = 15$. We claim that for $n \geq 1$,

$$3n^5 + 10n^4 - 4n^3 + n + 1 \leq 3n^5 + 10n^5 + n^5 + n^5 = 15n^5 \tag{3.1}$$

We can check that inequality (3.1) holds by matching the positive terms; that is, $3n^5 \leq 3n^5$, $10n^4 \leq 10n^5$, $n \leq n^5$, and $1 \leq n^5$. Also, the negative term on the left side of (3.1) can be ignored, since $-4n^3 \leq 0$. Thus, the left side of (3.1), which is $T(n)$, is less than or equal to the right side, which is $15n^5$, or cn^5. We can conclude that $T(n)$ is $O(n^5)$.

In fact, the principle that low-order terms can be deleted applies not only to polynomials, but to any sum of expressions. That is, if the ratio $h(n)/g(n)$ approaches 0 as n approaches infinity, then $h(n)$ "grows more slowly" than $g(n)$, or "has a lower growth rate" than $g(n)$, and we may neglect $h(n)$. That is, $g(n)+h(n)$ is $O\big(g(n)\big)$.

Growth rate

For example, let $T(n) = 2^n + n^3$. It is known that every polynomial, such as n^3, grows more slowly than every exponential, such as 2^n. Since $n^3/2^n$ approaches 0 as n increases, we can throw away the lower-order term and conclude that $T(n)$ is $O(2^n)$.

To prove formally that $2^n + n^3$ is $O(2^n)$, let $n_0 = 10$ and $c = 2$. We must show that for $n \geq 10$, we have

$$2^n + n^3 \leq 2 \times 2^n$$

If we subtract 2^n from both sides, we see it is sufficient to show that for $n \geq 10$, it is the case that $n^3 \leq 2^n$.

For $n = 10$ we have $2^{10} = 1024$ and $10^3 = 1000$, and so $n^3 \leq 2^n$ for $n = 10$. Each time we add 1 to n, 2^n doubles, while n^3 is multiplied by a quantity $(n+1)^3/n^3$

that is less than 2 when $n \geq 10$. Thus, as n increases beyond 10, n^3 becomes progressively less than 2^n. We conclude that $n^3 \leq 2^n$ for $n \geq 10$, and thus that $2^n + n^3$ is $O(2^n)$. ✦

Proofs That a Big-Oh Relationship Does Not Hold

If a big-oh relationship holds between two functions, then we can prove it by finding witnesses. However, what if some function $T(n)$ is *not* big-oh of some other function $f(n)$? Can we ever hope to be sure that there is not such a big-oh relationship? The answer is that quite frequently we can prove a particular function $T(n)$ is not $O\bigl(f(n)\bigr)$. The method of proof is to assume that witnesses n_0 and c exist, and derive a contradiction. The following is an example of such a proof.

✦ **Example 3.4.** In the box on "Fallacious Arguments About Big-Oh," we claimed that n^2 is not $O(n)$. We can show this claim as follows. Suppose it were. Then there would be witnesses n_0 and c such that $n^2 \leq cn$ for all $n \geq n_0$. But if we pick n_1 equal to the larger of $2c$ and n_0, then the inequality

$$(n_1)^2 \leq cn_1 \tag{3.2}$$

must hold (because $n_1 \geq n_0$ and $n^2 \leq cn$ allegedly holds for all $n \geq n_0$).

If we divide both sides of (3.2) by n_1, we have $n_1 \leq c$. However, we also chose n_1 to be at least $2c$. Since witness c must be positive, n_1 cannot be both less than c and greater than $2c$. Thus, witnesses n_0 and c that would show n^2 to be $O(n)$ do not exist, and we conclude that n^2 is not $O(n)$. ✦

EXERCISES

3.4.1: Consider the four functions

f_1: n^2
f_2: n^3
f_3: n^2 if n is odd, and n^3 if n is even
f_4: n^2 if n is prime, and n^3 if n is composite

For each i and j equal to 1, 2, 3, 4, determine whether $f_i(n)$ is $O\bigl(f_j(n)\bigr)$. Either give values n_0 and c that prove the big-oh relationship, or assume that there are such values n_0 and c, and then derive a contradiction to prove that $f_i(n)$ is not $O\bigl(f_j(n)\bigr)$. *Hint*: Remember that all primes except 2 are odd. Also remember that there are an infinite number of primes and an infinite number of composite numbers (nonprimes).

3.4.2: Following are some big-oh relationships. For each, give witnesses n_0 and c that can be used to prove the relationship. Choose your witnesses to be minimal, in the sense that $n_0 - 1$ and c are not witnesses, and if $d < c$, then n_0 and d are not witnesses.

a) n^2 is $O(.001n^3)$
b) $25n^4 - 19n^3 + 13n^2 - 106n + 77$ is $O(n^4)$
c) 2^{n+10} is $O(2^n)$
d) n^{10} is $O(3^n)$
e)* $\log_2 n$ is $O(\sqrt{n})$

Template for Proofs That a Big-Oh Relationship Is False

The following is an outline of typical proofs that a function $T(n)$ is not $O\big(f(n)\big)$. Example 3.4 illustrates such a proof.

1. Start by supposing that there were witnesses n_0 and c such that for all $n \geq n_0$, we have $f(n) \leq cg(n)$. Here, n_0 and c are symbols standing for unknown witnesses.
2. Define a particular integer n_1, expressed in terms of n_0 and c (for example, $n_1 = \max(n_0, 2c)$ was chosen in Example 3.4). This n_1 will be the value of n for which we show $T(n_1) \leq cf(n_1)$ is false.
3. Show that for the chosen n_1 we have $n_1 \geq n_0$. This part can be very easy, since we may choose n_1 to be at least n_0 in step (2).
4. Claim that because $n_1 \geq n_0$, we must have $T(n_1) \leq cf(n_1)$.
5. Derive a contradiction by showing that for the chosen n_1 we have $T(n_1) > cf(n_1)$. Choosing n_1 in terms of c can make this part easy, as it was in Example 3.4.

3.4.3*: Prove that if $f(n) \leq g(n)$ for all n, then $f(n) + g(n)$ is $O\big(g(n)\big)$.

3.4.4**: Suppose that $f(n)$ is $O\big(g(n)\big)$ and $g(n)$ is $O\big(f(n)\big)$. What can you say about $f(n)$ and $g(n)$? Is it necessarily true that $f(n) = g(n)$? Does the limit $f(n)/g(n)$ as n goes to infinity necessarily exist?

3.5 Simplifying Big-Oh Expressions

As we saw in the previous section, it is possible to simplify big-oh expressions by dropping constant factors and low-order terms. We shall see how important it is to make such simplifications when we analyze programs. It is common for the running time of a program to be attributable to many different statements or program fragments, but it is also normal for a few of these pieces to account for the bulk of the running time (by the "90-10" rule). By dropping low-order terms, and by combining equal or approximately equal terms, we can often greatly simplify the expressions for running time.

The Transitive Law for Big-Oh Expressions

To begin, we shall take up a useful rule for thinking about big-oh expressions. A relationship like $\leq$ is said to be *transitive,* because it obeys the law "if $A \leq B$ and $B \leq C$, then $A \leq C$." For example, since $3 \leq 5$ and $5 \leq 10$, we can be sure that $3 \leq 10$.

The relationship "is big-oh of" is another example of a transitive relationship. That is, if $f(n)$ is $O\big(g(n)\big)$ and $g(n)$ is $O\big(h(n)\big)$, it follows that $f(n)$ is $O\big(h(n)\big)$. To see why, first suppose that $f(n)$ is $O\big(g(n)\big)$. Then there are witnesses n_1 and c_1 such that $f(n) \leq c_1g(n)$ for all $n \geq n_1$. Similarly, if $g(n)$ is $O\big(h(n)\big)$, then there are witnesses n_2 and c_2 such that $g(n) \leq c_2h(n)$ for all $n \geq n_2$.

Polynomial and Exponential Big-Oh Expressions

The *degree* of a polynomial is the highest exponent found among its terms. For example, the degree of the polynomial $T(n)$ mentioned in Examples 3.3 and 3.5 is 5, because $3n^5$ is its highest-order term. From the two principles we have enunciated (constant factors don't matter, and low-order terms don't matter), plus the transitive law for big-oh expressions, we know the following:

1. If $p(n)$ and $q(n)$ are polynomials and the degree of $q(n)$ is as high as or higher than the degree of $p(n)$, then $p(n)$ is $O\big(q(n)\big)$.
2. If the degree of $q(n)$ is lower than the degree of $p(n)$, then $p(n)$ is *not* $O\big(q(n)\big)$.
3. **Exponential** *Exponentials* are expressions of the form a^n for $a > 1$. Every exponential grows faster than every polynomial. That is, we can show for any polynomial $p(n)$ that $p(n)$ is $O(a^n)$. For example, n^5 is $O\big((1.01)^n\big)$.
4. Conversely, no exponential a^n, for $a > 1$, is $O\big(p(n)\big)$ for any polynomial $p(n)$.

Let n_0 be the larger of n_1 and n_2, and let $c = c_1c_2$. We claim that n_0 and c are witnesses to the fact that $f(n)$ is $O\big(h(n)\big)$. For suppose $n \geq n_0$. Since $n_0 = \max(n_1, n_2)$, we know that $n \geq n_1$ and $n \geq n_2$. Therefore, $f(n) \leq c_1g(n)$ and $g(n) \leq c_2h(n)$.

Now substitute $c_2h(n)$ for $g(n)$ in the inequality $f(n) \leq c_1g(n)$, to prove $f(n) \leq c_1c_2h(n)$. This inequality shows $f(n)$ is $O\big(h(n)\big)$.

✦ **Example 3.5.** We know from Example 3.3 that

$$T(n) = 3n^5 + 10n^4 - 4n^3 + n + 1$$

is $O(n^5)$. We also know from the rule that "constant factors don't matter" that n^5 is $O(.01n^5)$. By the transitive law for big-oh, we know that $T(n)$ is $O(.01n^5)$. ✦

Describing the Running Time of a Program

We defined the running time $T(n)$ of a program to be the maximum number of time units taken on any input of size n. We also said that determining a precise formula for $T(n)$ is a difficult, if not impossible, task. Frequently, we can simplify matters considerably by using a big-oh expression $O(f(n))$ as an upper bound on $T(n)$.

For example, an upper bound on the running time $T(n)$ of `SelectionSort` is an^2, for some constant a and any $n \geq 1$; we shall demonstrate this fact in Section 3.6. Then we can say the running time of `SelectionSort` is $O(n^2)$. That statement is intuitively the most useful one to make, because n^2 is a very simple function, and stronger statements about other simple functions, like "$T(n)$ is $O(n)$," are false.

However, because of the nature of big-oh notation, we can also state that the running time $T(n)$ is $O(.01n^2)$, or $O(7n^2-4n+26)$, or in fact big-oh of any quadratic polynomial. The reason is that n^2 is big-oh of any quadratic, and so the transitive law plus the fact that $T(n)$ is $O(n^2)$ tells us $T(n)$ is big-oh of any quadratic.

Worse, n^2 is big-oh of any polynomial of degree 3 or higher, or of any exponential. Thus, by transitivity again, $T(n)$ is $O(n^3)$, $O(2^n + n^4)$, and so on. However, we shall explain why $O(n^2)$ is the preferred way to express the running time of `SelectionSort`.

Tightness

First, we generally want the "tightest" big-oh upper bound we can prove. That is, if $T(n)$ is $O(n^2)$, we want to say so, rather than make the technically true but weaker statement that $T(n)$ is $O(n^3)$. On the other hand, this way lies madness, because if we like $O(n^2)$ as an expression of running time, we should like $O(0.5n^2)$ even better, because it is "tighter," and we should like $O(.01n^2)$ still more, and so on. However, since constant factors don't matter in big-oh expressions, there is really no point in trying to make the estimate of running time "tighter" by shrinking the constant factor. Thus, whenever possible, we try to use a big-oh expression that has a constant factor 1.

Figure 3.4 lists some of the more common running times for programs and their informal names. Note in particular that $O(1)$ is an idiomatic shorthand for "some constant," and we shall use $O(1)$ repeatedly for this purpose.

BIG-OH	INFORMAL NAME
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	n log n
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential

Fig. 3.4. Informal names for some common big-oh running times.

Tight bound

More precisely, we shall say that $f(n)$ is a *tight* big-oh bound on $T(n)$ if

1. $T(n)$ is $O\big(f(n)\big)$, and
2. If $T(n)$ is $O\big(g(n)\big)$, then it is also true that $f(n)$ is $O\big(g(n)\big)$. Informally, we cannot find a function $g(n)$ that grows at least as fast as $T(n)$ but grows slower than $f(n)$.

✦ **Example 3.6.** Let $T(n) = 2n^2 + 3n$ and $f(n) = n^2$. We claim that $f(n)$ is a tight bound on $T(n)$. To see why, suppose $T(n)$ is $O\big(g(n)\big)$. Then there are constants c and n_0 such that for all $n \geq n_0$, we have $T(n) = 2n^2 + 3n \leq cg(n)$. Then $g(n) \geq (2/c)n^2$ for $n \geq n_0$. Since $f(n)$ is n^2, we have $f(n) \leq (c/2)g(n)$ for $n \geq n_0$. Thus, $f(n)$ is $O\big(g(n)\big)$.

On the other hand, $f(n) = n^3$ is not a tight big-oh bound on $T(n)$. Now we can pick $g(n) = n^2$. We have seen that $T(n)$ is $O\big(g(n)\big)$, but we cannot show that $f(n)$ is $O\big(g(n)\big)$, since n^3 is not $O(n^2)$. Thus, n^3 is not a tight big-oh bound on $T(n)$. ✦

Simplicity

Simple function

The other goal in our choice of a big-oh bound is simplicity in the expression of the function. Unlike tightness, simplicity can sometimes be a matter of taste. However, we shall generally regard a function $f(n)$ as *simple* if

1. It is a single term and
2. The coefficient of that term is 1.

❖ **Example 3.7.** The function n^2 is simple; $2n^2$ is not simple because the coefficient is not 1, and $n^2 + n$ is not simple because there are two terms. ❖

Tightness and simplicity may conflict

There are some situations, however, where the tightness of a big-oh upper bound and simplicity of the bound are conflicting goals. The following is an example where the simple bound doesn't tell the whole story. Fortunately, such cases are rare in practice.

```
                int PowersOfTwo(int n)
                {
                    int i;
(1)                 i = 0;
(2)                 while (n%2 == 0) {
(3)                     n = n/2;
(4)                     i++;
                    }
(5)                 return i;
                }
```

Fig. 3.5. Counting factors of 2 in a positive integer n.

❖ **Example 3.8.** Consider the function `PowersOfTwo` in Fig. 3.5, which takes a positive argument n and counts the number times 2 divides n. That is, the test of line (2) asks whether n is even and, if so, removes a factor of 2 at line (3) of the loop body. Also in the loop, we increment `i`, which counts the number of factors we have removed from the original value of `n`.

Let the size of the input be the value of n itself. The body of the while-loop consists of two C assignment statements, lines (3) and (4), and so we can say that the time to execute the body once is $O(1)$, that is, some constant amount of time, independent of n. If the loop is executed m times, then the total time spent going around the loop will be $O(m)$, or some amount of time that is proportional to m. To this quantity we must add $O(1)$, or some constant, for the single executions of lines (1) and (5), plus the first test of the while-condition, which is technically not part of any loop iteration. Thus, the time spent by the program is $O(m) + O(1)$. Following our rule that low-order terms can be neglected, the time is $O(m)$, unless $m = 0$, in which case it is $O(1)$. Put another way, the time spent on input n is proportional to 1 plus the number of times 2 divides n.

Using Big-Oh Notation in Mathematical Expressions

Strictly speaking, the only mathematically correct way to use a big-oh expression is after the word "is," as in "$2n^2$ is $O(n^3)$." However, in Example 3.8, and for the remainder of the chapter, we shall take a liberty and use big-oh expressions as operands of addition and other arithmetic operators, as in $O(n) + O(n^2)$. We should interpret a big-oh expression used this way as meaning "some function that is big-oh of." For example, $O(n) + O(n^2)$ means "the sum of some linear function and some quadratic function." Also, $O(n) + T(n)$ should be interpreted as the sum of some linear function and the particular function $T(n)$.

How many times does 2 divide n? For every odd n, the answer is 0, so the function `PowersOfTwo` takes time $O(1)$ on every odd n. However, when n is a power of 2 — that is, when $n = 2^k$ for some k — 2 divides n exactly k times. When $n = 2^k$, we may take logarithms to base 2 of both sides and conclude that $\log_2 n = k$. That is, m is at most logarithmic in n, or $m = O(\log n)$.[4]

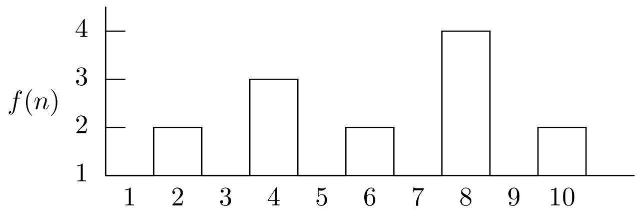


Fig. 3.6. The function $f(n) = m(n) + 1$, where $m(n)$ is the number of times 2 divides n.

We may thus say that the running time of `PowersOfTwo` is $O(\log n)$. This bound meets our definition of simplicity. However, there is another, more precise way of stating an upper bound on the running time of `PowersOfTwo`, which is to say that it is big-oh of the function $f(n) = m(n) + 1$, where $m(n)$ is the number of times 2 divides n. This function is hardly simple, as Fig. 3.6 shows. It oscillates wildly but never goes above $1 + \log_2 n$.

