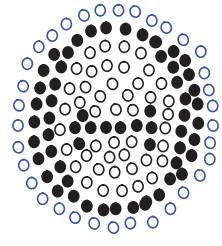
Chapter 4 Analysis Tools



Contents

4.1 The Seven Functions Used in This Book

In this section, we briefly discuss the seven most important functions used in the analysis of algorithms. We use only these seven simple functions for almost all the analysis we do in this book. In fact, sections that use a function other than one of these seven are marked with a star ($\star$) to indicate that they are optional. In addition to these seven fundamental functions, Appendix A contains a list of other useful mathematical facts that apply in the context of data structure and algorithm analysis.

4.1.1 The Constant Function

The simplest function we can think of is the ***constant function***. This is the function,

$$f(n) = c,$$

for some fixed constant c, such as $c = 5$, $c = 27$, or $c = 2^{10}$. That is, for any argument n, the constant function $f(n)$ assigns the value c. In other words, it doesn't matter what the value of n is, $f(n)$ is always be equal to the constant value c.

Since we are most interested in integer functions, the most fundamental constant function is $g(n) = 1$, and this is the typical constant function we use in this book. Note that any other constant function, $f(n) = c$, can be written as a constant c times $g(n)$. That is, $f(n) = cg(n)$ in this case.

As simple as it is, the constant function is useful in algorithm analysis because it characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to some variable, or comparing two numbers.

4.1.2 The Logarithm Function

One of the interesting and sometimes even surprising aspects of the analysis of data structures and algorithms is the ubiquitous presence of the ***logarithm function***, $f(n) = \log_b n$, for some constant $b > 1$. This function is defined as follows:

$$x = \log_b n \quad \text{if and only if} \quad b^x = n.$$

By definition, $\log_b 1 = 0$. The value b is known as the ***base*** of the logarithm.

Computing the logarithm function exactly for any integer n involves the use of calculus, but we can use an approximation that is good enough for our purposes without calculus. In particular, we can easily compute the smallest integer greater than or equal to $\log_a n$, since this number is equal to the number of times

we can divide n by a until we get a number less than or equal to 1. For example, this evaluation of $\log_3 27$ is 3, since $27/3/3/3 = 1$. Likewise, this evaluation of $\log_4 64$ is 3, since $64/4/4/4 = 1$, and this approximation to $\log_2 12$ is 4, since $12/2/2/2/2 = 0.75 \leq 1$. This base-2 approximation arises in algorithm analysis, since a common operation in many algorithms is to repeatedly divide an input in half.

Indeed, since computers store integers in binary, the most common base for the logarithm function in computer science is 2. In fact, this base is so common that we typically leave it off when it is 2. That is, for us,

$$\log n = \log_2 n.$$

We note that most handheld calculators have a button marked LOG, but this is typically for calculating the logarithm base 10, not base 2.

There are some important rules for logarithms, similar to the exponent rules.

Proposition 4.1 (Logarithm Rules): *Given real numbers $a > 0$, $b > 1$, $c > 0$ and $d > 1$, we have:*

1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \log_b a$
4. $\log_b a = (\log_d a)/\log_d b$
5. $b^{\log_d a} = a^{\log_d b}$

Also, as a notational shorthand, we use $\log^c n$ to denote the function $(\log n)^c$. Rather than show how we could derive each of the identities above which all follow from the definition of logarithms and exponents, let us illustrate these identities with a few examples instead.

Example 4.2: *We demonstrate below some interesting applications of the logarithm rules from Proposition 4.1 (using the usual convention that the base of a logarithm is 2 if it is omitted).*

- $\log(2n) = \log 2 + \log n = 1 + \log n$, *by rule 1*
- $\log(n/2) = \log n - \log 2 = \log n - 1$, *by rule 2*
- $\log n^3 = 3 \log n$, *by rule 3*
- $\log 2^n = n \log 2 = n \cdot 1 = n$, *by rule 3*
- $\log_4 n = (\log n)/\log 4 = (\log n)/2$, *by rule 4*
- $2^{\log n} = n^{\log 2} = n^1 = n$, *by rule 5*

As a practical matter, we note that rule 4 gives us a way to compute the base-2 logarithm on a calculator that has a base-10 logarithm button, LOG, *for*

$$\log_2 n = \mathsf{LOG}\, n / \mathsf{LOG}\, 2.$$

4.1.3 The Linear Function

Another simple yet important function is the ***linear function***,

$$f(n) = n.$$

That is, given an input value n, the linear function f assigns the value n itself.

This function arises in algorithm analysis any time we have to do a single basic operation for each of n elements. For example, comparing a number x to each element of an array of size n requires n comparisons. The linear function also represents the best running time we can hope to achieve for any algorithm that processes a collection of n objects that are not already in the computer's memory, since reading in the n objects itself requires n operations.

4.1.4 The N-Log-N Function

The next function we discuss in this section is the ***n-log-n function***,

$$f(n) = n \log n.$$

That is, the function that assigns to an input n the value of n times the logarithm base 2 of n. This function grows a little faster than the linear function and a lot slower than the quadratic function. Thus, as we show on several occasions, if we can improve the running time of solving some problem from quadratic to n-log-n, we have an algorithm that runs much faster in general.