Since the running time of `PowersOfTwo` is $O\big(f(n)\big)$, but $\log n$ is not $O\big(f(n)\big)$, we claim that $\log n$ is not a tight bound on the running time. On the other hand, $f(n)$ is a tight bound, but it is not simple. ✦

The Summation Rule

Suppose a program consists of two parts, one of which takes $O(n^2)$ time and the other of which takes $O(n^3)$ time. We can "add" these two big-oh bounds to get the running time of the entire program. In many cases, such as this one, it is possible to "add" big-oh expressions by making use of the following *summation rule*:

[4] Note that when we speak of logarithms within a big-oh expression, there is no need to specify the base. The reason is that if a and b are bases, then $\log_a n = (\log_b n)(\log_a b)$. Since $\log_a b$ is a constant, we see that $\log_a n$ and $\log_b n$ differ by only a constant factor. Thus, the functions $\log_x n$ to different bases x are big-oh of each other, and we can, by the transitive law, replace within a big-oh expression any $\log_a n$ by $\log_b n$ where b is a base different from a.

Logarithms in Running Times

If you think of logarithms as something having to do with integral calculus ($\log_e a = \int_1^a \frac{1}{x}dx$), you may be surprised to find them appearing in analyses of algorithms. Computer scientists generally think of "$\log n$" as meaning $\log_2 n$, rather than $\log_e n$ or $\log_{10} n$. Notice that $\log_2 n$ is the number of times we have to divide n by 2 to get down to 1, or alternatively, the number of 2's we must multiply together to reach n. You may easily check that $n = 2^k$ is the same as saying $\log_2 n = k$; just take logarithms to the base 2 of both sides.

The function `PowersOfTwo` divides n by 2 as many times as it can, And when n is a power of 2, then the number of times n can be divided by 2 is $\log_2 n$. Logarithms arise quite frequently in the analysis of divide-and-conquer algorithms (e.g., merge sort) that divide their input into two equal, or nearly equal, parts at each stage. If we start with an input of size n, then the number of stages at which we can divide the input in half until pieces are of size 1 is $\log_2 n$, or, if n is not a power of 2, then the smallest integer greater than $\log_2 n$.

Suppose $T_1(n)$ is known to be $O\big(f_1(n)\big)$, while $T_2(n)$ is known to be $O\big(f_2(n)\big)$. Further, suppose that f_2 grows no faster than f_1; that is, $f_2(n)$ is $O\big(f_1(n)\big)$. Then we can conclude that $T_1(n) + T_2(n)$ is $O\big(f_1(n)\big)$.

In proof, we know that there are constants n_1, n_2, n_3, c_1, c_2, and c_3 such that

1. If $n \geq n_1$, then $T_1(n) \leq c_1 f_1(n)$.
2. If $n \geq n_2$, then $T_2(n) \leq c_2 f_2(n)$.
3. If $n \geq n_3$, then $f_2(n) \leq c_3 f_1(n)$.

Let n_0 be the largest of n_1, n_2, and n_3, so that (1), (2), and (3) hold when $n \geq n_0$. Thus, for $n \geq n_0$, we have

$$T_1(n) + T_2(n) \leq c_1 f_1(n) + c_2 f_2(n)$$

If we use (3) to provide an upper bound on $f_2(n)$, we can get rid of $f_2(n)$ altogether and conclude that

$$T_1(n) + T_2(n) \leq c_1 f_1(n) + c_2 c_3 f_1(n)$$

Therefore, for all $n \geq n_0$ we have

$$T_1(n) + T_2(n) \leq c f_1(n)$$

if we define c to be $c_1 + c_2 c_3$. This statement is exactly what we need to conclude that $T_1(n) + T_2(n)$ is $O\big(f_1(n)\big)$.

✦ **Example 3.9.** Consider the program fragment in Fig. 3.7. This program makes `A` an $n \times n$ identity matrix. Lines (2) through (4) place 0 in every cell of the $n \times n$ array, and then lines (5) and (6) place 1's in all the diagonal positions from `A[0][0]` to `A[n-1][n-1]`. The result is an identity matrix `A` with the property that

$$\mathtt{A} \times \mathtt{M} = \mathtt{M} \times \mathtt{A} = \mathtt{M}$$

```
(1)        scanf("%d",&n);
(2)        for (i = 0; i < n; i++)
(3)            for (j = 0; j < n; j++)
(4)                A[i][j] = 0;
(5)        for (i = 0; i < n; i++)
(6)            A[i][i] = 1;
```

Fig. 3.7. Program fragment to make `A` an identity matrix.

for any $n \times n$ matrix `M`.

Line (1), which reads `n`, takes $O(1)$ time, that is, some constant amount of time, independent of the value n. The assignment statement in line (6) also takes $O(1)$ time, and we go around the loop of lines (5) and (6) exactly n times, for a total of $O(n)$ time spent in that loop. Similarly, the assignment of line (4) takes $O(1)$ time. We go around the loop of lines (3) and (4) n times, for a total of $O(n)$ time. We go around the outer loop, lines (2) to (4), n times, taking $O(n)$ time per iteration, for a total of $O(n^2)$ time.

Thus, the time taken by the program in Fig. 3.7 is $O(1) + O(n^2) + O(n)$, for the statement (1), the loop of lines (2) to (4), and the loop of lines (5) and (6), respectively. More formally, if

$T_1(n)$ is the time taken by line (1),
$T_2(n)$ is the time taken by lines (2) to (4), and
$T_3(n)$ is the time taken by lines (5) and (6),

then

$T_1(n)$ is $O(1)$,
$T_2(n)$ is $O(n^2)$, and
$T_3(n)$ is $O(n)$.

We thus need an upper bound on $T_1(n) + T_2(n) + T_3(n)$ to derive the running time of the entire program.

Since the constant 1 is certainly $O(n^2)$, we can apply the rule for sums to conclude that $T_1(n) + T_2(n)$ is $O(n^2)$. Then, since n is $O(n^2)$, we can apply the rule of sums to $(T_1(n) + T_2(n))$ and $T_3(n)$, to conclude that $T_1(n) + T_2(n) + T_3(n)$ is $O(n^2)$. That is, the entire program fragment of Fig. 3.7 has running time $O(n^2)$. Informally, it spends almost all its time in the loop of lines (2) through (4), as we might expect, simply from the fact that, for large n, the area of the matrix, n^2, is much larger than its diagonal, which consists of n cells. ✦

Example 3.9 is just an application of the rule that low order terms don't matter, since we threw away the terms 1 and n, which are lower-degree polynomials than n^2. However, the rule of sums allows us to do more than just throw away low-order terms. If we have any constant number of terms that are, to within big-oh, the same, such as a sequence of 10 assignment statements, each of which takes $O(1)$ time, then we can "add" ten $O(1)$'s to get $O(1)$. Less formally, the sum of 10 constants is still a constant. To see why, note that 1 is $O(1)$, so that any of the ten $O(1)$'s, can be "added" to any other to get $O(1)$ as a result. We keep combining terms until only one $O(1)$ is left.

However, we must be careful not to confuse "a constant number" of some term like $O(1)$ with a number of these terms that varies with the input size. For example, we might be tempted to observe that it takes $O(1)$ time to go once around the loop of lines (5) and (6) of Fig. 3.7. The number of times we go around the loop is n, so that the running time for lines (5) and (6) together is $O(1) + O(1) + O(1) + \cdots$ (n times). The rule for sums tells us that the sum of two $O(1)$'s is $O(1)$, and by induction we can show that the sum of any constant number of $O(1)$'s is $O(1)$. However, in this program, n is not a constant; it varies with the input size. Thus, no one sequence of applications of the sum rule tells us that n $O(1)$'s has any value in particular. Of course, if we think about the matter, we know that the sum of n c's, where c is some constant, is cn, a function that is $O(n)$, and that is the true running time of lines (5) and (6).

Incommensurate Functions

It would be nice if any two functions $f(n)$ and $g(n)$ could be compared by big-oh; that is, either $f(n)$ is $O\big(g(n)\big)$, or $g(n)$ is $O\big(f(n)\big)$ (or both, since as we observed, there are functions such as $2n^2$ and $n^2 + 3n$ that are each big-oh of the other). Unfortunately, there are pairs of *incommensurate* functions, neither of which is big-oh of the other.

✦ **Example 3.10.** Consider the function $f(n)$ that is n for odd n and n^2 for even n. That is, $f(1) = 1$, $f(2) = 4$, $f(3) = 3$, $f(4) = 16$, $f(5) = 5$, and so on. Similarly, let $g(n)$ be n^2 for odd n and let $g(n)$ be n for even n. Then $f(n)$ cannot be $O\big(g(n)\big)$, because of the even n's. For as we observed in Section 3.4, n^2 is definitely not $O(n)$. Similarly, $g(n)$ cannot be $O\big(f(n)\big)$, because of the odd n's, where the values of g outrace the corresponding values of f. ✦

EXERCISES

3.5.1: Prove the following:

a) n^a is $O(n^b)$ if $a \leq b$.
b) n^a is not $O(n^b)$ if $a > b$.
c) a^n is $O(b^n)$ if $1 < a \leq b$.
d) a^n is not $O(b^n)$ if $1 < b < a$.
e) n^a is $O(b^n)$ for any a, and for any $b > 1$.
f) a^n is not $O(n^b)$ for any b, and for any $a > 1$.
g) $(\log n)^a$ is $O(n^b)$ for any a, and for any $b > 0$.
h) n^a is not $O\big((\log n)^b\big)$ for any b, and for any $a > 0$.

3.5.2: Show that $f(n) + g(n)$ is $O\Big(\max\big(f(n), g(n)\big)\Big)$.

3.5.3: Suppose that $T(n)$ is $O\big(f(n)\big)$ and $g(n)$ is a function whose value is never negative. Prove that $g(n)T(n)$ is $O\big(g(n)f(n)\big)$.

3.5.4: Suppose that $S(n)$ is $O\big(f(n)\big)$ and $T(n)$ is $O\big(g(n)\big)$. Assume that none of these functions is negative for any n. Prove that $S(n)T(n)$ is $O\big(f(n)g(n)\big)$.

3.5.5: Suppose that $f(n)$ is $O\big(g(n)\big)$. Show that $\max\big(f(n), g(n)\big)$ is $O\big(g(n)\big)$.

3.5.6*: Show that if $f_1(n)$ and $f_2(n)$ are both tight bounds on some function $T(n)$, then $f_1(n)$ and $f_2(n)$ are each big-oh of the other.

3.5.7*: Show that $\log_2 n$ is not $O\bigl(f(n)\bigr)$, where $f(n)$ is the function from Fig. 3.6.

3.5.8: In the program of Fig. 3.7, we created an identity matrix by first putting 0's everywhere and then putting 1's along the diagonal. It might seem that a faster way to do the job is to replace line (4) by a test that asks if $i = j$, putting 1 in `A[i][j]` if so and 0 if not. We can then eliminate lines (5) and (6).

a) Write this program.

b)* Consider the programs of Fig. 3.7 and your answer to (a). Making simplifying assumptions like those of Example 3.1, compute the number of time units taken by each of the programs. Which is faster? Run the two programs on various-sized arrays and plot their running times.

3.6 Analyzing the Running Time of a Program

Armed with the concept of big-oh and the rules from Sections 3.4 and 3.5 for manipulating big-oh expressions, we shall now learn how to derive big-oh upper bounds on the running times of typical programs. Whenever possible, we shall look for simple and tight big-oh bounds. In this section and the next, we shall consider only programs without function calls (other than library functions such as `printf`), leaving the matter of function calls to Sections 3.8 and beyond.

We do not expect to be able to analyze arbitrary programs, since questions about running time are as hard as any in mathematics. On the other hand, we can discover the running time of most programs encountered in practice, once we learn some simple rules.

The Running Time of Simple Statements

We ask the reader to accept the principle that certain simple operations on data can be done in $O(1)$ time, that is, in time that is independent of the size of the input. These primitive operations in C consist of

1. Arithmetic operations (e.g. `+` or `%`).
2. Logical operations (e.g., `&&`).
3. Comparison operations (e.g., `<=`).
4. Structure accessing operations (e.g. array-indexing like `A[i]`, or pointer following with the `->` operator).
5. Simple assignment such as copying a value into a variable.
6. Calls to library functions (e.g., `scanf`, `printf`).

The justification for this principle requires a detailed study of the machine instructions (primitive steps) of a typical computer. Let us simply observe that each of the described operations can be done with some small number of machine instructions; often only one or two instructions are needed.

Simple statement

As a consequence, several kinds of statements in C can be executed in $O(1)$ time, that is, in some constant amount of time independent of input. These *simple statements* include

1. Assignment statements that do not involve function calls in their expressions.
2. Read statements.
3. Write statements that do not require function calls to evaluate arguments.
4. The jump statements `break`, `continue`, `goto`, and `return` *expression*, where *expression* does not contain a function call.

In (1) through (3), the statements each consist of some finite number of primitive operations, each of which we take on faith to require $O(1)$ time. The summation rule then tells us that the entire statements take $O(1)$ time. Of course, the constants hidden in the big-oh are larger for statements than for single primitive operations, but as we already know, we cannot associate concrete constants with running time of C statements anyway.

✦ **Example 3.11.** We observed in Example 3.9 that the read statement of line (1) of Fig. 3.7 and the assignments of lines (4) and (6) each take $O(1)$ time. For another illustration, consider the fragment of the selection-sort program shown in Fig. 3.8. The assignments of lines (2), (5), (6), (7), and (8) each take $O(1)$ time. ✦

```
(1)        for (i = 0; i < n-1; i++) {
(2)            small = i;
(3)            for (j = i+1; j < n; j++)
(4)                if (A[j] < A[small])
(5)                    small = j;
(6)            temp = A[small];
(7)            A[small] = A[i];
(8)            A[i] = temp;
           }
```

Fig. 3.8. Selection-sort fragment.

Blocks of simple statements

Frequently, we find a block of simple statements that are executed consecutively. If the running time of each of these statements is $O(1)$, then the entire block takes $O(1)$ time, by the summation rule. That is, any constant number of $O(1)$'s sums to $O(1)$.

✦ **Example 3.12.** Lines (6) through (8) of Fig. 3.8 form a block, since they are always executed consecutively. Since each takes $O(1)$ time, the block of lines (6) to (8) takes $O(1)$ time.

Note that we should not include line (5) in the block, since it is part of the if-statement on line (4). That is, sometimes lines (6) to (8) are executed without executing line (5). ✦

The Running Time of Simple For-Loops

In C, many for-loops are formed by initializing an index variable to some value and incrementing that variable by 1 each time around the loop. The for-loop ends when the index reaches some limit. For instance, the for-loop of line (1) of Fig. 3.8 uses index variable `i`. It increments i by 1 each time around the loop, and the iterations stop when i reaches $n-1$.

There are more complex for-loops in C that behave more like while-statements; these loops iterate an unpredictable number of times. We shall take up this sort of loop later in the section. However, for the moment, focus on the simple form of for-loop, where the difference between the final and initial values, divided by the amount by which the index variable is incremented tells us how many times we go around the loop. That count is exact, unless there are ways to exit the loop via a jump statement; it is an upper bound on the number of iterations in any case. For instance, the for-loop of line (1) of Fig. 3.8 iterates $\big((n-1)-0\big)/1 = n-1$ times, since 0 is the initial value of i, $n-1$ is the highest value reached by i (i.e., when i reaches $n-1$, the loop stops and no iteration occurs with $i = n-1$), and 1 is added to `i` at each iteration of the loop.

To bound the running time of the for-loop, we must obtain an upper bound on the amount of time spent in one iteration of the loop body. Note that the time for an iteration includes the time to increment the loop index (e.g., the increment statement `i++` in line (1) of Fig. 3.8), which is $O(1)$, and the time to compare the loop index with the upper limit (e.g., the test statement `i<n-1` in line (1) of Fig. 3.8), which is also $O(1)$. In all but the exceptional case where the loop body is empty, these $O(1)$'s can be dropped by the summation rule.

In the simplest case, where the time spent in the loop body is the same for each iteration, we can multiply the big-oh upper bound for the body by the number of times around the loop. Strictly speaking, we must then add $O(1)$ time to initialize the loop index and $O(1)$ time for the first comparison of the loop index with the limit, because we test one more time than we go around the loop. However, unless it is possible to execute the loop zero times, the time to initialize the loop and test the limit once is a low-order term that can be dropped by the summation rule.

✦ **Example 3.13.** Consider the for-loop of lines (3) and (4) in Fig. 3.7, which is

```
(3)            for (j = 0; j < n; j++)
(4)                A[i][j] = 0;
```

We know that line (4) takes $O(1)$ time. Clearly, we go around the loop n times, as we can determine by subtracting the lower limit from the upper limit found on line (3) and then adding 1. Since the body, line (4), takes $O(1)$ time, we can neglect the time to increment `j` and the time to compare j with n, both of which are also $O(1)$. Thus, the running time of lines (3) and (4) is the product of n and $O(1)$, which is

$O(n)$.

Similarly, we can bound the running time of the outer loop consisting of lines (2) through (4), which is

```
(2)         for (i = 0; i < n; i++)
(3)             for (j = 0; j < n; j++)
(4)                 A[i][j] = 0;
```

We have already established that the loop of lines (3) and (4) takes $O(n)$ time. Thus, we can neglect the $O(1)$ time to increment `i` and to test whether $i < n$ in each iteration, concluding that each iteration of the outer loop takes $O(n)$ time. The initialization `i = 0` of the outer loop and the $(n+1)$st test of the condition $i < n$ likewise take $O(1)$ time and can be neglected. Finally, we observe that we go around the outer loop n times, taking $O(n)$ time for each iteration, giving a total $O(n^2)$ running time. ✦

✦ **Example 3.14.** Now, let us consider the for-loop of lines (3) to (5) of Fig. 3.8. Here, the body is an if-statement, a construct we shall learn how to analyze next. It is not hard to deduce that line (4) takes $O(1)$ time to perform the test and line (5), if we execute it, takes $O(1)$ time because it is an assignment with no function calls. Thus, we take $O(1)$ time to execute the body of the for-loop, regardless of whether line (5) is executed. The incrementation and test in the loop add $O(1)$ time, so that the total time for one iteration of the loop is just $O(1)$.

Now we must calculate the number of times we go around the loop. The number of iterations is not related to n, the size of the input. Rather, the formula "last value minus initial value divided by the increment" gives us $\bigl(n-(i+1)\bigr)/1$, or $n-i-1$, as the number of times around the loop. Strictly speaking, that formula holds only if $i < n$. Fortunately, we can observe from line (1) of Fig. 3.8 that we do not enter the loop body of lines (2) through (8) unless $i \le n-2$. Thus, not only is $n-i-1$ the number of iterations, but we also know that this number cannot be 0. We conclude that the time spent in the loop is $(n-i-1) \times O(1)$, or $O(n-i-1)$.[5] We do not have to add in $O(1)$ for initializing `j`, since we have established that $n-i-1$ cannot be 0. If we had not observed that $n-i-1$ was positive, then we would have to write the upper bound on the running time as $O\bigl(\max(1, n-i-1)\bigr)$. ✦

The Running Time of Selection Statements

An if-else selection statement has the form

```
if (<condition>)
    <if-part>
else
    <else-part>
```

where

1. The condition is an expression to be evaluated,

[5] Technically, we have not discussed a big-oh operator applied to a function of more than one variable. In this case, we can regard $O(n-i-1)$ as saying "at most some constant times $n-i-1$." That is, we can consider $n-i-1$ as a surrogate for a single variable.

2. The if-part is a statement that is executed only if the condition is true (the value of the expression is not zero), and

3. The else-part is a statement that is executed if the condition is false (evaluates to 0). The **else** followed by the <`else-part`> is optional.

A condition, no matter how complex, requires the computer to perform only a constant number of primitive operations, as long as there are no function calls within the condition. Thus, the evaluation of the condition will take $O(1)$ time.

Suppose that there are no function calls in the condition, and that the if- and else-parts have big-oh upper bounds $f(n)$ and $g(n)$, respectively. Let us also suppose that $f(n)$ and $g(n)$ are not both 0; that is, while the else-part may be missing, the if-part is something other than an empty block. We leave as an exercise the question of determining what happens if both parts are missing or are empty blocks.

If $f(n)$ is $O\big(g(n)\big)$, then we can take $O\big(g(n)\big)$ to be an upper bound on the running time of the selection statement. The reason is that

1. We can neglect the $O(1)$ for the condition,

2. If the else-part is executed, $g(n)$ is known to be a bound on the running time, and

3. If the if-part is executed instead of the else-part, the running time will be $O\big(g(n)\big)$ because $f(n)$ is $O\big(g(n)\big)$.

Similarly, if $g(n)$ is $O\big(f(n)\big)$, we can bound the running time of the selection statement by $O\big(f(n)\big)$. Note that when the else-part is missing, as it often is, $g(n)$ is 0, which is surely $O\big(f(n)\big)$.

The problem case is when neither f nor g is big-oh of the other. We know that either the if-part or the else-part, but not both, will be executed, and so a safe upper bound on the running time is the larger of $f(n)$ and $g(n)$. Which is larger can depend on n, as we saw in Example 3.10. Thus, we must write the running time of the selection statement as $O\Big(\max\big(f(n), g(n)\big)\Big)$.

✦ **Example 3.15.** As we observed in Example 3.12, the selection statement of lines (4) and (5) of Fig. 3.8 has an if-part, line (5), which takes $O(1)$ time, and a missing else-part, which takes 0 time. Thus, $f(n)$ is 1 and $g(n)$ is 0. As $g(n)$ is $O\big(f(n)\big)$, we get $O(1)$ as an upper bound on running time for lines (4) and (5). Note that the $O(1)$ time to perform the test `A[j]` < `A[small]` at line (4) can be neglected. ✦

✦ **Example 3.16.** For a more complicated example, consider the fragment of code in Fig. 3.9, which performs the (relatively pointless) task of either zeroing the matrix `A` or setting its diagonal to 1's. As we learned in Example 3.13, the running time of lines (2) through (4) is $O(n^2)$, while the running time of lines (5) and (6) is $O(n)$. Thus, $f(n)$ is n^2 here, and $g(n)$ is n. Since n is $O(n^2)$, we can neglect the time of the else-part and take $O(n^2)$ as a bound on the running time of the entire fragment of Fig. 3.9. That is to say, we have no idea if or when the condition of line (1) will be true, but the only safe upper bound results from assuming the worst: that the condition is true and the if-part is executed. ✦

```
if (A[1][1] == 0)
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            A[i][j] = 0;
else
    for (i = 0; i < n; i++)
        A[i][i] = 1;
```

Fig. 3.9. Example of an if-else selection statement.

The Running Time of Blocks

Compound statement

We already mentioned that a sequence of assignments, reads, and writes, each of which takes $O(1)$ time, together takes $O(1)$ time. More generally, we must be able to combine sequences of statements, some of which are *compound statements,* that is, selection statements or loops. Such a sequence of simple and compound statements is called a *block.* The running time of a block is calculated by taking the sum of the big-oh upper bounds for each of the (possibly compound) statements in the block. With luck, we can use the summation rule to eliminate all but one of the terms in the sum.

Example 3.17. In the selection sort fragment of Fig. 3.8, we can view the body of the outer loop, that is, lines (2) through (8), as a block. This block consists of five statements:

1. The assignment of line (2)
2. The loop of lines (3), (4), and (5)
3. The assignment of line (6)
4. The assignment of line (7)
5. The assignment of line (8)

Note that the selection statement of lines (4) and (5), and the assignment of line (5), are not visible at the level of this block; they are hidden within a larger statement, the for-loop of lines (3) to (5).

We know that the four assignment statements take $O(1)$ time each. In Example 3.14 we learned that the running time of the second statement of the block — that is, lines (3) through (5) — is $O(n-i-1)$. Thus, the running time of the block is

$$O(1) + O(n-i-1) + O(1) + O(1) + O(1)$$

Since 1 is $O(n-i-1)$ (recall we also deduced that i never gets higher than $n-2$), we can eliminate all the $O(1)$'s by the summation rule. Thus, the entire block takes $O(n-i-1)$ time.

For another example, consider the program fragment of Fig. 3.7 again. It can be considered a single block consisting of three statements:

1. The read statement of line (1)
2. The loop of lines (2) through (4)
3. The loop of lines (5) and (6)

We know that line (1) takes $O(1)$ time. From Example 3.13, lines (2) through (4) take $O(n^2)$ time; lines (5) and (6) take $O(n)$ time. The block itself takes

$$O(1) + O(n^2) + O(n)$$

time. By the summation rule, we can eliminate $O(1)$ and $O(n)$ in favor of $O(n^2)$. We conclude that the fragment of Fig. 3.7 takes $O(n^2)$ time. ✦

The Running Time of Complex Loops

In C, there are while-, do-while-, and some for-loops that do not offer an explicit count of the number of times we go around the loop. For these loops, part of the analysis is an argument that provides an upper bound on the number of iterations of the loop. These proofs typically follow the pattern we learned in Section 2.5. That is, we prove some statement by induction on the number of times around the loop, and the statement implies that the loop condition must become false after the number of iterations reaches a certain limit.

We must also establish a bound on the time to perform one iteration of the loop. Thus, we examine the body and obtain a bound on its execution. To that, we must add $O(1)$ time to test the condition after the execution of the loop body, but unless the loop body is missing, we can neglect this $O(1)$ term. We get a bound on the running time of the loop by multiplying an upper bound on the number of iterations by our upper bound on the time for one iteration. Technically, if the loop is a for- or while-loop rather than a do-while-loop, we must include the time needed to test the condition the first time, before entering the body. However, that $O(1)$ term can normally be neglected.

✦ **Example 3.18.** Consider the program fragment shown in Fig. 3.10. The program searches an array `A[0..n-1]` for the location of an element x that is believed to be in the array.

```
(1)        i = 0;
(2)        while(x != A[i])
(3)            i++;
```

Fig. 3.10. Program fragment for linear search.

The two assignment statements (1) and (3) in Fig. 3.10 have a running time of $O(1)$. The while-loop of lines (2) and (3) may be executed as many as n times, but no more, because we assume one of the array elements is x. Since the loop body, line (3), requires time $O(1)$, the running time of the while-loop is $O(n)$. From the summation rule, the running time of the entire program fragment is $O(n)$, because that is the maximum of the time for the assignment of line (1) and the time for the while-loop. In Chapter 6, we shall see how this $O(n)$ program can be replaced by an $O(\log n)$ program using binary search. ✦

EXERCISES

3.6.1: In a for-loop headed

```
for (i = a; i <= b; i++)
```

how many times do we go around the loop, as a function of a and b? What about a for-loop headed

```
for (i = a; i <= b; i--)
```

What about A for-loop headed

```
for(i = a; i <= b; i = i+c)
```

3.6.2: Give a big-oh upper bound on the running time of the trivial selection statement

```
if ( C ) {  }
```

where C is a condition that does not involve any function calls.

3.6.3: Repeat Exercise 3.6.2 for the trivial while-loop

```
while ( C ) {  }
```

3.6.4*: Give a rule for the running time of a C switch-statement.

3.6.5: Give a rule for the running time of a selection statement in which we can tell which branch is taken, such as

```
if (1==2)
    something O(f(n));
else
    something O(g(n));
```

3.6.6: Give a rule for the running time of a degenerate while-loop, in which the condition is known to be false right from the start, such as

```
while (1 != 1)
    something O(f(n));
```

✦✦✦ 3.7 A Recursive Rule for Bounding Running Time

In the previous section, we informally described a number of rules for defining the running time of certain program constructs in terms of the running times of their parts. For instance, we said that the time of a for-loop is essentially the time taken by the body multiplied by the number of iterations. Hidden among these rules was the notion that programs are built using inductive rules by which compound statements (loops, selections, and other statements that have substatements as constituent parts) are constructed from a basis consisting of simple statements such as assignment, read, write, and jump statements. The inductive rules covered loop formation, selection statements, and blocks, which are sequences of compound statements.

We shall state some syntactic rules for building statements of C as a recursive definition. These rules correspond to the grammatical rules for defining C that often appear in a text on C. We shall see in Chapter 11 that grammars can serve as a succinct recursive notation for specifying the syntax of programming languages.

More Defensive Programming

If you think that the array `A` in Example 3.18 will always have the element x just because we believe it does, there's a bridge we'd like to sell you. Notice that if x fails to appear in the array, the loop of Fig. 3.10 will eventually err by trying to access an element of the array that is beyond the array's upper limit.

Sentinel

Fortunately, there is a simple way to avoid this error without spending a lot of extra time in each iteration of the loop. We allow an extra $(n+1)$st cell at the end of the array, and before starting the loop, we place x there. Then we really can be sure that x will appear somewhere in the array. When the loop ends, we test to see whether $i = n$. If so, then x wasn't really in the array, and we fell through the array to the copy of x we had placed as a *sentinel.* If $i < n$, then i does indicate a position where x appears. An example program with this protection feature is

```
A[n] = x;
i = 0;
while (x != A[i])
    i++;
if (i == n) /* do something appropriate to the case
                       that x is not in the array */
else /* do something appropriate to the case
                       that x is found at position i */
```

BASIS. The following are simple statements in C:

1. *Expressions*, including assignment statements and read and write statements; the latter are calls to functions such as `printf` and `scanf`.

2. *Jump statements*, including `goto`'s, `break`'s, `continue`'s, and `return`'s.

3. The *null statement.*

Note that in C, simple statements end in a semicolon, which we take to be part of the statement.

INDUCTION. The following rules let us construct statements from smaller statements:

1. *While-statement.* If S is a statement and C is a condition (an expression with an arithmetic value), then

   while (C) S

   is a statement. The body S is executed as long as C is true (has a nonzero value).

2. *Do-while-statement.* If S is a statement and C is a condition, then

   do S **while** (C) ;

   is a statement. The do-while is similar to the while-statement except that the body S is executed at least once.

3. *For-statement.* If S is a statement and E_1, E_2, and E_3 are expressions, then

   for (E_1 ; E_2 ; E_3) S

   is a statement. The first expression E_1 is evaluated once and specifies the initialization for the loop body S. The second expression E_2 is the test for loop termination, which is evaluated before each iteration. If its value is nonzero, the body is executed; otherwise, the for-loop is terminated. The third expression E_3 is evaluated after each iteration and specifies the reinitialization (incrementation) for the next iteration of the loop. For instance, a common for-loop is

   `for (i = 0; i < n; i++)` S

   where S is iterated n times with i having the values $0, 1, \ldots, n-1$. Here, `i = 0` is the initialization, `i < n` is the termination test, and `i++` is the reinitialization.

4. *Selection statement.* If S_1 and S_2 are statements and C is a condition, then

   if (C) S_1 **else** S_2

   is a statement, and

   if (C) S_1

   is also a statement. In the first case, if C is true (nonzero), then S_1 is executed; otherwise, S_2 is executed. In the second case, S_1 is executed only if C is true.

5. *Block.* If $S_1, S_2, \ldots, S_n$ are statements, then

   {S_1 $S_2 \cdots S_n$}

   is a statement.

We have omitted from this list the switch-statement, which has a complex form but can be thought of as nested selection statements for analysis of running time.

Using this recursive definition of statements, it is possible to parse a program by identifying its constituent parts. That is, we begin with the simple statements and group them into progressively larger compound statements.

✦ **Example 3.19.** Consider the selection-sort program fragment shown in Fig. 3.11. For the basis, each of the assignments, lines (2), (5), (6), (7), and (8), is a statement by itself. Then lines (4) and (5) are grouped into a selection statement. Next lines (3) through (5) are grouped into a for-statement. Then lines (2) through (8) are grouped into a block. Finally, the entire program fragment is a for-statement. ✦

A Tree Representation of a Program's Structure

We can represent the structure of programs by a tree such as that shown in Fig. 3.12. Leaves (the circles) are simple statements and other nodes represent compound statements.[6] Nodes are labeled by the kind of construct they represent and by the

[6] Trees are discussed in detail in Chapter 5.

```
(1)        for (i = 0; i < n-1; i++) {
(2)            small = i;
(3)            for (j = i+1; j < n; j++)
(4)                if (A[j] < A[small])
(5)                    small = j;
(6)            temp = A[small];
(7)            A[small] = A[i];
(8)            A[i] = temp;
           }
```

Fig. 3.11. Selection sort fragment.

Structure tree

line or lines of the program that form the simple or compound statement represented by the node. From each node N that represents a compound statement we find lines downward to its "children" nodes. The children of node N represent the substatements forming the compound statement that N represents. Such a tree is called the *structure tree* for a program.

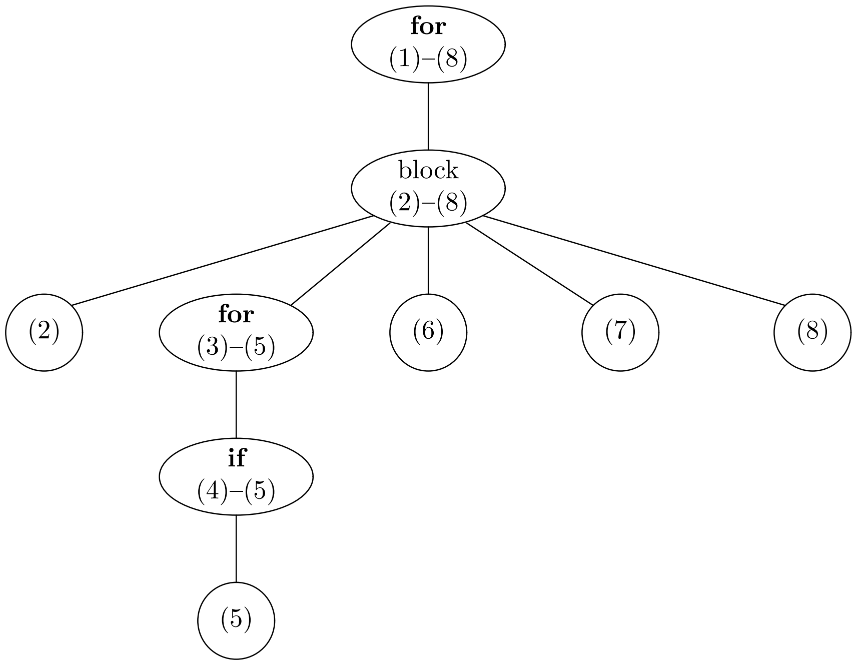


Fig. 3.12. Tree showing grouping of statements.

✦ **Example 3.20.** Figure 3.12 is the structure tree for the program fragment of Fig. 3.11. Each of the circles is a leaf representing one of the five assignment statements of Fig. 3.11. We have omitted from Fig. 3.12 an indication that these five statements are assignment statements.

At the top (the "root") of the tree is a node representing the entire fragment of lines (1) through (8); it is a for-statement. The body of the for-loop is a block consisting of lines (2) through (8).[7] This block is represented by a node just below the root. That block node has five children, representing the five statements of the block. Four of them are assignment statements, lines (2), (6), (7), and (8). The fifth is the for-loop of lines (3) through (5).