4.1.5 The Quadratic Function

Another function that appears quite often in algorithm analysis is the ***quadratic function***,

$$f(n) = n^2.$$

That is, given an input value n, the function f assigns the product of n with itself (in other words, "n squared").

The main reason why the quadratic function appears in the analysis of algorithms is that there are many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. Thus, in such cases, the algorithm performs $n \cdot n = n^2$ operations.

Nested Loops and the Quadratic Function

The quadratic function can also arise in the context of nested loops where the first iteration of a loop uses one operation, the second uses two operations, the third uses three operations, and so on. That is, the number of operations is

$$1+2+3+\cdots+(n-2)+(n-1)+n.$$

In other words, this is the total number of operations that are performed by the nested loop if the number of operations performed inside the loop increases by one with each iteration of the outer loop. This quantity also has an interesting history.

In 1787, a German schoolteacher decided to keep his 9- and 10-year-old pupils occupied by adding up the integers from 1 to 100. But almost immediately one of the children claimed to have the answer! The teacher was suspicious, for the student had only the answer on his slate. But the answer was correct—5,050—and the student, Carl Gauss, grew up to be one of the greatest mathematicians of his time. It is widely suspected that young Gauss used the following identity.

Proposition 4.3: *For any integer $n \geq 1$, we have:*

$$1+2+3+\cdots+(n-2)+(n-1)+n = \frac{n(n+1)}{2}.$$

We give two "visual" justifications of Proposition 4.3 in Figure 4.1.

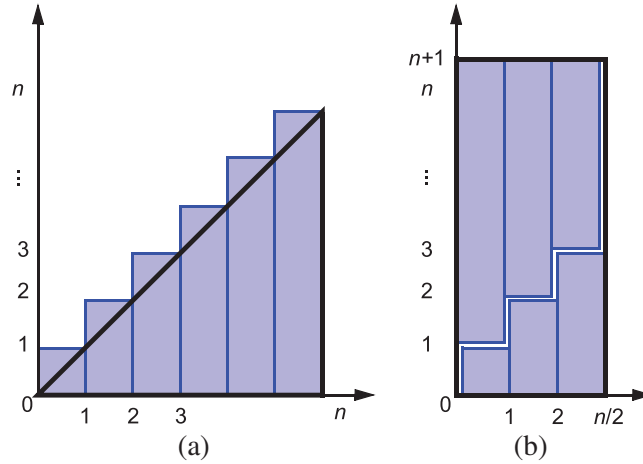


Figure 4.1: Visual justifications of Proposition 4.3. Both illustrations visualize the identity in terms of the total area covered by n unit-width rectangles with heights $1,2,\ldots,n$. In (a), the rectangles are shown to cover a big triangle of area $n^2/2$ (base n and height n) plus n small triangles of area $1/2$ each (base 1 and height 1). In (b), which applies only when n is even, the rectangles are shown to cover a big rectangle of base $n/2$ and height $n+1$.

The lesson to be learned from Proposition 4.3 is that if we perform an algorithm with nested loops such that the operations in the inner loop increase by one each time, then the total number of operations is quadratic in the number of times, n, we perform the outer loop. In particular, the number of operations is $n^2/2 + n/2$, in this case, which is a little more than a constant factor $(1/2)$ times the quadratic function n^2. In other words, such an algorithm is only slightly better than an algorithm that uses n operations each time the inner loop is performed. This observation might at first seem nonintuitive, but it is nevertheless true as shown in Figure 4.1.

4.1.6 The Cubic Function and Other Polynomials

Continuing our discussion of functions that are powers of the input, we consider the ***cubic function***,

$$f(n) = n^3,$$

which assigns to an input value n the product of n with itself three times. This function appears less frequently in the context of algorithm analysis than the constant, linear, and quadratic functions previously mentioned, but it does appear from time to time.

Polynomials

Interestingly, the functions we have listed so far can be viewed as all being part of a larger class of functions, the ***polynomials***.

A ***polynomial*** function is a function of the form,

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d,$$

where $a_0, a_1, \ldots, a_d$ are constants, called the ***coefficients*** of the polynomial, and $a_d \neq 0$. Integer d, which indicates the highest power in the polynomial, is called the ***degree*** of the polynomial.

For example, the following functions are all polynomials:

- $f(n) = 2 + 5n + n^2$
- $f(n) = 1 + n^3$
- $f(n) = 1$
- $f(n) = n$
- $f(n) = n^2$

Therefore, we could argue that this book presents just four important functions used in algorithm analysis, but we stick to saying that there are seven, since the constant, linear, and quadratic functions are too important to be lumped in with other polynomials. Running times that are polynomials with degree, d, are generally better than polynomial running times of larger degree.

Summations

A notation that appears again and again in the analysis of data structures and algorithms is the ***summation***, which is defined as

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b),$$

where a and b are integers and $a \leq b$. Summations arise in data structure and algorithm analysis because the running times of loops naturally give rise to summations.