The node for this for-loop has a child representing its body, the if-statement of lines (4) and (5). The latter node has a child representing its constituent statement, the assignment of line (5). ✦

Climbing the Structure Tree to Determine Running Time

Just as program structures are built recursively, we can define big-oh upper bounds on the running time of programs, using an analogous recursive method. As in Section 3.6, we presume that there are no function calls within the expressions that form assignment statements, print statements, conditions in selections, conditions in while-, for-, or do-while- loops, or the initialization or reinitialization of a for-loop. The only exception is a call to a read- or write-function such as `printf`.

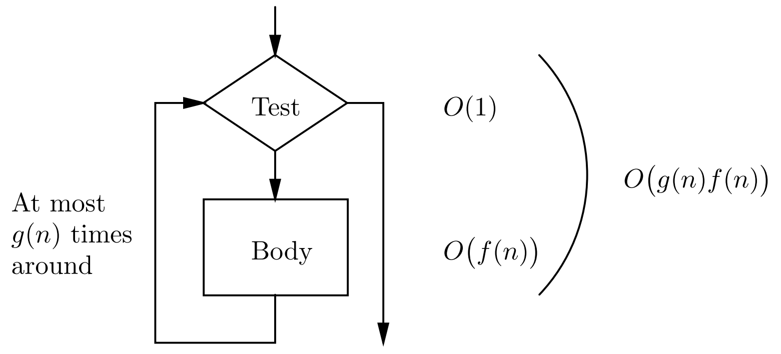
BASIS. The bound for a simple statement — that is, for an assignment, a read, a write, or a jump — is $O(1)$.

INDUCTION. For the five compound constructs we have discussed, the rules for computing their running time are as follows.

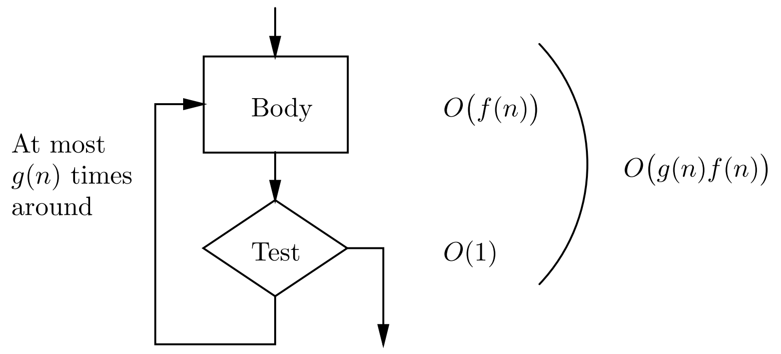
1. *While-statement.* Let $O\big(f(n)\big)$ be the upper bound on the running time of the body of the while-statement; $f(n)$ was discovered by the recursive application of these rules. Let $g(n)$ be an upper bound on the number of times we may go around the loop. Then $O\Big(1 + \big(f(n) + 1\big)g(n)\Big)$ is an upper bound on the running time of the while-loop. That is, $O\big(f(n)+1\big)$ is an upper bound on the running time of the body plus the test after the body. The additional 1 at the beginning of the formula accounts for the first test, before entering the loop. In the common case where $f(n)$ and $g(n)$ are at least 1 (or we can define them to be 1 if their value would otherwise be 0), we can write the running time of the while-loop as $O\big(f(n)g(n)\big)$. This common formula for running time is suggested by Fig. 3.13(a).

2. *Do-while-statement.* If $O\big(f(n)\big)$ is an upper bound on the body of the loop and $g(n)$ is an upper bound on the number of times we can go around the loop, then $O\Big(\big(f(n)+1\big)g(n)\Big)$ is an upper bound on the running time of the do-while-loop. The "+1" represents the time to compute and test the condition at the end of each iteration of the loop. Note that for a do-while-loop, $g(n)$ is always at least 1. In the common case that $f(n) \geq 1$ for all n, the running time of the do-while-loop is $O\big(f(n)g(n)\big)$. Figure 3.13(b) suggests the way that running time is computed for the common case of a do-while-loop.

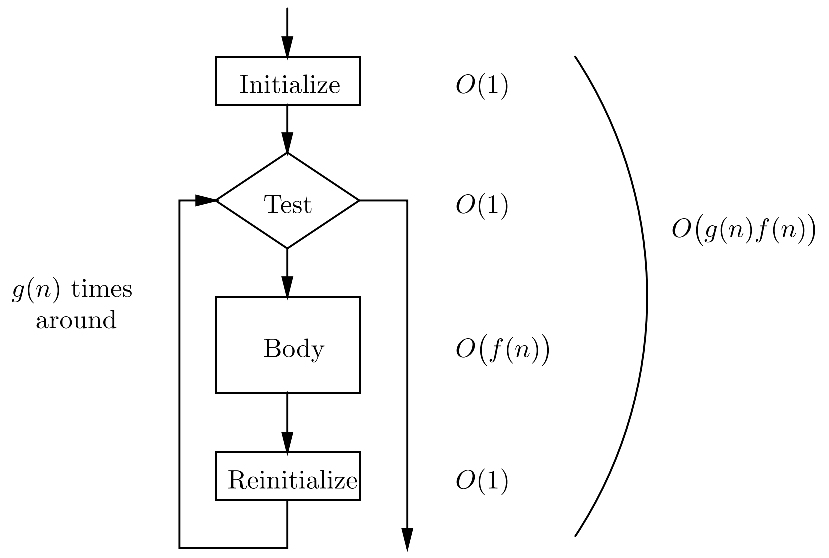
[7] A more detailed structure tree would have children representing the expressions for the initialization, termination test, and reinitialization of the for-loop.



(a) While-statement.



(b) Do-while-statement.



(c) For-statement.

Fig. 3.13. Computing the running time of loop statements without function calls.

3. *For-statement.* If $O\big(f(n)\big)$ is an upper bound on the running time of the body, and $g(n)$ is an upper bound on the number of times around the loop, then an upper bound on the time of a for-statement is $O\Big(1 + \big(f(n) + 1\big)g(n)\Big)$. The factor $f(n) + 1$ represents the cost of going around once, including the body, the test, and the reinitialization. The "1+" at the beginning represents the first initialization and the possibility that the first test is negative, resulting in zero iterations of the loop. In the common case that $f(n)$ and $g(n)$ are both at least 1, or can be redefined to be at least 1, then the running time of the for-statement is $O\big(f(n)g(n)\big)$, as is illustrated by Fig. 3.13(c).

4. *Selection statement.* If $O\big(f_1(n)\big)$ and $O\big(f_2(n)\big)$ are upper bounds on the running time of the if-part and the else-part, respectively ($f_2(n)$ is 0 if the else-part is missing), then an upper bound on the running time of the selection statement is $O\Big(1 + \max\big(f_1(n), f_2(n)\big)\Big)$. The "1+" represents the test; in the common case, where at least one of $f_1(n)$ and $f_2(n)$ are positive for all n, the "1+" can be omitted. Further, if one of $f_1(n)$ and $f_2(n)$ is big-oh of the other, this expression may be simplified to whichever is the larger, as was stated in Exercise 3.5.5. Figure 3.14 suggests the computation of running time for an if-statement.

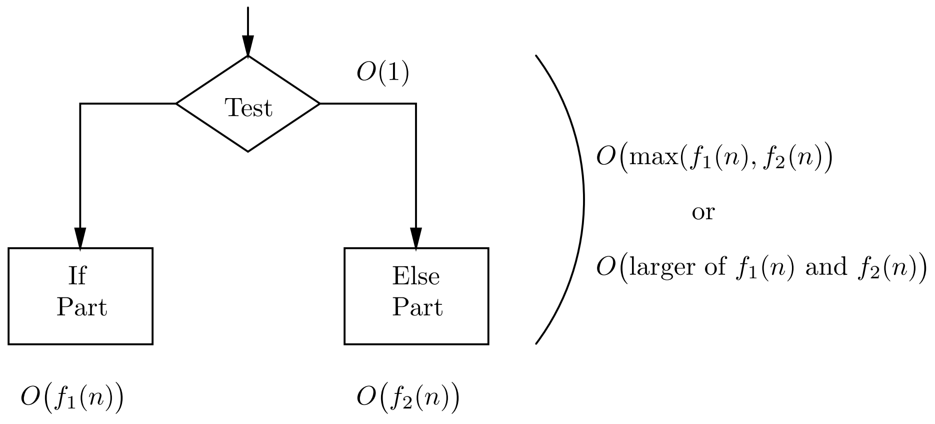

Test
$O(1)$
If Part
Else Part
$O\big(f_1(n)\big)$
$O\big(f_2(n)\big)$
$O\big(\max(f_1(n), f_2(n)\big)$
or
$O\big(\text{larger of } f_1(n) \text{ and } f_2(n)\big)$

Fig. 3.14. Computing the running time of an if-statement without function calls.

5. *Block.* If $O\big(f_1(n)\big), O\big(f_2(n)\big), \ldots, O\big(f_k(n)\big)$ are our upper bounds on the statements within the block, then $O\big(f_1(n) + f_2(n) + \cdots + f_k(n)\big)$ is an upper bound on the running time of the block. If possible, use the summation rule to simplify this expression. The rule is illustrated in Fig. 3.15.

We apply these rules traveling up the structure tree that represents the construction of compound statements from smaller statements. Alternatively, we can see the application of these rules as beginning with the simple statements covered by the basis and then proceeding to progressively larger compound statements, applying whichever of the five inductive rules is appropriate at each step. However we view the process of computing the upper bound on running time, we analyze

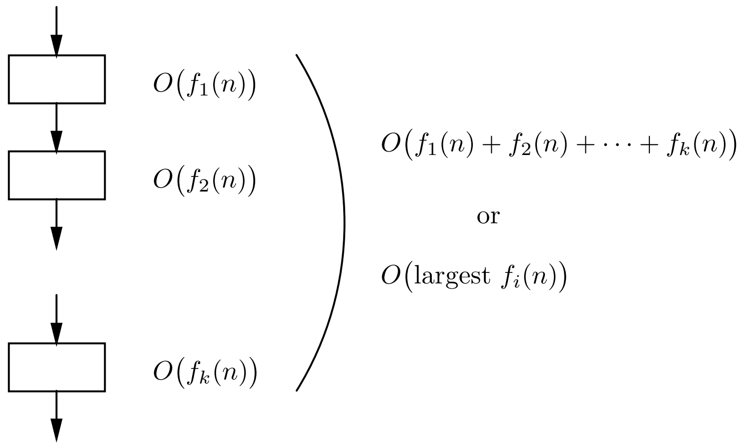

$O(f_1(n))$
$O(f_2(n))$
$O(f_k(n))$
$O(f_1(n) + f_2(n) + \cdots + f_k(n))$
or
$O(\text{largest } f_i(n))$

Fig. 3.15. Computing the running time of a block without function calls.

a compound statement only after we have analyzed all the statements that are its constituent parts.

✦ **Example 3.21.** Let us revisit our selection sort program of Fig. 3.11, whose structure tree is in Fig. 3.12. To begin, we know that each of the assignment statements at the leaves in Fig. 3.12 takes $O(1)$ time. Proceeding up the tree, we next come to the if-statement of lines (4) and (5). We recall from Example 3.15 that this compound statement takes $O(1)$ time.

Next as we travel up the tree (or proceed from smaller statements to their surrounding larger statements), we must analyze the for-loop of lines (3) to (5). We did so in Example 3.14, where we discovered that the time was $O(n-i-1)$. Here, we have chosen to express the running time as a function of the two variables `n` and `i`. That choice presents us with some computational difficulties, and as we shall see, we could have chosen the looser upper bound of $O(n)$. Working with $O(n-i-1)$ as our bound, we must now observe from line (1) of Fig. 3.11 that i can never get as large as $n-1$. Thus, $n-i-1$ is strictly greater than 0 and dominates $O(1)$. Consequently, we need not add to $O(n-i-1)$ an $O(1)$ term for initializing the index `j` of the for-loop.

Now we come to the block of lines (2) through (8). As discussed in Example 3.17, the running time of this block is the sum of four $O(1)$'s corresponding to the four assignment statements, plus the term $O(n-i-1)$ for the compound statement of lines (3) to (5). By the rule for sums, plus our observation that $i < n$, we drop the $O(1)$'s, leaving $O(n-i-1)$ as the running time of the block.

Finally, we must consider the for-loop of lines (1) through (8). This loop was not analyzed in Section 3.6, but we can apply inductive rule (3). That rule needs an upper bound on the running time of the body, which is the block of lines (2) through (8). We just determined the bound $O(n-i-1)$ for this block, presenting us with a situation we have not previously seen. While `i` is constant within the block, `i` is the index of the outer for-loop, and therefore varies within the loop. Thus, the bound $O(n-i-1)$ makes no sense if we think of it as the running time of all iterations of the loop. Fortunately, we can see from line (1) that i is never below 0, and so $O(n-1)$ is an upper bound on $O(n-i-1)$. Moreover, by our rule that low-order terms don't matter, we can simplify $O(n-1)$ to $O(n)$.

Next, we need to determine the number of times we go around the loop. Since `i` ranges from 0 to $n-2$, evidently we go around $n-1$ times. When we multiply $n-1$ by $O(n)$, we get $O(n^2-n)$. Throwing away low-order terms again, we see that $O(n^2)$ is an upper bound on the running time of the whole selection sort program. That is to say, selection sort has a quadratic upper bound on its running time. The quadratic upper bound is the tightest possible, since we can show that if the elements are initially in reverse of sorted order, then selection sort does make $n(n-1)/2$ comparison steps. ✦

As we shall see, we can derive $n \log n$ upper and lower bounds for the running time of merge sort. In practice, merge sort is more efficient than selection sort for all but some small values of n. The reason merge sort is sometimes slower than selection sort is because the $O(n \log n)$ upper bound hides a larger constant than the $O(n^2)$ bound for selection sort. The true situation is a pair of crossing curves, as shown in Fig. 3.2 of Section 3.3.

More Precise Upper Bounds for Loop Running Times

We have suggested that to evaluate the running time of a loop we need to find a uniform bound that applies to each of the iterations of the loop. However, a more careful analysis of a loop would treat each iteration separately. We could instead sum the upper bounds for each iteration. Technically, we must include the time to increment the index (if the loop is a for-loop) and to test the loop condition in the rare situation when the time for these operations makes a difference. Generally, the more careful analysis doesn't change the answer, although there are some unusual loops in which most iterations take very little time, but one or a few take a lot of time. Then the sum of the times for each iteration might be significantly less than the product of the number of iterations times the maximum time taken by any iteration.

✦ **Example 3.22.** We shall perform this more precise analysis on the outer loop of selection sort. Despite this extra effort, we still get a quadratic upper bound. As Example 3.21 demonstrated, the running time of the iteration of the outer loop with the value i for the index variable `i` is $O(n-i-1)$. Thus an upper bound on the time taken by all iterations, as `i` ranges from its initial value 0 up to $n-2$, is $O\big(\sum_{i=0}^{n-2}(n-i-1)\big)$. The terms in this sum form an arithmetic progression, so we can use the formula "average of the first and last terms times the number of terms." That formula tells us that

$$\sum_{i=0}^{n-2}(n-i-1) = n(n-1)/2 = 0.5n^2 - 0.5n$$

Neglecting low-order terms and constant factors, we see that $O(0.5n^2 - 0.5n)$ is the same as $O(n^2)$. We again conclude that selection sort has a quadratic upper bound on its running time. ✦

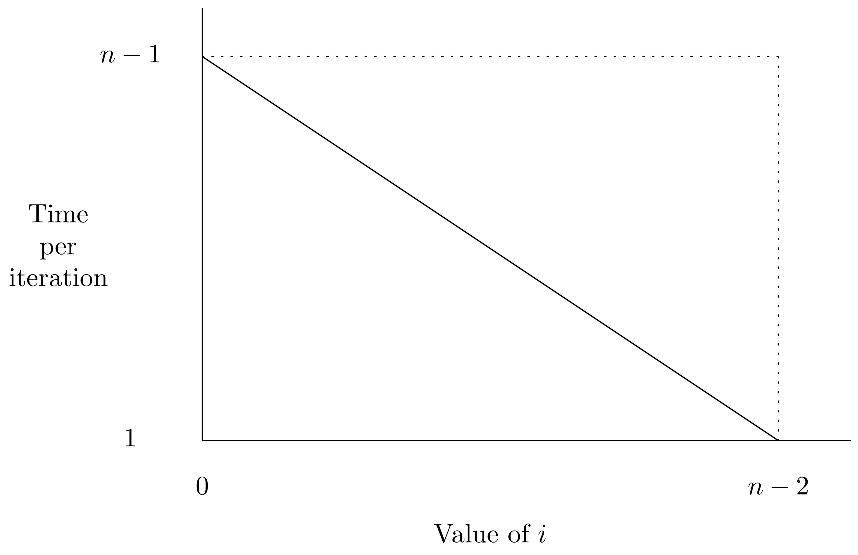


Fig. 3.16. Simple and precise estimates of loop running time.

The difference between the simple analysis in Example 3.21 and the more detailed analysis in Example 3.22 is illustrated by Fig. 3.16. In Example 3.21, we took the maximum time of any iteration as the time for each iteration, thus getting the area of the rectangle as our bound on the running time of the outer for-loop in Fig. 3.11. In Example 3.22, we bounded the running time of each iteration by the diagonal line, since the time for each iteration decreases linearly with i. Thus, we obtained the area of the triangle as our estimate of the running time. However, it is well known that the area of the triangle is half the area of the rectangle. Since the factor of 2 gets lost with the other constant factors that are hidden by the big-oh notation anyway, the two upper bounds on the running time are really the same.

EXERCISES

3.7.1: Figure 3.17 contains a C program to find the average of the elements of an array `A[0..n-1]` and to print the index of the element that is closest to average (a tie is broken in favor of the element appearing first). We assume that $n \geq 1$ and do not include the necessary check for an empty array. Build a structure tree showing how the statements are grouped into progressively more complex statements, and give a simple and tight big-oh upper bound on the running time for each of the statements in the tree. What is the running time of the entire program?

3.7.2: The fragment in Fig. 3.18 transforms an n by n matrix `A`. Show the structure tree for the fragment. Give a big-oh upper bound on the running time of each compound statement

a) Making the bound for the two inner loops a function of n and i.
b) Making the bound for all loops be a function of n only.

For the whole program, is there a big-oh difference between your answers to parts (a) and (b)?

```
#include <stdio.h>
#define MAX 100
int A[MAX];

main()
{
    int closest, i, n;
    float avg, sum;

    for (n = 0; n < MAX && scanf("%d", &A[n]) != EOF; n++)
        ;
    sum = 0;
    for (i = 0; i < n; i++)
        sum += A[i];
    avg = sum/n;
    closest = 0;
    i = 1;
    while (i < n) {
        /* squaring elements in the test below eliminates
           the need to distinguish positive and negative
           differences */
        if ((A[i]-avg)*(A[i]-avg) <
          (A[closest]-avg)*(A[closest]-avg))
                closest = i;
        i++;
    }
    printf("%d\n",closest);
}
```

Fig. 3.17. Program for Exercise 3.7.1.

```
for (i = 0; i < n-1; i++)
    for (j = i+1; j < n; j++)
        for (k = i; k < n; k++)
            A[j][k] = A[j][k] - A[i][k]*A[j][i]/A[i][i];
```

Fig. 3.18. Program for Exercise 3.7.2.

3.7.3*: Figure 3.19 contains a program fragment that applies the powers-of-2 operation discussed in Example 3.8 to the integers i from 1 to n. Show the structure tree for the fragment. Give a big-oh upper bound on the running time of each compound statement

a) Making the bound for the while-loop a function of (the factors of) i.
b) Making the bound for the while-loop a function of n only.

For the whole program, is there a big-oh difference between your answers to parts (a) and (b)?

```
(1)         for (i = 1; i <= n; i++) {
(2)             m = 0;
(3)             j = i;
(4)             while (j%2 == 0) {
(5)                 j = j/2;
(6)                 m++;
                }
            }
```

Fig. 3.19. Program for Exercise 3.7.3.

3.7.4: In Fig. 3.20 is a function that determines whether the argument n is a prime. Note that if n is not a prime, then it is divisible evenly by some integer i between 2 and $\sqrt{n}$. Show the structure tree for the function. Give a big-oh upper bound on the running time of each compound statement, as a function of n. What is the running time of the function as a whole?

```
        int prime(int n)
        {
            int i;

(1)         i = 2;
(2)         while (i*i <= n)
(3)             if (n%i == 0)
(4)                 return FALSE;
                else
(5)                 i++;
(6)         return TRUE;
        }
```

Fig. 3.20. Program for Exercise 3.7.4.

3.8 Analyzing Programs with Function Calls

We now show how to analyze the running time of a program or program fragment that contains function calls. To begin, if all functions are nonrecursive, we can determine the running time of the functions making up the program one at a time, starting with those functions that do not call any other function. Then we evaluate the running times of the functions that call only functions whose running times we have already determined. We proceed in this fashion until we have evaluated the running time for all functions.

There are some complexities introduced by the fact that for different functions there may be different natural measures of the size of the input. In general, the input to a function is the list of the arguments of that function. If function F calls function G, we must relate the size measure for the arguments of G to the measure of size that is used for F. It is hard to give useful generalities, but some examples in this section and the next will help us see how the process of bounding the running time for functions works in simple cases.

Suppose we have determined that a good upper bound on the running time of a function F is $O\big(h(n)\big)$, where n is a measure of the size of the arguments of F. Then when a call to F is made from within a simple statement (e.g., in an assignment), we add $O\big(h(n)\big)$ to the cost of that statement.

```
          #include <stdio.h>
          int bar(int x, int n);
          int foo(int x, int n);

          main()
          {
              int a, n;

    (1)       scanf("%d", &n);
    (2)       a = foo(0,n);
    (3)       printf("%d\n", bar(a,n));
          }

          int bar(int x, int n)
          {
              int i;

    (4)       for (i = 1; i <= n; i++)
    (5)           x += i;
    (6)       return x;
          }

          int foo(int x, int n)
          {
              int i;

    (7)       for (i = 1; i <= n; i++)
    (8)           x += bar(i,n);
    (9)       return x;
          }
```

Fig. 3.21. Program illustrating nonrecursive function calls.

When a function call with upper bound $O\big(h(n)\big)$ appears in a condition of a while-, do-while-, or if-statement, or in the initialization, test, or reinitialization of a for-statement, the time of that function call is accounted for as follows:

1. If the function call is in the condition of a while- or do-while-loop, or in the condition or reinitialization of a for-loop, add $h(n)$ to the bound on the time for each iteration. Then proceed as in Section 3.7 to obtain the running time of the loop.

Program Analysis in a Nutshell

Here are the major points that you should take from Sections 3.7 and 3.8.

- ✦ The running time of a sequence of statements is the sum of the running times of the statements. Often, one of the statements dominates the others by having a running time that is at least as big as any. By the summation rule the big-oh running time of the sequence is just the running time of the dominant statement.
- ✦ The running time of a loop is computed by taking the running time of the body, plus any control steps (e.g., reinitializing the index of a for-loop and comparing it to the limit). Multiply this running time by an upper bound on the number of times the loop can be iterated. Then, add anything that is done just once, such as the initialization or the first test for termination, if the loop might be iterated zero times.
- ✦ The running time of a selection statement (if-else, e.g.) is the time to decide which branch to take plus the larger of the running times of the branches.

2. If the function call is in the initialization of a for-loop, add $O\big(h(n)\big)$ to the cost of the loop.
3. If the function call is in the condition of an if-statement, add $h(n)$ to the cost of the statement.

✦ **Example 3.23.** Let us analyze the (meaningless) program of Fig. 3.21. First, let us note that it is not a recursive program. The function `main` calls both functions `foo` and `bar`, and `foo` calls `bar`, but that is all. The diagram of Fig. 3.22, called a *calling graph,* indicates how functions call one another. Since there are no cycles, there is no recursion, and we can analyze the functions by starting with "group 0," those that do not call other functions (`bar` in this case), then working on "group 1," those that call only functions in group 0 (`foo` in this case), and then on "group 2," those that call only functions in groups 0 and 1 (`main` in this case). At this point, we are done, since all functions are in groups. In general, we would have to consider a larger number of groups, but as long as there are no cycles, we can eventually place each function in a group.

Calling graph

The order in which we analyze the running time of the functions is also the order in which we should examine them in order to gain an understanding of what the program does. Thus, let us first consider what the function `bar` does. The for-loop of lines (4) and (5) adds each of the integers 1 through n to x. As a result, $bar(x, n)$ is equal to $x + \sum_{i=1}^{n} i$. The summation $\sum_{i=1}^{n} i$ is another example of summing an arithmetic progression, which we can calculate by adding the first and last terms, multiplying by the number of terms, and dividing by 2. That is, $\sum_{i=1}^{n} i = (1+n)n/2$. Thus, $bar(x, n) = x + n(n+1)/2$.

Now, consider function `foo`, which adds to its argument `x` the sum

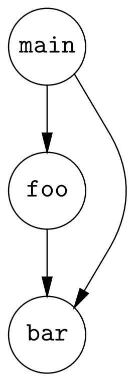


Fig. 3.22. Calling graph for Fig. 3.21.

Proofs and Program Understanding

The reader may notice that in our examination of the program in Fig. 3.21, we were able to understand what the program does, but we did not prove anything formally as we did in Chapter 2. However, there are many simple inductive proofs lurking just below the surface. For example, we need to prove by induction on the number of times around the loop of lines (4) and (5) that the value of `x`, just before we begin the iteration with value i for `i`, is the initial value of `x` plus $\sum_{j=1}^{i-1} j$. Note that if $i = 1$, then this sum consists of 0 terms and therefore has the value 0.

$$\sum_{i=1}^{n} bar(i, n)$$

From our understanding of `bar` we know that $bar(i, n) = i + n(n + 1)/2$. Thus `foo` adds to `x` the quantity $\sum_{i=1}^{n}(i+n(n+1)/2)$. We have another arithmetic progression to sum, and this one requires more algebraic manipulation. However, the reader may check that the quantity `foo` adds to its argument `x` is $(n^3 + 2n^2 + n)/2$.

Finally, let us consider the function `main`. We read n at line (1). At line (2) we apply `foo` to 0 and n. By our understanding of `foo`, the value of `foo(0,n)` at line (2) will be 0 plus $(n^3 + 2n^2 + n)/2$. At line (3), we print the value of `bar(foo(0,n),n)`, which, by our understanding of `bar` is the sum of $n(n + 1)/2$ and the current value of `foo(a,n)`. Thus, the value printed is $(n^3 + 3n^2 + 2n)/2$.

Now let us analyze the running time of the program in Fig. 3.21, working from `bar` to `foo` and then to `main`, as we did in Example 3.23. In this case, we shall take the value n to be the size of the input to all three functions. That is, even though we generally want to consider the "size" of all the arguments to a function, in this case the running time of the functions depends only on n.

To analyze `bar`, we note that line (5) takes $O(1)$ time. The for-loop of lines (4) and (5) iterates n times, and so the time for lines (4) and (5) is $O(n)$. Line (6) takes $O(1)$ time, and so the time for the block of lines (4) to (6) is $O(n)$.

Now we proceed to `foo`. The assignment in line (8) takes $O(1)$ plus the time of a call to `bar(i,n)`. That call, we already know, takes $O(n)$ time, and so the time for line (8) is $O(n)$. The for-loop of lines (7) and (8) iterates n times, and so we multiply the $O(n)$ for the body by n to get $O(n^2)$ as the running time of a call to `foo`.

Finally, we may analyze `main`. Line (1) takes $O(1)$ time. The call to `foo` at line (2) takes $O(n^2)$ time, as we just deduced. The print-statement of line (3) takes $O(1)$ plus the time for a call to `bar`. The latter takes $O(n)$ time, and so line (3) as a whole takes $O(1) + O(n)$ time. The total time for the block of lines (1) through (3) is therefore $O(1)+O(n^2)+O(1)+O(n)$. By the rule for sums, we can eliminate all but the second term, concluding that the function `main` takes $O(n^2)$ time. That is, the call to `foo` at line (2) is the dominant cost. ✦

EXERCISES

3.8.1: Show the claim in Example 3.23, that

$$\sum_{i=1}^{n}(i + n(n+1)/2) = (n^3 + 2n^2 + n)/2$$

3.8.2: Suppose that $prime(n)$ is a function call that takes time $O(\sqrt{n})$. Consider a function whose body is

```
if ( prime(n) )
    A;
else
    B;
```

Give a simple and tight big-oh upper bound on the running time of this function, as a function of n, on the assumption that

a) A takes $O(n)$ time and B takes $O(1)$ time
b) A and B both take $O(1)$ time

3.8.3: Consider a function whose body is

```
sum = 0;
for (i = 1; i <= f(n); i++)
    sum += i;
```

where $f(n)$ is a function call. Give a simple and tight big-oh upper bound on the running time of this function, as a function of n, on the assumption that

a) The running time of $f(n)$ is $O(n)$, and the value of $f(n)$ is $n!$
b) The running time of $f(n)$ is $O(n)$, and the value of $f(n)$ is n
c) The running time of $f(n)$ is $O(n^2)$, and the value of $f(n)$ is n
d) The running time of $f(n)$ is $O(1)$, and the value of $f(n)$ is 0

3.8.4: Draw the calling graph for the functions in the merge sort program from Section 2.8. Is the program recursive?

3.8.5*: Suppose that line (7) in the function `foo` of Fig. 3.21 were replaced by

```
for (i = 1; i <= bar(n,n); i++)
```

What would the running time of `main` be then?

3.9 Analyzing Recursive Functions

Determining the running time of a function that calls itself recursively requires more work than analyzing nonrecursive functions. The analysis for a recursive function requires that we associate with each function F in a program an unknown running time $T_F(n)$. This unknown function represents F's running time as a function of n, the size of F's arguments. We then establish an inductive definition, called a *recurrence relation* for $T_F(n)$, that relates $T_F(n)$ to functions of the form $T_G(k)$ for the other functions G in the program and their associated argument sizes k. If F is directly recursive, then one or more of the G's will be the same as F.

Recurrence relation

The value of $T_F(n)$ is normally established by an induction on the argument size n. Thus, it is necessary to pick a notion of argument size that guarantees functions are called with progressively smaller arguments as the recursion proceeds. The requirement is the same as what we encountered in Section 2.9, when we tried to prove statements about recursive programs. That should be no surprise, because a statement about the running time of a program is just one example of something that we might try to prove about a program.

Once we have a suitable notion of argument size, we can consider two cases:

1. The argument size is sufficiently small that no recursive calls will be made by F. This case corresponds to the basis in an inductive definition of $T_F(n)$.
2. For larger argument sizes, one or more recursive calls may occur. Note that whatever recursive calls F makes, whether to itself or to some other function G, will be made with smaller arguments. This case corresponds to the inductive step in the definition of $T_F(n)$.

The recurrence relation defining $T_F(n)$ is derived by examining the code for function F and doing the following:

a) For each call to a function G or use of a function G in an expression (note that G may be F), use $T_G(k)$ as the running time of the call, where k is the appropriate measure of the size of the arguments in the call.

b) Evaluate the running time of the body of function F, using the same techniques as in previous sections, but leaving terms like $T_G(k)$ as unknown functions, rather than concrete functions such as n^2. These terms cannot generally be combined with concrete functions using simplification tricks such as the summation rule. We must analyze F twice — once on the assumption that F's argument size n is sufficiently small that no recursive function calls are made, and once assuming that n is not that small. As a result, we obtain two expressions for the running time of F — one (the *basis expression*) serving as the basis of the recurrence relation for $T_F(n)$, and the other (the *induction expression*) serving as the inductive part.

Basis and induction expressions

c) In the resulting basis and induction expressions for the running time of F, replace big-oh terms like $O\big(f(n)\big)$ by a specific constant times the function involved — for example, $cf(n)$.

d) If a is a basis value for the input size, set $T_F(a)$ equal to the basis expression resulting from step (c) on the assumption that there are no recursive calls. Also, set $T_F(n)$ equal to the induction expression from (c) for the case where n is not a basis value.

```
            int fact(int n)
            {
(1)             if (n <= 1)
(2)                 return 1; /* basis */
                else
(3)                 return n*fact(n-1); /* induction */
            }
```

Fig. 3.23. Program to compute $n!$.

The running time of the entire function is determined by solving this recurrence relation. In Section 3.11, we shall give general techniques for solving recurrences of the kind that arise in the analysis of common recursive functions. For the moment, we solve these recurrences by ad hoc means.

✦ **Example 3.24.** Let us reconsider the recursive program from Section 2.7 to compute the factorial function; the code is shown in Fig. 3.23. Since there is only one function, `fact`, involved, we shall use $T(n)$ for the unknown running time of this function. We shall use n, the value of the argument, as the size of the argument. Clearly, recursive calls made by `fact` when the argument is n have a smaller argument, $n-1$ to be precise.

For the basis of the inductive definition of $T(n)$ we shall take $n = 1$, since no recursive call is made by `fact` when its argument is 1. With $n = 1$, the condition of line (1) is true, and so the call to `fact` executes lines (1) and (2). Each takes $O(1)$ time, and so the running time of `fact` in the basis case is $O(1)$. That is, $T(1)$ is $O(1)$.

Now, consider what happens when $n > 1$. The condition of line (1) is false, and so we execute only lines (1) and (3). Line (1) takes $O(1)$ time, and line (3) takes $O(1)$ for the multiplication and assignment, plus $T(n-1)$ for the recursive call to `fact`. That is, for $n > 1$, the running time of `fact` is $O(1) + T(n-1)$. We can thus define $T(n)$ by the following recurrence relation:

BASIS. $T(1) = O(1)$.

INDUCTION. $T(n) = O(1) + T(n-1)$, for $n > 1$.

We now invent constant symbols to stand for those constants hidden within the various big-oh expressions, as was suggested by rule (c) above. In this case, we can replace the $O(1)$ in the basis by some constant a, and the $O(1)$ in the induction by some constant b. These changes give us the following recurrence relation:

BASIS. $T(1) = a$.

INDUCTION. $T(n) = b + T(n-1)$, for $n > 1$.

Now we must solve this recurrence for $T(n)$. We can calculate the first few values easily. $T(1) = a$ by the basis. Thus, by the inductive rule, we have

$$T(2) = b + T(1) = a + b$$

Continuing to use the inductive rule, we get

$$T(3) = b + T(2) = b + (a + b) = a + 2b$$

Then

$$T(4) = b + T(3) = b + (a + 2b) = a + 3b$$

By this point, it should be no surprise if we guess that $T(n) = a + (n - 1)b$, for all $n \geq 1$. Indeed, computing some sample values, then guessing a solution, and finally proving our guess correct by an inductive proof is a common method of dealing with recurrences.