Using a summation, we can rewrite the formula of Proposition 4.3 as

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

Likewise, we can write a polynomial $f(n)$ of degree d with coefficients $a_0, \ldots, a_d$ as

$$f(n) = \sum_{i=0}^{d} a_i n^i.$$

Thus, the summation notation gives us a shorthand way of expressing sums of increasing terms that have a regular structure.

4.1.7 The Exponential Function

Another function used in the analysis of algorithms is the ***exponential function***,

$$f(n) = b^n,$$

where b is a positive constant, called the ***base***, and the argument n is the ***exponent***. That is, function $f(n)$ assigns to the input argument n the value obtained by multiplying the base b by itself n times. In algorithm analysis, the most common base for the exponential function is $b = 2$. For instance, if we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the nth iteration is 2^n. In addition, an integer word containing n bits can represent all the nonnegative integers less than 2^n. Thus, the exponential function with base 2 is quite common. The exponential function is also referred to as ***exponent function***.

We sometimes have other exponents besides n, however; hence, it is useful for us to know a few handy rules for working with exponents. In particular, the following ***exponent rules*** are quite helpful.

Proposition 4.4 (Exponent Rules): *Given positive integers a, b, and c, we have:*

1. $(b^a)^c = b^{ac}$
2. $b^a b^c = b^{a+c}$
3. $b^a / b^c = b^{a-c}$

For example, we have the following:

- $256 = 16^2 = (2^4)^2 = 2^{4\cdot 2} = 2^8 = 256$ (Exponent Rule 1)
- $243 = 3^5 = 3^{2+3} = 3^2 3^3 = 9 \cdot 27 = 243$ (Exponent Rule 2)
- $16 = 1024/64 = 2^{10}/2^6 = 2^{10-6} = 2^4 = 16$ (Exponent Rule 3)

We can extend the exponential function to exponents that are fractions or real numbers and to negative exponents, as follows. Given a positive integer k, we define $b^{1/k}$ to be kth root of b, that is, the number r such that $r^k = b$. For example, $25^{1/2} = 5$, since $5^2 = 25$. Likewise, $27^{1/3} = 3$ and $16^{1/4} = 2$. This approach allows us to define any power whose exponent can be expressed as a fraction, since $b^{a/c} = (b^a)^{1/c}$, by Exponent Rule 1. For example, $9^{3/2} = (9^3)^{1/2} = 729^{1/2} = 27$. Thus, $b^{a/c}$ is really just the cth root of the integral exponent b^a.

We can further extend the exponential function to define b^x for any real number x, by computing a series of numbers of the form $b^{a/c}$ for fractions a/c that get progressively closer and closer to x. Any real number x can be approximated arbitrarily close by a fraction a/c; hence, we can use the fraction a/c as the exponent of b to get arbitrarily close to b^x. So, for example, the number 2^π is well defined. Finally, given a negative exponent d, we define $b^d = 1/b^{-d}$, which corresponds to applying Exponent Rule 3 with $a = 0$ and $c = -d$.

Geometric Sums

Suppose we have a loop where each iteration takes a multiplicative factor longer than the previous one. This loop can be analyzed using the following proposition.

Proposition 4.5: *For any integer $n \geq 0$ and any real number a such that $a > 0$ and $a \neq 1$, consider the summation*

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

(remembering that $a^0 = 1$ if $a > 0$). This summation is equal to

$$\frac{a^{n+1} - 1}{a - 1}.$$

Summations as shown in Proposition 4.5 are called ***geometric*** summations, because each term is geometrically larger than the previous one if $a > 1$. For example, everyone working in computing should know that

$$1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1,$$

since this is the largest integer that can be represented in binary notation using n bits.

4.1.8 Comparing Growth Rates

To sum up, Table 4.1 shows each of the seven common functions used in algorithm analysis in order.

constant	*logarithm*	*linear*	*n-log-n*	*quadratic*	*cubic*	*exponential*
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Table 4.1: Classes of functions. Here we assume that $a > 1$ is a constant.

Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function, and we would like our algorithms to run in linear or n-log-n time. Algorithms with quadratic or cubic running times are less practical, but algorithms with exponential running times are infeasible for all but the smallest sized inputs. Plots of the seven functions are shown in Figure 4.2.

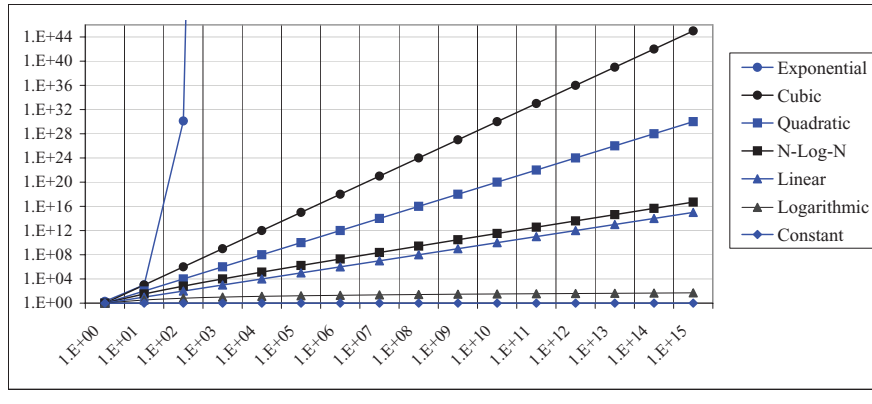


Figure 4.2: Growth rates for the seven fundamental functions used in algorithm analysis. We use base $a = 2$ for the exponential function. The functions are plotted in a log-log chart, to compare the growth rates primarily as slopes. Even so, the exponential function grows too fast to display all its values on the chart. Also, we use the scientific notation for numbers, where aE+b denotes $a10^b$.

The Ceiling and Floor Functions

One additional comment concerning the functions above is in order. The value of a logarithm is typically not an integer, yet the running time of an algorithm is usually expressed by means of an integer quantity, such as the number of operations performed. Thus, the analysis of an algorithm may sometimes involve the use of the ***floor function*** and ***ceiling function***, which are defined respectively as follows:

- $\lfloor x \rfloor$ = the largest integer less than or equal to x
- $\lceil x \rceil$ = the smallest integer greater than or equal to x

4.2 Analysis of Algorithms

In a classic story, the famous mathematician Archimedes was asked to determine if a golden crown commissioned by the king was indeed pure gold, and not part silver, as an informant had claimed. Archimedes discovered a way to perform this analysis while stepping into a (Greek) bath. He noted that water spilled out of the bath in proportion to the amount of him that went in. Realizing the implications of this fact, he immediately got out of the bath and ran naked through the city shouting, "Eureka, eureka!," for he had discovered an analysis tool (displacement), which, when combined with a simple scale, could determine if the king's new crown was good or not. That is, Archimedes could dip the crown and an equal-weight amount of gold into a bowl of water to see if they both displaced the same amount. This discovery was unfortunate for the goldsmith, however, for when Archimedes did his analysis, the crown displaced more water than an equal-weight lump of pure gold, indicating that the crown was not, in fact, pure gold.

In this book, we are interested in the design of "good" data structures and algorithms. Simply put, a ***data structure*** is a systematic way of organizing and accessing data, and an ***algorithm*** is a step-by-step procedure for performing some task in a finite amount of time. These concepts are central to computing, but to be able to classify some data structures and algorithms as "good," we must have precise ways of analyzing them.

The primary analysis tool we use in this book involves characterizing the running times of algorithms and data structure operations, with space usage also being of interest. Running time is a natural measure of "goodness," since time is a precious resource—computer solutions should run as fast as possible.

In general, the running time of an algorithm or data structure method increases with the input size, although it may also vary for different inputs of the same size. Also, the running time is affected by the hardware environment (as reflected in the processor, clock rate, memory, disk, etc.) and software environment (as reflected in the operating system, programming language, compiler, interpreter, etc.) in which the algorithm is implemented, compiled, and executed. All other factors being equal, the running time of the same algorithm on the same input data is smaller if the computer has, say, a much faster processor or if the implementation is done in a program compiled into native machine code instead of an interpreted implementation run on a virtual machine. Nevertheless, in spite of the possible variations that come from different environmental factors, we would like to focus on the relationship between the running time of an algorithm and the size of its input.

We are interested in characterizing an algorithm's running time as a function of the input size. But what is the proper way of measuring it?

4.2.1 Experimental Studies

If an algorithm has been implemented, we can study its running time by executing it on various test inputs and recording the actual time spent in each execution. Fortunately, such measurements can be taken in an accurate manner by using system calls that are built into the language or operating system (for example, by using the clock() function or calling the run-time environment with profiling enabled). Such tests assign a specific running time to a specific input size, but we are interested in determining the general dependence of running time on the size of the input. In order to determine this dependence, we should perform several experiments on many different test inputs of various sizes. Then we can visualize the results of such experiments by plotting the performance of each run of the algorithm as a point with x-coordinate equal to the input size, n, and y-coordinate equal to the running time, t. (See Figure 4.3.) From this visualization and the data that supports it, we can perform a statistical analysis that seeks to fit the best function of the input size to the experimental data. To be meaningful, this analysis requires that we choose good sample inputs and test enough of them to be able to make sound statistical claims about the algorithm's running time.