Repeated substitution

In this case, however, we can derive the solution directly by a method known as *repeated substitution.* First, let us make a substitution of variables, m for n, in the recursive equation, which now becomes

$$T(m) = b + T(m - 1), \text{ for } m > 1 \tag{3.3}$$

Now, we can substitute n, $n - 1$, $n - 2, \ldots, 2$ for m in equation (3.3) to get the sequence of equations

$$\begin{array}{rlcl} 1) & T(n) & = & b + T(n-1) \\ 2) & T(n-1) & = & b + T(n-2) \\ 3) & T(n-2) & = & b + T(n-3) \\ & & \cdots & \\ n-1) & T(2) & = & b + T(1) \end{array}$$

Next, we can use line (2) above to substitute for $T(n-1)$ in (1), to get the equation

$$T(n) = b + \big(b + T(n - 2)\big) = 2b + T(n - 2)$$

Now, we use line (3) to substitute for $T(n - 2)$ in the above to get

$$T(n) = 2b + \big(b + T(n - 3)\big) = 3b + T(n - 3)$$

We proceed in this manner, each time replacing $T(n - i)$ by $b + T(n - i - 1)$, until we get down to $T(1)$. At that point, we have the equation

$$T(n) = (n - 1)b + T(1)$$

We can then use the basis to replace $T(1)$ by a, and conclude that $T(n) = a+(n-1)b$.

To make this analysis more formal, we need to prove by induction our intuitive observations about what happens when we repeatedly substitute for $T(n-i)$. Thus we shall prove the following by induction on i:

STATEMENT $S(i)$: If $1 \leq i < n$, then $T(n) = ib + T(n - i)$.

BASIS. The basis is $i = 1$. $S(1)$ says that $T(n) = b+T(n-1)$. This is the inductive part of the definition of $T(n)$ and therefore known to be true.

INDUCTION. If $i \geq n - 1$, there is nothing to prove. The reason is that statement $S(i + 1)$ begins, "If $1 \leq i + 1 < n \cdots$," and when the condition of an if-statement is false, the statement is true regardless of what follows the "then." In this case, where $i \geq n - 1$, the condition $i + 1 < n$ must be false, so $S(i + 1)$ is true.

The hard part is when $i \leq n-2$. In that case, $S(i)$ says that $T(n) = ib+T(n-i)$. Since $i \leq n-2$, the argument of $T(n-i)$ is at least 2. Thus we can apply the inductive rule for T — that is, (3.3) with $n-i$ in place of m — to get the equation $T(n-i) = b + T(n-i-1)$. When we substitute $b + T(n-i-1)$ for $T(n-i)$ in the equation $T(n) = ib + T(n-i)$, we obtain $T(n) = ib + \big(b + T(n-i-1)\big)$, or regrouping terms,

$$T(n) = (i+1)b + T\big(n-(i+1)\big)$$

This equation is the statement $S(i+1)$, and we have now proved the induction step.

We have now shown that $T(n) = a + (n-1)b$. However, a and b are unknown constants. Thus, there is no point in presenting the solution this way. Rather, we can express $T(n)$ as a polynomial in n, namely, $bn + (a-b)$, and then replace terms by big-oh expressions, giving $O(n) + O(1)$. Using the summation rule, we can eliminate $O(1)$, which tells us that $T(n)$ is $O(n)$. That makes sense; it says that to compute $n!$, we make on the order of n calls to `fact` (the actual number is exactly n), each of which requires $O(1)$ time, excluding the time spent performing the recursive call to `fact`. ✦

EXERCISES

3.9.1: Set up a recurrence relation for the running time of the function `sum` mentioned in Exercise 2.9.2, as a function of the length of the list that is input to the program. Replace big-oh's by (unknown) constants, and try to solve your recurrence. What is the running time of `sum`?

3.9.2: Repeat Exercise 3.9.1 for the function `find0` from Exercise 2.9.3. What is a suitable size measure?

3.9.3*: Repeat Exercise 3.9.1 for the recursive selection sort program in Fig. 2.22 of Section 2.7. What is a suitable size measure?

Fibonacci numbers

3.9.4**: Repeat Exercise 3.9.1 for the function of Fig. 3.24, which computes the Fibonacci numbers. (The first two are 1, and each succeeding number is the sum of the previous two. The first seven Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13.) Note that the value of n is the appropriate size of an argument and that you need both 1 and 2 as basis cases.

```
int fibonacci(int n)
{
    if (n <= 2)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Fig. 3.24. C function computing the Fibonacci numbers.

3.9.5*: Write a recursive program to compute $gcd(i, j)$, the greatest common divisor of two integers i and j, as outlined in Exercise 2.7.8. Show that the running time of the program is $O(\log i)$. *Hint*: Show that after two calls we invoke $gcd(m, n)$ where $m \leq i/2$.

3.10 Analysis of Merge Sort

We shall now analyze the merge sort algorithm that we presented in Section 2.8. First, we show that the `merge` and `split` functions each take $O(n)$ time on lists of length n, and then we use these bounds to show that the `MergeSort` function takes $O(n \log n)$ time on lists of length n.

Analysis of the Merge Function

We begin with the analysis of the recursive function `merge`, whose code we repeat as Fig. 3.25. The appropriate notion of size n for the argument of `merge` is the sum of the lengths of the lists $list1$ and $list2$. Thus, we let $T(n)$ be the time taken by `merge` when the sum of the lengths of its argument lists is n. We shall take $n = 1$ to be the basis case, and so we must analyze Fig. 3.25 on the assumption that one of `list1` and `list2` is empty and the other has only one element. There are two cases:

1. If the test of line (1) — that is, $list1$ equals `NULL` — succeeds, then we return `list2`, which takes $O(1)$ time. Lines (2) through (7) are not executed. Thus, the entire function call takes $O(1)$ time to test the selection of line (1) and $O(1)$ time to perform the assignment on line (1), a total of $O(1)$ time.

2. If the test of line (1) fails, then $list1$ is not empty. Since we assume that the sum of the lengths of the lists is only 1, $list2$ must therefore be empty. Thus, the test on line (2) — namely, $list2$ equals `NULL` — must succeed. We then take $O(1)$ time to perform the test of line (1), $O(1)$ to perform the test of line (2), and $O(1)$ to return `list1` on line (2). Lines (3) through (7) are not executed. Again, we take only $O(1)$ time.

We conclude that in the basis case `merge` takes $O(1)$ time.

Now let us consider the inductive case, where the sum of the list lengths is greater than 1. Of course, even if the sum of the lengths is 2 or more, one of the lists could still be empty. Thus, any of the four cases represented by the nested selection statements could be taken. The structure tree for the program of Fig. 3.25 is shown in Fig. 3.26. We can analyze the program by working from the bottom, up the structure tree.

The innermost selection begins with the "if" on line (3), where we test which list has the smaller first element and then either execute lines (4) and (5) or execute lines (6) and (7). The condition of line (3) takes $O(1)$ time to evaluate. Line (5) takes $O(1)$ time to evaluate, and line (4) takes $O(1)$ time plus $T(n-1)$ time for the recursive call to `merge`. Note that $n - 1$ is the argument size for the recursive call, since we have eliminated exactly one element from one of the lists and left the other list as it was. Thus, the block of lines (4) and (5) takes $O(1) + T(n-1)$ time.

The analysis for the else-part in lines (6) and (7) is exactly the same: line (7) takes $O(1)$ time and line (6) takes $O(1) + T(n-1)$ time. Thus, when we take

```
        LIST merge(LIST list1, LIST list2)
        {
(1)         if (list1 == NULL) return list2;
(2)         else if (list2 == NULL) return list1;
(3)         else if (list1->element <= list2->element) {
(4)             list1->next = merge(list1->next, list2);
(5)             return list1;
            }
            else { /* list2 has smaller first element */
(6)             list2->next = merge(list1, list2->next);
(7)             return list2;
            }
        }
```

Fig. 3.25. The function `merge`.

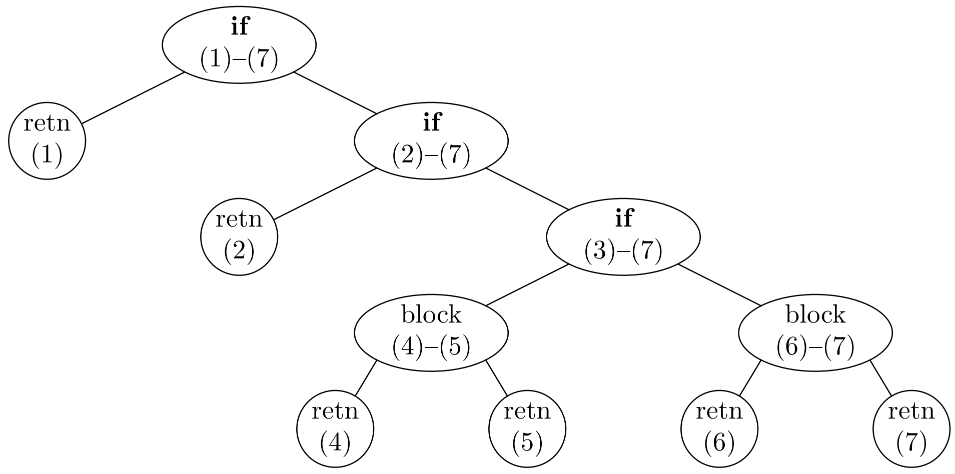


Fig. 3.26. Structure of `merge`.

the maximum of the running times of the if- and else-parts, we find these times to be the same. The $O(1)$ for the test of the condition can be neglected, and so we conclude that the running time of the innermost selection is $O(1) + T(n-1)$.

Now we proceed to the selection beginning on line (2), where we test whether *list*2 equals `NULL`. The time for testing the condition is $O(1)$, and the time for the if-part, which is the return on line (2), is also $O(1)$. However, the else-part is the selection statement of lines (3) through (7), which we just determined takes $O(1) + T(n-1)$ time. Thus, the time for the selection of lines (2) through (7) is

$$O(1) + \max\big(O(1), O(1) + T(n-1)\big)$$

The second term of the maximum dominates the first and also dominates the $O(1)$ contributed by the test of the condition. Thus, the time for the if-statement beginning at line (2) is also $O(1) + T(n-1)$.

Finally, we perform the same analysis for the outermost if-statement. Essentially, the dominant time is the else-part, which consists of lines (2) through (7).

A Common Form of Recursion

Many of the simplest recursive functions, such as `fact` and `merge`, perform some operation that takes $O(1)$ time and then make a recursive call to themselves on an argument one size smaller. Assuming the basis case takes $O(1)$ time, we see that such a function always leads to a recurrence relation $T(n) = O(1) + T(n-1)$. The solution for $T(n)$ is $O(n)$, or linear in the size of the argument. In Section 3.11 we shall see some generalizations of this principle.

That is, the time for the cases in which there is a recursive call, lines (4) and (5) or lines (6) and (7), dominates the time for the cases in which there is no recursive call, represented by lines (1) and (2), and also dominates the time for all three tests on lines (1), (2), and (3). Thus, the time for the function `merge`, when $n > 1$, is bounded above by $O(1) + T(n-1)$. We therefore have the following recurrence relation for defining $T(n)$:

BASIS. $T(1) = O(1)$.

INDUCTION. $T(n) = O(1) + T(n-1)$, for $n > 1$.

These equations are exactly the same as those derived for the function `fact` in Example 3.24. Thus, the solution is the same and we can conclude that $T(n)$ is $O(n)$. That result makes intuitive sense, since `merge` works by eliminating an element from one of the lists, taking $O(1)$ time to do so, and then calling itself recursively on the remaining lists. It follows that the number of recursive calls will be no greater than the sum of the lengths of the lists. Since each call takes $O(1)$ time, exclusive of the time taken by its recursive call, we expect the time for `merge` to be $O(n)$.

```
          LIST split(LIST list)
          {
              LIST pSecondCell;

 (1)          if (list == NULL) return NULL;
 (2)          else if (list->next == NULL) return NULL;
              else { /* there are at least two cells */
 (3)              pSecondCell = list->next;
 (4)              list->next = pSecondCell->next;
 (5)              pSecondCell->next = split(pSecondCell->next);
 (6)              return pSecondCell;
              }
          }
```

Fig. 3.27. The function `split`.

Analysis of the Split Function

Now let us consider the `split` function, which we reproduce as Fig. 3.27. The analysis is quite similar to that for `merge`. We let the size n of the argument be the length of the list, and we here use $T(n)$ for the time taken by `split` on a list of length n.

For the basis, we take both $n = 0$ and $n = 1$. If $n = 0$ — that is, *list* is empty — the test of line (1) succeeds and we return `NULL` on line (1). Lines (2) through (6) are not executed, and we therefore take $O(1)$ time. If $n = 1$, that is, *list* is a single element, the test of line (1) fails, but the test of line (2) succeeds. We therefore return `NULL` on line (2) and do not execute lines (3) through (6). Again, only $O(1)$ time is needed for the two tests and one return statement.

For the induction, $n > 1$, there is a three-way selection branch, similar to the four-way branch we encountered in `merge`. To save time in analysis, we may observe — as we eventually concluded for `merge` — that we take $O(1)$ time to do one or both of the selection tests of lines (1) and (2). Also, in the cases in which one of these two tests is true, where we return on line (1) or (2), the additional time is only $O(1)$. The dominant time is the case in which both tests fail, that is, in which the list is of length at least 2; in this case we execute the statements of lines (3) through (6). All but the recursive call in line (5) contributes $O(1)$ time. The recursive call takes $T(n-2)$ time, since the argument list is the original value of `list`, missing its first two elements (to see why, refer to the material in Section 2.8, especially the diagram of Fig. 2.28). Thus, in the inductive case, $T(n)$ is $O(1) + T(n-2)$.

We may set up the following recurrence relation:

BASIS. $T(0) = O(1)$ and $T(1) = O(1)$.

INDUCTION. $T(n) = O(1) + T(n-2)$, for $n > 1$.

As in Example 3.24, we must next invent constants to represent the constants of proportionality hidden by the $O(1)$'s. We shall let a and b be the constants represented by $O(1)$ in the basis for the values of $T(0)$ and $T(1)$, respectively, and we shall use c for the constant represented by $O(1)$ in the inductive step. Thus, we may rewrite the recursive definition as

BASIS. $T(0) = a$ and $T(1) = b$.

INDUCTION. $T(n) = c + T(n-2)$ for $n \geq 2$.

Let us evaluate the first few values of $T(n)$. Evidently $T(0) = a$ and $T(1) = b$ by the basis. We may use the inductive step to deduce

$$\begin{array}{lllllll}
T(2) & = & c + T(0) & = & a + c & & \\
T(3) & = & c + T(1) & = & b + c & & \\
T(4) & = & c + T(2) & = & c + (a + c) & = & a + 2c \\
T(5) & = & c + T(3) & = & c + (b + c) & = & b + 2c \\
T(6) & = & c + T(4) & = & c + (a + 2c) & = & a + 3c
\end{array}$$

The calculation of $T(n)$ is really two separate calculations, one for odd n and the other for even n. For even n, we get $T(n) = a + cn/2$. That makes sense, since with an even-length list, we eliminate two elements, taking time c to do so, and after $n/2$ recursive calls, we are left with an empty list, on which we make no more recursive calls and take a time.

On an odd-length list, we again eliminate two elements, taking time c to do so. After $(n-1)/2$ calls, we are down to a list of length 1, for which time b is required. Thus, the time for odd-length lists will be $b + c(n-1)/2$.

The inductive proofs of these observations closely parallel the proof in Example 3.24. That is, we prove the following:

STATEMENT $S(i)$: If $1 \leq i \leq n/2$, then $T(n) = ic + T(n-2i)$.

In the proof, we use the inductive rule in the definition of $T(n)$, which we can rewrite with argument m as

$$T(m) = c + T(m-2), \text{ for } m \geq 2 \tag{3.4}$$

We may then prove $S(i)$ by induction as follows:

BASIS. The basis, $i = 1$, is (3.4) with n in place of m.

INDUCTION. Because $S(i)$ has an if-then form, $S(i+1)$ is always true if $i \geq n/2$. Thus, the inductive step — that $S(i)$ implies $S(i+1)$ — requires no proof if $i \geq n/2$.

The hard case occurs when $1 \leq i < n/2$. In this situation, suppose that the inductive hypothesis $S(i)$ is true; $T(n) = ic + T(n-2i)$. We substitute $n - 2i$ for m in (3.4), giving us

$$T(n-2i) = c + T(n-2i-2)$$

If we substitute for $T(n-2i)$ in $S(i)$, we get

$$T(n) = ic + \big(c + T(n-2i-2)\big)$$

If we then group terms, we get

$$T(n) = (i+1)c + T\big(n - 2(i+1)\big)$$

which is the statement $S(i+1)$. We have thus proved the inductive step, and we conclude $T(n) = ic + T(n-2i)$.

Now if n is even, let $i = n/2$. Then $S(n/2)$ says that $T(n) = cn/2 + T(0)$, which is $a + cn/2$. If n is odd, let $i = (n-1)/2$. $S((n-1)/2)$ tells us that $T(n)$ is

$$c(n-1)/2 + T(1)$$

which equals $b + c(n-1)/2$ since $T(1) = b$.

Finally, we must convert to big-oh notation the constants a, b, and c, which represent compiler- and machine-specific quantities. Both the polynomials $a + cn/2$ and $b + c(n-1)/2$ have high-order terms proportional to n. Thus, the question whether n is odd or even is actually irrelevant; the running time of `split` is $O(n)$ in either case. Again, that is the intuitively correct answer, since on a list of length n, `split` makes about $n/2$ recursive calls, each taking $O(1)$ time.