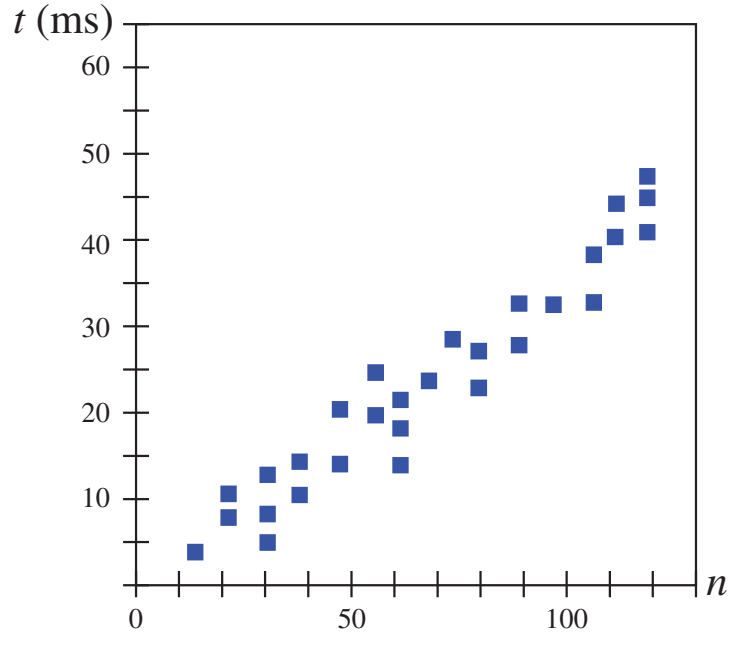


Figure 4.3: Results of an experimental study on the running time of an algorithm. A dot with coordinates (n,t) indicates that on an input of size n, the running time of the algorithm is t milliseconds (ms).

While experimental studies of running times are useful, they have three major limitations:

- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
- We have difficulty comparing the experimental running times of two algorithms unless the experiments were performed in the same hardware and software environments.
- We have to fully implement and execute an algorithm in order to study its running time experimentally.

This last requirement is obvious, but it is probably the most time consuming aspect of performing an experimental analysis of an algorithm. The other limitations impose serious hurdles too, of course. Thus, we would ideally like to have an analysis tool that allows us to avoid performing experiments.

In the rest of this chapter, we develop a general way of analyzing the running times of algorithms that:

- Takes into account all possible inputs.
- Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent from the hardware and software environment.
- Can be performed by studying a high-level description of the algorithm without actually implementing it or running experiments on it.

This methodology aims at associating, with each algorithm, a function $f(n)$ that characterizes the running time of the algorithm as a function of the input size n. Typical functions that are encountered include the seven functions mentioned earlier in this chapter.

4.2.2 Primitive Operations

As noted above, experimental analysis is valuable, but it has its limitations. If we wish to analyze a particular algorithm without performing experiments on its running time, we can perform an analysis directly on the high-level pseudo-code instead. We define a set of ***primitive operations*** such as the following:

- Assigning a value to a variable
- Calling a function
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a function

Counting Primitive Operations

Specifically, a primitive operation corresponds to a low-level instruction with an execution time that is constant. Instead of trying to determine the specific execution time of each primitive operation, we simply ***count*** how many primitive operations are executed, and use this number t as a measure of the running time of the algorithm.

This operation count correlates to an actual running time in a specific computer, since each primitive operation corresponds to a constant-time instruction, and there are only a fixed number of primitive operations. The implicit assumption in this approach is that the running times of different primitive operations is fairly similar. Thus, the number, t, of primitive operations an algorithm performs is proportional to the actual running time of that algorithm.

An algorithm may run faster on some inputs than it does on others of the same size. Thus, we may wish to express the running time of an algorithm as the function of the input size obtained by taking the average over all possible inputs of the same size. Unfortunately, such an ***average-case*** analysis is typically quite challenging. It requires us to define a probability distribution on the set of inputs, which is often a difficult task. Figure 4.4 schematically shows how, depending on the input distribution, the running time of an algorithm can be anywhere between the worst-case time and the best-case time. For example, what if inputs are really only of types "A" or "D"?

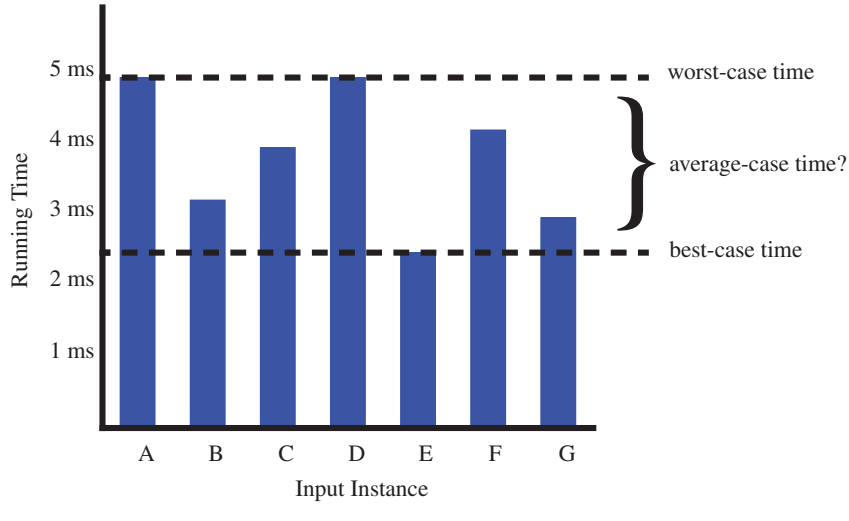


Figure 4.4: The difference between best-case and worst-case time. Each bar represents the running time of some algorithm on a different possible input.

Focusing on the Worst Case

An average-case analysis usually requires that we calculate expected running times based on a given input distribution, which usually involves sophisticated probability theory. Therefore, for the remainder of this book, unless we specify otherwise, we characterize running times in terms of the ***worst case***, as a function of the input size, n, of the algorithm.

Worst-case analysis is much easier than average-case analysis, as it requires only the ability to identify the worst-case input, which is often simple. Also, this approach typically leads to better algorithms. Making the standard of success for an algorithm to perform well in the worst case necessarily requires that it does well on ***every*** input. That is, designing for the worst case leads to stronger algorithmic "muscles," much like a track star who always practices by running up an incline.

4.2.3 Asymptotic Notation

In general, each basic step in a pseudo-code description or a high-level language implementation corresponds to a small number of primitive operations (except for function calls, of course). Thus, we can perform a simple analysis of an algorithm written in pseudo-code that estimates the number of primitive operations executed up to a constant factor, by pseudo-code steps (but we must be careful, since a single line of pseudo-code may denote a number of steps in some cases).

In algorithm analysis, we focus on the growth rate of the running time as a function of the input size n, taking a "big-picture" approach. It is often enough just to know that the running time of an algorithm such as arrayMax, shown in Code Fragment 4.1, ***grows proportionally to*** n, with its true running time being n times a constant factor that depends on the specific computer.

We analyze algorithms using a mathematical notation for functions that disregards constant factors. Namely, we characterize the running times of algorithms by using functions that map the size of the input, n, to values that correspond to the main factor that determines the growth rate in terms of n. This approach allows us to focus on the "big-picture" aspects of an algorithm's running time.

Algorithm arrayMax(A,n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

```
currMax ← A[0]
for i ← 1 to n − 1 do
    if currMax < A[i] then
        currMax ← A[i]
return currMax
```

Code Fragment 4.1: Algorithm arrayMax.

The "Big-Oh" Notation

Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq cg(n), \quad \text{for} \quad n \geq n_0.$$

This definition is often referred to as the "big-Oh" notation, for it is sometimes pronounced as "$f(n)$ is ***big-Oh*** of $g(n)$." Alternatively, we can also say "$f(n)$ is ***order of*** $g(n)$." (This definition is illustrated in Figure 4.5.)

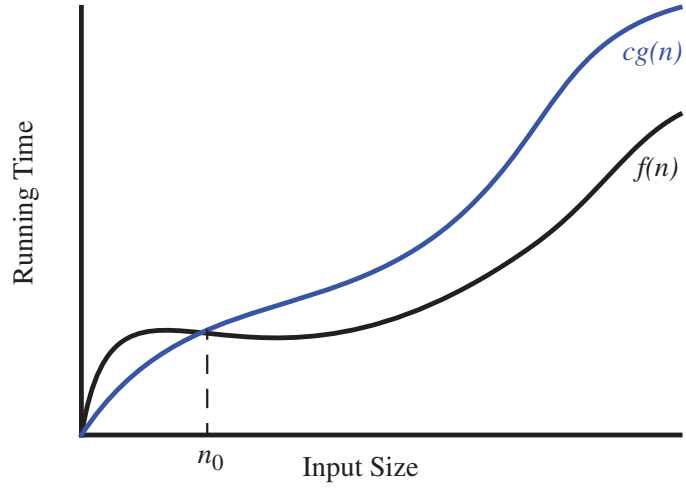


Figure 4.5: The "big-Oh" notation. The function $f(n)$ is $O(g(n))$, since $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

Example 4.6: *The function* $8n - 2$ *is* $O(n)$.

Justification: By the big-Oh definition, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $8n - 2 \leq cn$ for every integer $n \geq n_0$. It is easy to see that a possible choice is $c = 8$ and $n_0 = 1$. Indeed, this is one of infinitely many choices available because any real number greater than or equal to 8 works for c, and any integer greater than or equal to 1 works for n_0. ■

The big-Oh notation allows us to say that a function $f(n)$ is "less than or equal to" another function $g(n)$ up to a constant factor and in the ***asymptotic*** sense as n grows toward infinity. This ability comes from the fact that the definition uses "$\leq$" to compare $f(n)$ to a $g(n)$ times a constant, c, for the asymptotic cases when $n \geq n_0$.

Characterizing Running Times using the Big-Oh Notation

The big-Oh notation is used widely to characterize running times and space bounds in terms of some parameter n, which varies from problem to problem, but is always defined as a chosen measure of the "size" of the problem. For example, if we are interested in finding the largest element in an array of integers, as in the arrayMax algorithm, we should let n denote the number of elements of the array. Using the big-Oh notation, we can write the following mathematically precise statement on the running time of algorithm arrayMax for ***any*** computer.

Proposition 4.7: *The Algorithm* arrayMax*, for computing the maximum element in an array of n integers, runs in $O(n)$ time.*

Justification: The number of primitive operations executed by algorithm arrayMax in each iteration is a constant. Hence, since each primitive operation runs in constant time, we can say that the running time of algorithm arrayMax on an input of size n is at most a constant times n, that is, we may conclude that the running time of algorithm arrayMax is $O(n)$. ■

Some Properties of the Big-Oh Notation

The big-Oh notation allows us to ignore constant factors and lower order terms and focus on the main components of a function that affect its growth.