```
        LIST MergeSort(LIST list)
        {
            LIST SecondList;

(1)         if (list == NULL) return NULL;
(2)         else if (list->next == NULL) return list;
            else {
                /* at least two elements on list */
(3)             SecondList = split(list);
(4)             return merge(MergeSort(list), MergeSort(SecondList));
            }
        }
```

Fig. 3.28. The merge sort algorithm.

The Function MergeSort

Finally, we come to the function `MergeSort`, which is reproduced in Fig. 3.28. The appropriate measure n of argument size is again the length of the list to be sorted. Here, we shall use $T(n)$ as the running time of `MergeSort` on a list of length n.

We take $n = 1$ as the basis case and $n > 1$ as the inductive case, where recursive calls are made. If we examine `MergeSort`, we observe that, unless we call `MergeSort` from another function with an argument that is an empty list, then there is no way to get a call with an empty list as argument. The reason is that we execute line (4) only when `list` has at least two elements, in which case the lists that result from a split will have at least one element each. Thus, we can ignore the case $n = 0$ and start our induction at $n = 1$.

BASIS. If `list` consists of a single element, then we execute lines (1) and (2), but none of the other code. Thus, in the basis case, $T(1)$ is $O(1)$.

INDUCTION. In the inductive case, the tests of lines (1) and (2) both fail, and so we execute the block of lines (3) and (4). To make things simpler, let us assume that n is a power of 2. The reason it helps to make this assumption is that when n is even, the split of the list is into two pieces of length exactly $n/2$. Moreover, if n is a power of 2, then $n/2$ is also a power of 2, and the divisions by 2 are all into equal-sized pieces until we get down to pieces of 1 element each, at which time the recursion ends. The time spent by `MergeSort` when $n > 1$ is the sum of the following terms:

1. $O(1)$ for the two tests
2. $O(1) + O(n)$ for the assignment and call to `split` on line (3)
3. $T(n/2)$ for the first recursive call to `MergeSort` on line (4)
4. $T(n/2)$ for the second recursive call to `MergeSort` on line (4)
5. $O(n)$ for the call to `merge` on line (4)
6. $O(1)$ for the return on line (4).

Inductions that Skip Some Values

The reader should not be concerned by the new kind of induction that is involved in the analysis of `MergeSort`, where we skip over all but the powers of 2 in our proof. In general, if $i_1, i_2, \ldots$ is a sequence of integers about which we want to prove a statement S, we can show $S(i_1)$ as a basis and then show for the induction that $S(i_j)$ implies $S(i_{j+1})$, for all j. That is an ordinary induction if we think of it as an induction on j. More precisely, define the statement S' by $S'(j) = S(i_j)$. Then we prove $S'(j)$ by induction on j. For the case at hand, $i_1 = 1$, $i_2 = 2$, $i_3 = 4$, and in general, $i_j = 2^{j-1}$.

Incidentally, note that $T(n)$, the running time of `MergeSort`, surely does not decrease as n increases. Thus, showing that $T(n)$ is $O(n \log n)$ for n equal to a power of 2 also shows that $T(n)$ is $O(n \log n)$ for all n.

If we add these terms, and drop the $O(1)$'s in favor of the larger $O(n)$'s that come from the calls to `split` and `merge`, we get the bound $2T(n/2) + O(n)$ for the time spent by `MergeSort` in the inductive case. We thus have the recurrence relation:

BASIS. $T(1) = O(1)$.

INDUCTION. $T(n) = 2T(n/2) + O(n)$, where n is a power of 2 and greater than 1.

Our next step is to replace the big-oh expressions by functions with concrete constants. We shall replace the $O(1)$ in the basis by constant a, and the $O(n)$ in the inductive step by bn, for some constant b. Our recurrence relation thus becomes

BASIS. $T(1) = a$.

INDUCTION. $T(n) = 2T(n/2) + bn$, where n is a power of 2 and greater than 1.

This recurrence is rather more complicated than the ones studied so far, but we can apply the same techniques. First, let us explore the values of $T(n)$ for some small n's. The basis tells us that $T(1) = a$. Then the inductive step says

$$\begin{array}{lllllll}
T(2) & = & 2T(1) + 2b & & & = & 2a + 2b \\
T(4) & = & 2T(2) + 4b & = & 2(2a + 2b) + 4b & = & 4a + 8b \\
T(8) & = & 2T(4) + 8b & = & 2(4a + 8b) + 8b & = & 8a + 24b \\
T(16) & = & 2T(8) + 16b & = & 2(8a + 24b) + 16b & = & 16a + 64b
\end{array}$$

It may not be easy to see what is going on. Evidently, the coefficient of a keeps pace with the value of n; that is, $T(n)$ is n times a plus some number of b's. But the coefficient of b grows faster than any multiple of n. The relationship between n and the coefficient of b is summarized as follows:

Value of n	2	4	8	16
Coefficient of b	2	8	24	64
Ratio	1	2	3	4

The ratio is the coefficient of b divided by the value of n. Thus, it appears that the coefficient of b is n times another factor that grows by 1 each time n doubles. In particular, we can see that this ratio is $\log_2 n$, because $\log_2 2 = 1$, $\log_2 4 = 2$, $\log_2 8 = 3$, and $\log_2 16 = 4$. It is thus reasonable to conjecture that the solution to our recurrence relation is $T(n) = an + bn \log_2 n$, at least for n a power of 2. We shall see that this formula is correct.

To get a solution to the recurrence, let us follow the same strategy as for previous examples. We write the inductive rule with argument m, as

$$T(m) = 2T(m/2) + bm, \text{ for } m \text{ a power of 2 and } m > 1 \tag{3.5}$$

We then start with $T(n)$ and use (3.5) to replace $T(n)$ by an expression involving smaller values of the argument; in this case, the replacing expression involves $T(n/2)$. That is, we begin with

$$T(n) = 2T(n/2) + bn \tag{3.6}$$

Next, we use (3.5), with $n/2$ in place of m, to get a replacement for $T(n/2)$ in (3.6). That is, (3.5) says that $T(n/2) = 2T(n/4) + bn/2$, and we can replace (3.6) by

$$T(n) = 2\bigl(2T(n/4) + bn/2\bigr) + bn = 4T(n/4) + 2bn$$

Then, we can replace $T(n/4)$ by $2T(n/8) + bn/4$; the justification is (3.5) with $n/4$ in place of m. That gives us

$$T(n) = 4\bigl(2T(n/8) + bn/4\bigr) + 2bn = 8T(n/8) + 3bn$$

The statement that we shall prove by induction on i is

STATEMENT $S(i)$: If $1 \le i \le \log_2 n$, then $T(n) = 2^i T(n/2^i) + ibn$.

BASIS. For $i = 1$, the statement $S(1)$ says that $T(n) = 2T(n/2) + bn$. This equality is the inductive rule in the definition of $T(n)$, the running time of merge and sort, so we know that the basis holds.

INDUCTION. As in similar inductions where the inductive hypothesis is of the if-then form, the inductive step holds whenever i is outside the assumed range; here, $i \ge \log_2 n$ is the simple case, where $S(i+1)$ is seen to hold.

For the hard part, suppose that $i < \log_2 n$. Also, assume the inductive hypothesis $S(i)$; that is, $T(n) = 2^i T(n/2^i) + ibn$. Substitute $n/2^i$ for m in (3.5) to get

$$T(n/2^i) = 2T(n/2^{i+1}) + bn/2^i \tag{3.7}$$

Substitute the right side of (3.7) for $T(n/2^i)$ in $S(i)$ to get

$$\begin{aligned} T(n) &= 2^i\bigl(2T(n/2^{i+1}) + bn/2^i\bigr) + ibn \\ &= 2^{i+1}T(n/2^{i+1}) + bn + ibn \\ &= 2^{i+1}T(n/2^{i+1}) + (i+1)bn \end{aligned}$$

The last equality is the statement $S(i+1)$, and so we have proved the inductive step.

We conclude that the equality $S(i)$ — that is, $T(n) = 2^i T(n/2^i) + ibn$ — holds for any i between 1 and $\log_2 n$. Now consider the formula $S(\log_2 n)$, that is,

$$T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + (\log_2 n)bn$$

We know that $2^{\log_2 n} = n$ (recall that the definition of $\log_2 n$ is the power to which we must raise 2 to equal n). Also, $n/2^{\log_2 n} = 1$. Thus, $S(\log_2 n)$ can be written

$$T(n) = nT(1) + bn \log_2 n$$

We also know that $T(1) = a$, by the basis of the definition of T. Thus,

$$T(n) = an + bn \log_2 n$$

After this analysis, we must replace the constants a and b by big-oh expressions. That is, $T(n)$ is $O(n) + O(n \log n)$.[8] Since n grows more slowly than $n \log n$, we may neglect the $O(n)$ term and say that $T(n)$ is $O(n \log n)$. That is, merge sort is an $O(n \log n)$-time algorithm. Remember that selection sort was shown to take $O(n^2)$ time. Although strictly speaking, $O(n^2)$ is only an upper bound, it is in fact the tightest simple bound for selection sort. Thus, we can be sure that, as n gets large, merge sort will always run faster than selection sort. In practice, merge sort is faster than selection sort for n's larger than a few dozen.

EXERCISES

3.10.1: Draw structure trees for the functions

a) `split`
b) `MergeSort`

Indicate the running time for each node of the trees.

3.10.2*: Define a function k-mergesort that splits a list into k pieces, sorts each piece, and then merges the result.

a) What is the running time of k-mergesort as a function of k and n?
b)**What value of k gives the fastest algorithm (as a function of n)? This problem requires that you estimate the running times sufficiently precisely that you can distinguish constant factors. Since you cannot be that precise in practice, for the reasons we discussed at the beginning of the chapter, you really need to examine how the running time from (a) varies with k and get an approximate minimum.

3.11 Solving Recurrence Relations

There are many techniques for solving recurrence relations. In this section, we shall discuss two such approaches. The first, which we have already seen, is repeatedly substituting the inductive rule into itself until we get a relationship between $T(n)$ and $T(1)$ or — if 1 is not the basis — between $T(n)$ and $T(i)$ for some i that is covered by the basis. The second method we introduce is guessing a solution and checking its correctness by substituting into the basis and the inductive rules.

[8] Remember that inside a big-oh expression, we do not have to specify the base of a logarithm, because logarithms to all bases are the same, to within a constant factor.

In the previous two sections, we have solved exactly for $T(n)$. However, since $T(n)$ is really a big-oh upper bound on the exact running time, it is sufficient to find a tight upper bound on $T(n)$. Thus, especially for the "guess-and-check" approach, we require only that the solution be an upper bound on the true solution to the recurrence.

Solving Recurrences by Repeated Substitution

Probably the simplest form of recurrence that we encounter in practice is that of Example 3.24:

BASIS. $T(1) = a$.

INDUCTION. $T(n) = T(n-1) + b$, for $n > 1$.

We can generalize this form slightly if we allow the addition of some function $g(n)$ in place of the constant b in the induction. We can write this form as

BASIS. $T(1) = a$.

INDUCTION. $T(n) = T(n-1) + g(n)$, for $n > 1$.

This form arises whenever we have a recursive function that takes time $g(n)$ and then calls itself with an argument one smaller than the argument with which the current function call was made. Examples are the factorial function of Example 3.24, the function `merge` of Section 3.10, and the recursive selection sort of Section 2.7. In the first two of these functions, $g(n)$ is a constant, and in the third it is linear in n. The function `split` of Section 3.10 is almost of this form; it calls itself with an argument that is smaller by 2. We shall see that this difference is unimportant.

Let us solve this recurrence by repeated substitution. As in Example 3.24, we first write the inductive rule with the argument m, as

$$T(m) = T(m-1) + g(m)$$

and then proceed to substitute for T repeatedly in the right side of the original inductive rule. Doing this, we get the sequence of expressions

$$\begin{aligned} T(n) &= T(n-1) + g(n) \\ &= T(n-2) + g(n-1) + g(n) \\ &= T(n-3) + g(n-2) + g(n-1) + g(n) \\ &\cdots \\ &= T(n-i) + g(n-i+1) + g(n-i+2) + \cdots + g(n-1) + g(n) \end{aligned}$$

Using the technique in Example 3.24, we can prove by induction on i, for $i = 1, 2, \ldots, n-1$, that

$$T(n) = T(n-i) + \sum_{j=0}^{i-1} g(n-j)$$

We want to pick a value for i so that $T(n-i)$ is covered by the basis; thus, we pick $i = n-1$. Since $T(1) = a$, we have $T(n) = a + \sum_{j=0}^{n-2} g(n-j)$. Put another way, $T(n)$ is the constant a plus the sum of all the values of g from 2 to n, or $a + g(2) + g(3) + \cdots + g(n)$. Unless all the $g(j)$'s are 0, the a term will not matter when we convert this expression to a big-oh expression, and so we generally just need the sum of the $g(j)$'s.

✦ **Example 3.25.** Consider the recursive selection sort function of Fig. 2.22, the body of which we reproduce as Fig. 3.29. If we let $T(m)$ be the running time of the function `SelectionSort` when given an array of m elements to sort (that is, when the value of its argument i is $n-m$), we can develop a recurrence relation for $T(m)$ as follows. First, the basis case is $m = 1$. Here, only line (1) is executed, taking $O(1)$ time.