Example 4.8: $5n^4 + 3n^3 + 2n^2 + 4n + 1$ *is* $O(n^4)$.

Justification: Note that $5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5+3+2+4+1)n^4 = cn^4$, for $c = 15$, when $n \geq n_0 = 1$. ■

In fact, we can characterize the growth rate of any polynomial function.

Proposition 4.9: *If $f(n)$ is a polynomial of degree d, that is,*

$$f(n) = a_0 + a_1 n + \cdots + a_d n^d,$$

and $a_d > 0$, then $f(n)$ is $O(n^d)$.

Justification: Note that, for $n \geq 1$, we have $1 \leq n \leq n^2 \leq \cdots \leq n^d$; hence,

$$a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d \leq (a_0 + a_1 + a_2 + \cdots + a_d) n^d.$$

Therefore, we can show $f(n)$ is $O(n^d)$ by defining $c = a_0 + a_1 + \cdots + a_d$ and $n_0 = 1$. ■

Thus, the highest-degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial. We consider some additional properties of the big-Oh notation in the exercises. Let us consider some further examples here, however, focusing on combinations of the seven fundamental functions used in algorithm design.

Example 4.10: $5n^2 + 3n\log n + 2n + 5$ *is* $O(n^2)$.

Justification: $5n^2 + 3n\log n + 2n + 5 \leq (5+3+2+5)n^2 = cn^2$, for $c = 15$, when $n \geq n_0 = 2$ (note that $n\log n$ is zero for $n = 1$). ■

Example 4.11: $20n^3 + 10n\log n + 5$ *is* $O(n^3)$.

Justification: $20n^3 + 10n\log n + 5 \leq 35n^3$, for $n \geq 1$. ■

Example 4.12: $3\log n + 2$ *is* $O(\log n)$.

Justification: $3\log n + 2 \leq 5\log n$, for $n \geq 2$. Note that $\log n$ is zero for $n = 1$. That is why we use $n \geq n_0 = 2$ in this case. ■

Example 4.13: 2^{n+2} *is* $O(2^n)$.

Justification: $2^{n+2} = 2^n 2^2 = 4 \cdot 2^n$; hence, we can take $c = 4$ and $n_0 = 1$ in this case. ■

Example 4.14: $2n + 100\log n$ *is* $O(n)$.

Justification: $2n + 100\log n \leq 102n$, for $n \geq n_0 = 2$; hence, we can take $c = 102$ in this case. ■

Characterizing Functions in Simplest Terms

In general, we should use the big-Oh notation to characterize a function as closely as possible. While it is true that the function $f(n) = 4n^3 + 3n^2$ is $O(n^5)$ or even $O(n^4)$, it is more accurate to say that $f(n)$ is $O(n^3)$. Consider, by way of analogy, a scenario where a hungry traveler driving along a long country road happens upon a local farmer walking home from a market. If the traveler asks the farmer how much longer he must drive before he can find some food, it may be truthful for the farmer to say, "certainly no longer than 12 hours," but it is much more accurate (and helpful) for him to say, "you can find a market just a few minutes drive up this road." Thus, even with the big-Oh notation, we should strive as much as possible to tell the whole truth.

It is also considered poor taste to include constant factors and lower order terms in the big-Oh notation. For example, it is not fashionable to say that the function $2n^2$ is $O(4n^2 + 6n\log n)$, although this is completely correct. We should strive instead to describe the function in the big-Oh in ***simplest terms***.

The seven functions listed in Section 4.1 are the most common functions used in conjunction with the big-Oh notation to characterize the running times and space usage of algorithms. Indeed, we typically use the names of these functions to refer to the running times of the algorithms they characterize. So, for example, we would say that an algorithm that runs in worst-case time $4n^2 + n\log n$ is a ***quadratic-time*** algorithm, since it runs in $O(n^2)$ time. Likewise, an algorithm running in time at most $5n + 20\log n + 4$ would be called a ***linear-time*** algorithm.

Big-Omega

Just as the big-Oh notation provides an asymptotic way of saying that a function is "less than or equal to" another function, the following notations provide an asymptotic way of saying that a function grows at a rate that is "greater than or equal to" that of another.

Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that $f(n)$ is $\Omega(g(n))$ (pronounced "$f(n)$ is big-Omega of $g(n)$") if $g(n)$ is $O(f(n))$, that is, there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \geq cg(n), \quad \text{for} \quad n \geq n_0.$$

This definition allows us to say asymptotically that one function is greater than or equal to another, up to a constant factor.

Example 4.15: $3n\log n + 2n$ *is* $\Omega(n\log n)$.