```
(1)         if (i < n-1) {
(2)                small = i;
(3)                for (j = i+1; j < n; j++)
(4)                    if (A[j] < A[small])
(5)                        small = j;
(6)                temp = A[small];
(7)                A[small] = A[i];
(8)                A[i] = temp;
(9)                recSS(A, i+1, n);
               }
       }
```

Fig. 3.29. Recursive selection sort.

For the inductive case, $m > 1$, we execute the test of line (1) and the assignments of lines (2), (6), (7), and (8), all of which take $O(1)$ time. The for-loop of lines (3) to (5) takes $O(n-i)$ time, or $O(m)$ time, as we discussed in connection with the iterative selection sort program in Example 3.17. To review why, note that the body, lines (4) and (5), takes $O(1)$ time, and we go $m-1$ times around the loop. Thus, the time of the for-loop dominates lines (1) through (8), and we can write $T(m)$, the time of the entire function, as $T(m-1) + O(m)$. The second term, $O(m)$, covers lines (1) through (8), and the $T(m-1)$ term is the time for line (9), the recursive call. If we replace the hidden constant factors in the big-oh expressions by concrete constants, we get the recurrence relation

BASIS. $T(1) = a$.

INDUCTION. $T(m) = T(m-1) + bm$, for $m > 1$.

This recurrence is of the form we studied, with $g(m) = bm$. That is, the solution is

$$\begin{aligned} T(m) &= a + \sum_{j=0}^{m-2} b(m-j) \\ &= a + 2b + 3b + \cdots + mb \\ &= a + b(m-1)(m+2)/2 \end{aligned}$$

Thus, $T(m)$ is $O(m^2)$. Since we are interested in the running time of function `SelectionSort` on the entire array of length n, that is, when called with $i = 1$, we need the expression for $T(n)$ and find that it is $O(n^2)$. Thus, the recursive version of selection sort is quadratic, just like the iterative version. ✦

Another common form of recurrence generalizes the recurrence we derived for `MergeSort` in the previous section:

BASIS. $T(1) = a$.

INDUCTION. $T(n) = 2T(n/2) + g(n)$, for n a power of 2 and greater than 1.

This is the recurrence for a recursive algorithm that solves a problem of size n by subdividing it into two subproblems, each of size $n/2$. Here $g(n)$ is the amount of time taken to create the subproblems and combine the solutions. For example, `MergeSort` divides a problem of size n into two problems of size $n/2$. The function $g(n)$ is bn for some constant b, since the time taken by `MergeSort` exclusive of recursive calls to itself is $O(n)$, principally for the `split` and `merge` algorithms.

To solve this recurrence, we substitute for T on the right side. Here we assume that $n = 2^k$ for some k. The recursive equation can be written with m as its argument: $T(m) = 2T(m/2) + g(m)$. If we substitute $n/2^i$ for m, we get

$$T(n/2^i) = 2T(n/2^{i+1}) + g(n/2^i) \tag{3.8}$$

If we start with the inductive rule and proceed to substitute for T using (3.8) with progressively greater values of i, we find

$$\begin{aligned} T(n) &= 2T(n/2) + g(n) \\ &= 2\big(2T(n/2^2) + g(n/2)\big) + g(n) \\ &= 2^2T(n/2^2) + 2g(n/2) + g(n) \\ &= 2^2\big(2T(n/2^3) + g(n/2^2)\big) + 2g(n/2) + g(n) \\ &= 2^3T(n/2^3) + 2^2g(n/2^2) + 2g(n/2) + g(n) \\ &\cdots \\ &= 2^iT(n/2^i) + \sum_{j=0}^{i-1} 2^j g(n/2^j) \end{aligned}$$

If $n = 2^k$, we know that $T(n/2^k) = T(1) = a$. Thus, when $i = k$, that is, when $i = \log_2 n$, we obtain the solution

$$T(n) = an + \sum_{j=0}^{(\log_2 n)-1} 2^j g(n/2^j) \tag{3.9}$$

to our recurrence.

Intuitively, the first term of (3.9) represents the contribution of the basis value a. That is, there are n calls to the recursive function with an argument of size 1. The summation is the contribution of the recursion, and it represents the work performed by all the calls with argument size greater than 1.

Figure 3.30 suggests the accumulation of time during the execution of `MergeSort`. It represents the time to sort eight elements. The first row represents the work on the outermost call, involving all eight elements; the second row represents the work of the two calls with four elements each; and the third row is the four calls with two elements each. Finally, the bottom row represents the eight calls to `MergeSort` with a list of length one. In general, if there are n elements in the original unsorted list, there will be $\log_2 n$ levels at which bn work is done by calls to `MergeSort` that result in other calls. The accumulated time of all these calls is thus $bn \log_2 n$. There will be one level at which calls are made that do not result in further calls, and an is the total time spent by these calls. Note that the first $\log_2 n$ levels represent the terms of the summation in (3.9) and the lowest level represents the term an.

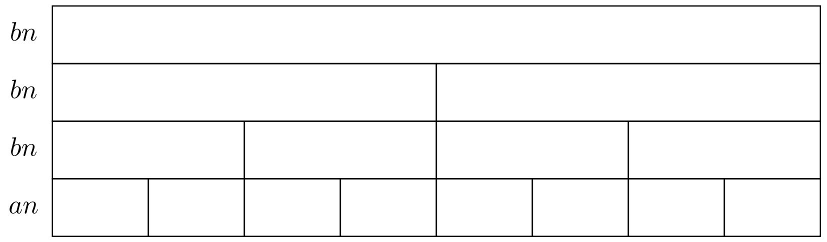


Fig. 3.30. Time spent by the calls to `mergesort`.

✦ **Example 3.26.** In the case of `MergeSort`, the function $g(n)$ is bn for some constant b. The solution (3.9) with these parameters is therefore

$$\begin{aligned} T(n) &= an + \sum_{j=0}^{(\log_2 n)-1} 2^j bn/2^j \\ &= an + bn \sum_{j=0}^{(\log_2 n)-1} 1 \\ &= an + bn \log n \end{aligned}$$

The last equality comes from the fact there are $\log_2 n$ terms in the sum and each term is 1. Thus, when $g(n)$ is linear, the solution to (3.9) is $O(n \log n)$. ✦

Solving Recurrences by Guessing a Solution

Another useful approach to solving recurrences is to guess a solution $f(n)$ and then use the recurrence to show that $T(n) \leq f(n)$. That may not give the exact value of $T(n)$, but if it gives a tight upper bound, we are satisfied. Often we guess only the functional form of $f(n)$, leaving some parameters unspecified; for example, we may guess that $f(n) = an^b$, for some a and b. The values of the parameters will be forced, as we try to prove $T(n) \leq f(n)$ for all n.

Although we may consider it bizarre to imagine that we can guess solutions accurately, we can frequently deduce the high-order term by looking at the values of $T(n)$ for a few small values of n. We then throw in some lower-order terms as well, and see whether their coefficients turn out to be nonzero.[9]

✦ **Example 3.27.** Let us again examine the `MergeSort` recurrence relation from the previous section, which we wrote as

BASIS. $T(1) = a$.

INDUCTION. $T(n) = 2T(n/2) + bn$, for n a power of 2 and greater than 1.

We are going to guess that an upper bound to $T(n)$ is $f(n) = cn\log_2 n + d$ for some constants c and d. Recall that this form is not exactly right; in the previous example we derived the solution and saw that it had an $O(n\log n)$ term with an $O(n)$ term, rather than with a constant term. However, this guess turns out to be good enough to prove an $O(n\log n)$ upper bound on $T(n)$.

We shall use complete induction on n to prove the following, for some constants c and d:

STATEMENT $S(n)$: If n is a power of 2 and $n \geq 1$, then $T(n) \leq f(n)$, where $f(n)$ is the function $cn\log_2 n + d$.

BASIS. When $n = 1$, $T(1) \leq f(1)$ provided $a \leq d$. The reason is that the $cn\log_2 n$ term of $f(n)$ is 0 when $n = 1$, so that $f(1) = d$, and it is given that $T(1) = a$.

INDUCTION. Let us assume $S(i)$ for all $i < n$, and prove $S(n)$ for some $n > 1$.[10] If n is not a power of 2, there is nothing to prove, since the if-portion of the if-then statement $S(n)$ is not true. Thus, consider the hard case, in which n is a power of 2. We may assume $S(n/2)$, that is,

$$T(n/2) \leq (cn/2)\log_2(n/2) + d$$

because it is part of the inductive hypothesis. For the inductive step, we need to show that

$$T(n) \leq f(n) = cn\log_2 n + d$$

When $n \geq 2$, the inductive part of the definition of $T(n)$ tells us that

$$T(n) \leq 2T(n/2) + bn$$

Using the inductive hypothesis to bound $T(n/2)$, we have

$$T(n) \leq 2[c(n/2)\log_2(n/2) + d] + bn$$

[9] If it is any comfort, the theory of differential equations, which in many ways resembles the theory of recurrence equations, also relies on known solutions to equations of a few common forms and then educated guessing to solve other equations.

[10] In most complete inductions we assume $S(i)$ for i up to n and prove $S(n+1)$. In this case it is notationally simpler to prove $S(n)$ from $S(i)$, for $i < n$, which amounts to the same thing.

Manipulating Inequalities

In Example 3.27 we derive one inequality, $T(n) \leq cn\log_2 n + d$, from another,

$$T(n) \leq cn\log_2 n + (b-c)n + 2d$$

Our method was to find an "excess" and requiring it to be at most 0. The general principle is that if we have an inequality $A \leq B + E$, and if we want to show that $A \leq B$, then it is sufficient to show that $E \leq 0$. In Example 3.27, A is $T(n)$, B is $cn\log_2 n + d$, and E, the excess, is $(b-c)n + d$.

Since $\log_2(n/2) = \log_2 n - \log_2 2 = \log_2 n - 1$, we may simplify this expression to

$$T(n) \leq cn\log_2 n + (b-c)n + 2d \tag{3.10}$$

We now show that $T(n) \leq cn\log_2 n + d$, provided that the excess over $cn\log_2 n + d$ on the right side of (3.10) is at most 0; that is, $(b-c)n + d \leq 0$. Since $n > 1$, this inequality is true when $d \geq 0$ and $b - c \leq -d$.

We now have three constraints for $f(n) = cn\log n + d$ to be an upper bound on $T(n)$:

1. The constraint $a \leq d$ comes from the basis.
2. $d \geq 0$ comes from the induction, but since we know that $a > 0$, this inequality is superseded by (1).
3. $b - c \leq -d$, or $c \geq b + d$, also comes from the induction.

These constraints are obviously satisfied if we let $d = a$ and $c = a + b$. We have now shown by induction on n that for all $n \geq 1$ and a power of 2,

$$T(n) \leq (a+b)n\log_2 n + a$$

This argument shows that $T(n)$ is $O(n\log n)$, that is, that $T(n)$ does not grow any faster than $n\log n$. However, the bound $(a+b)n\log_2 n + a$ that we obtained is slightly larger than the exact answer that we obtained in Example 3.26, which was $bn\log_2 n + an$. At least we were successful in obtaining a bound. Had we taken the simpler guess of $f(n) = cn\log_2 n$, we would have failed, because there is no value of c that can make $f(1) \geq a$. The reason is that $c \times 1 \times \log_2 1 = 0$, so that $f(1) = 0$. If $a > 0$, we evidently cannot make $f(1) \geq a$. ✦

✦ **Example 3.28.** Now let us consider a recurrence relation that we shall encounter later in the book:

BASIS. $G(1) = 3$.

INDUCTION. $G(n) = (2^{n/2} + 1)G(n/2)$, for $n > 1$.

This recurrence has actual numbers, like 3, instead of symbolic constants like a. In Chapter 13, we shall use recurrences such as this to count the number of gates in a circuit, and gates can be counted exactly, without needing the big-oh notation to

Summary of Solutions

In the table below, we list the solutions to some of the most common recurrence relations, including some we have not covered in this section. In each case, we assume that the basis equation is $T(1) = a$ and that $k \geq 0$.

INDUCTIVE EQUATION	$T(n)$
$T(n) = T(n-1) + bn^k$	$O(n^{k+1})$
$T(n) = cT(n-1) + bn^k$, for $c > 1$	$O(c^n)$
$T(n) = cT(n/d) + bn^k$, for $c > d^k$	$O(n^{\log_d c})$
$T(n) = cT(n/d) + bn^k$, for $c < d^k$	$O(n^k)$
$T(n) = cT(n/d) + bn^k$, for $c = d^k$	$O(n^k \log n)$

All the above also hold if bn^k is replaced by any kth-degree polynomial.

hide unknowable constant factors.

If we think about a solution by repeated substitution, we might see that we are going to make $\log_2 n - 1$ substitutions before $G(n)$ is expressed in terms of $G(1)$. As we make the substitutions, we generate factors

$$(2^{n/2} + 1)(2^{n/4} + 1)(2^{n/8} + 1) \cdots (2^1 + 1)$$

If we neglect the "+1" term in each factor, we have approximately the product $2^{n/2}2^{n/4}2^{n/8} \cdots 2^1$, which is

$$2^{n/2+n/4+n/8+\cdots+1}$$

or 2^{n-1} if we sum the geometric series in the exponent. That is half of 2^n, and so we might guess that 2^n is a term in the solution $G(n)$. However, if we guess that $f(n) = c2^n$ is an upper bound on $G(n)$, we shall fail, as the reader may check. That is, we get two inequalities involving c that have no solution.

We shall thus guess the next simplest form, $f(n) = c2^n + d$, and here we shall be successful. That is, we can prove the following statement by complete induction on n for some constants c and d:

STATEMENT $S(n)$: If n is a power of 2 and $n \geq 1$, then $G(n) \leq c2^n + d$.

BASIS. If $n = 1$, then we must show that $G(1) \leq c2^1 + d$, that is, that $3 \leq 2c + d$. This inequality becomes one of the constraints on c and d.

INDUCTION. As in Example 3.27, the only hard part occurs when n is a power of 2 and we want to prove $S(n)$ from $S(n/2)$. The equation in this case is

$$G(n/2) \leq c2^{n/2} + d$$

We must prove $S(n)$, which is $G(n) \leq c2^n + d$. We start with the inductive definition of G,

$$G(n) = (2^{n/2} + 1)G(n/2)$$

and then substitute our upper bound for $G(n/2)$, converting this expression to

$$G(n) \leq (2^{n/2} + 1)(c2^{n/2} + d)$$

Simplifying, we get

$$G(n) \leq c2^n + (c + d)2^{n/2} + d$$

That will give the desired upper bound, $c2^n + d$, on $G(n)$, provided that the excess on the right, $(c + d)2^{n/2}$, is no more than 0. It is thus sufficient that $c + d \leq 0$.

We need to select c and d to satisfy the two inequalities

1. $2c + d \geq 3$, from the basis, and
2. $c + d \leq 0$, from the induction.

For example, these inequalities are satisfied if $c = 3$ and $d = -3$. Then we know that $G(n) \leq 3(2^n - 1)$. Thus, $G(n)$ grows exponentially with n. It happens that this function is the exact solution, that is, that $G(n) = 3(2^n - 1)$, as the reader may prove by induction on n. ✦

EXERCISES

3.11.1: Let $T(n)$ be defined by the recurrence

$$T(n) = T(n - 1) + g(n), \text{ for } n > 1$$

Prove by induction on i that if $1 \leq i < n$, then

$$T(n) = T(n - i) + \sum_{j=0}^{i-1} g(n - j)$$

3.11.2: Suppose we have a recurrence of the form

$$T(1) = a$$
$$T(n) = T(n - 1) + g(n), \text{ for } n > 1$$

Give tight big-oh upper bounds on the solution if $g(n)$ is

a) n^2
b) $n^2 + 3n$
c) $n^{3/2}$
d) $n \log n$
e) 2^n

3.11.3: Suppose we have a recurrence of the form

$$T(1) = a$$
$$T(n) = T(n/2) + g(n), \text{ for } n \text{ a power of 2 and } n > 1$$

Give tight big-oh upper bounds on the solution if $g(n)$ is

a) n^2
b) $2n$
c) 10
d) $n \log n$
e) 2^n

3.11.4*: Guess each of the following as the solution to the recurrence

$$T(1) = a$$
$$T(n) = 2T(n/2) + bn, \text{ for } n \text{ a power of 2 and } n > 1$$

a) $cn \log_2 n + dn + e$
b) $cn + d$
c) cn^2

What constraints on the unknown constants c, d, and e are implied? For which of these forms does there exist an upper bound on $T(n)$?

3.11.5: Show that if we guess $G(n) \leq c2^n$ for the recurrence of Example 3.28, we fail to find a solution.

3.11.6*: Show that if

$$T(1) = a$$
$$T(n) = T(n-1) + n^k, \text{ for } n > 1$$

then $T(n)$ is $O(n^{k+1})$. You may assume $k \geq 0$. Also, show that this is the tightest simple big-oh upper bound, that is, that $T(n)$ is not $O(n^m)$ if $m < k+1$. *Hint*: Expand $T(n)$ in terms of $T(n-i)$, for $i = 1, 2, \ldots$, to get the upper bound. For the lower bound, show that $T(n)$ is at least cn^{k+1} for some particular $c > 0$.

3.11.7**: Show that if

$$T(1) = a$$
$$T(n) = cT(n-1) + p(n), \text{ for } n > 1$$

where $p(n)$ is any polynomial in n and $c > 1$, then $T(n)$ is $O(c^n)$. Also, show that this is the tightest simple big-oh upper bound, that is, that $T(n)$ is not $O(d^n)$ if $d < c$.

3.11.8**: Consider the recurrence

$$T(1) = a$$
$$T(n) = cT(n/d) + bn^k, \text{ for } n \text{ a power of } d$$

Iteratively expand $T(n)$ in terms of $T(n/d^i)$ for $i = 1, 2, \ldots$. Show that

a) If $c > d^k$, then $T(n)$ is $O(n^{\log_d c})$
b) If $c = d^k$, then $T(n)$ is $O(n^k \log n)$
c) If $c < d^k$, then $T(n)$ is $O(n^k)$

3.11.9: Solve the following recurrences, each of which has $T(1) = a$:

a) $T(n) = 3T(n/2) + n^2$, for n a power of 2 and $n > 1$
b) $T(n) = 10T(n/3) + n^2$, for n a power of 3 and $n > 1$
c) $T(n) = 16T(n/4) + n^2$, for n a power of 4 and $n > 1$

You may use the solutions of Exercise 3.11.8.

3.11.10: Solve the recurrence

$$T(1) = 1$$
$$T(n) = 3^n T(n/2), \text{ for } n \text{ a power of 2 and } n > 1$$

3.11.11: The *Fibonacci recurrence* is $F(0) = F(1) = 1$, and

$F(n) = F(n-1) + F(n-2)$, for $n > 1$

The values $F(0), F(1), F(2), \ldots$ form the sequence of Fibonacci numbers, in which each number after the first two is the sum of the two previous numbers. (See Exercise 3.9.4.) Let $r = (1 + \sqrt{5})/2$. This constant r is called the *golden ratio* and its value is about 1.62. Show that $F(n)$ is $O(r^n)$. *Hint*: For the induction, it helps to guess that $F(n) \leq ar^n$ for some n, and attempt to prove that inequality by induction on n. The basis must incorporate the two values $n = 0$ and $n = 1$. In the inductive step, it helps to notice that r satisfies the equation $r^2 = r + 1$.

Golden ratio

3.12 Summary of Chapter 3

Here are the important concepts covered in Chapter 3.

- Many factors go into the selection of an algorithm for a program, but simplicity, ease of implementation, and efficiency often dominate.
- Big-oh expressions provide a convenient notation for upper bounds on the running times of programs.
- There are recursive rules for evaluating the running time of the various compound statements of C, such as for-loops and conditions, in terms of the running times of their constituent parts.
- We can evaluate the running time of a function by drawing a structure tree that represents the nesting structure of statements and evaluating the running time of the various pieces in a bottom-up order.
- Recurrence relations are a natural way to model the running time of recursive programs.
- We can solve recurrence relations either by iterated substitution or by guessing a solution and checking our guess is correct.

Divide and conquer is an important algorithm-design technique in which a problem is partitioned into subproblems, the solutions to which can be combined to provide a solution to the whole problem. A few rules of thumb can be used to evaluate the running time of the resulting algorithm: A function that takes time $O(1)$ and then calls itself on a subproblem of size $n-1$ takes time $O(n)$. Examples are the factorial function and the merge function.

- More generally, a function that takes time $O(n^k)$ and then calls itself on a subproblem of size $n - 1$ takes time $O(n^{k+1})$.
- If a function calls itself twice but the recursion goes on for $\log_2 n$ levels (as in merge sort), then the total running time is $O(n \log n)$ times the work done at each call, plus $O(n)$ times the work done at the basis. In the case of merge sort, the work at each call, including basis calls, is $O(1)$, so the total running time is $O(n \log n) + O(n)$, or just $O(n \log n)$.

- If a function calls itself twice and the recursion goes on for n levels (as in the Fibonacci program of Exercise 3.9.4), then the running time is exponential in n.

3.13 Bibliographic Notes for Chapter 3

The study of the running time of programs and the computational complexity of problems was pioneered by Hartmanis and Stearns [1964]. Knuth [1968] was the book that established the study of the running time of algorithms as an essential ingredient in computer science.

Since that time, a rich theory for the difficulty of problems has been developed. Many of the key ideas are found in Aho, Hopcroft, and Ullman [1974, 1983].

In this chapter, we have concentrated on upper bounds for the running times of programs. Knuth [1976] describes analogous notations for lower bounds and exact bounds on running times.

For more discussion of divide and conquer as an algorithm-design technique, see Aho, Hopcroft, and Ullman [1974] or Borodin and Munro [1975]. Additional information on techniques for solving recurrence relations can be found in Graham, Knuth, and Patashnik [1989].

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1974]. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1983]. *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass.

Borodin, A. B., and I. Munro [1975]. *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York.

Graham, R. L., D. E. Knuth, and O. Patashnik [1989]. *Concrete Mathematics: a Foundation for Computer Science*, Addison-Wesley, Reading, Mass.

Knuth, D. E. [1968]. *The Art of Computer Programming* Vol. I: *Fundamental Algorithms*, Addison-Wesley, Reading, Mass.

Knuth, D. E. [1976]. "Big omicron, big omega, and big theta," *ACM SIGACT News* **8**:2, pp. 18–23.