Justification: $3n\log n + 2n \geq 3n\log n$, for $n \geq 2$. ■

Big-Theta

In addition, there is a notation that allows us to say that two functions grow at the same rate, up to constant factors. We say that $f(n)$ is $\Theta(g(n))$ (pronounced "$f(n)$ is big-Theta of $g(n)$") if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$, that is, there are real constants $c' > 0$ and $c'' > 0$, and an integer constant $n_0 \geq 1$ such that

$$c'g(n) \leq f(n) \leq c''g(n), \quad \text{for} \quad n \geq n_0.$$

Example 4.16: $3n\log n + 4n + 5\log n$ *is* $\Theta(n\log n)$.

Justification: $3n\log n \leq 3n\log n + 4n + 5\log n \leq (3+4+5)n\log n$ for $n \geq 2$. ■

4.2.4 Asymptotic Analysis

Suppose two algorithms solving the same problem are available: an algorithm A, which has a running time of $O(n)$, and an algorithm B, which has a running time of $O(n^2)$. Which algorithm is better? We know that n is $O(n^2)$, which implies that algorithm A is ***asymptotically better*** than algorithm B, although for a small value of n, B may have a lower running time than A.

We can use the big-Oh notation to order classes of functions by asymptotic growth rate. Our seven functions are ordered by increasing growth rate in the sequence below, that is, if a function $f(n)$ precedes a function $g(n)$ in the sequence, then $f(n)$ is $O(g(n))$:

$$1 \quad \log n \quad n \quad n\log n \quad n^2 \quad n^3 \quad 2^n.$$

We illustrate the growth rates of some important functions in Table 4.2.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Table 4.2: Selected values of fundamental functions in algorithm analysis.

We further illustrate the importance of the asymptotic viewpoint in Table 4.3. This table explores the maximum size allowed for an input instance that is processed by an algorithm in 1 second, 1 minute, and 1 hour. It shows the importance of good algorithm design, because an asymptotically slow algorithm is beaten in the long run by an asymptotically faster algorithm, even if the constant factor for the asymptotically faster algorithm is worse.

Running Time (μs)	*Maximum Problem Size (n)*		
	1 second	*1 minute*	*1 hour*
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

Table 4.3: Maximum size of a problem that can be solved in 1 second, 1 minute, and 1 hour, for various running times measured in microseconds.

The importance of good algorithm design goes beyond just what can be solved effectively on a given computer, however. As shown in Table 4.4, even if we achieve a dramatic speed-up in hardware, we still cannot overcome the handicap of an asymptotically slow algorithm. This table shows the new maximum problem size achievable for any fixed amount of time, assuming algorithms with the given running times are now run on a computer 256 times faster than the previous one.

Running Time	*New Maximum Problem Size*
$400n$	$256m$
$2n^2$	$16m$
2^n	$m+8$

Table 4.4: Increase in the maximum size of a problem that can be solved in a fixed amount of time by using a computer that is 256 times faster than the previous one. Each entry is a function of m, the previous maximum problem size.

4.2.5 Using the Big-Oh Notation

Having made the case of using the big-Oh notation for analyzing algorithms, let us briefly discuss a few issues concerning its use. It is considered poor taste, in general, to say "$f(n) \leq O(g(n))$," since the big-Oh already denotes the "less-than-or-equal-to" concept. Likewise, although common, it is not fully correct to say "$f(n) = O(g(n))$" (with the usual understanding of the "=" relation), since there is no way to make sense of the statement "$O(g(n)) = f(n)$." In addition, it is completely wrong to say "$f(n) \geq O(g(n))$" or "$f(n) > O(g(n))$," since the $g(n)$ in the big-Oh expresses an upper bound on $f(n)$. It is best to say,

"$f(n)$ ***is*** $O(g(n))$."

For the more mathematically inclined, it is also correct to say,

"$f(n) \in O(g(n))$,"

for the big-Oh notation is, technically speaking, denoting a whole collection of functions. In this book, we stick to presenting big-Oh statements as "$f(n)$ ***is*** $O(g(n))$." Even with this interpretation, there is considerable freedom in how we can use arithmetic operations with the big-Oh notation, and with this freedom comes a certain amount of responsibility.

Some Words of Caution

A few words of caution about asymptotic notation are in order at this point. First, note that the use of the big-Oh and related notations can be somewhat misleading should the constant factors they "hide" be very large. For example, while it is true that the function $10^{100}n$ is $O(n)$, if this is the running time of an algorithm being compared to one whose running time is $10n \log n$, we prefer the $O(n \log n)$ time algorithm, even though the linear-time algorithm is asymptotically faster. This preference is because the constant factor, 10^{100}, which is called "one googol," is believed by many astronomers to be an upper bound on the number of atoms in the observable universe. So we are unlikely to ever have a real-world problem that has this number as its input size. Thus, even when using the big-Oh notation, we should at least be somewhat mindful of the constant factors and lower order terms we are "hiding."

The observation above raises the issue of what constitutes a "fast" algorithm. Generally speaking, any algorithm running in $O(n \log n)$ time (with a reasonable constant factor) should be considered efficient. Even an $O(n^2)$ time method may be fast enough in some contexts, that is, when n is small. But an algorithm running in $O(2^n)$ time should almost never be considered efficient.

Exponential Running Times

There is a famous story about the inventor of the game of chess. He asked only that his king pay him 1 grain of rice for the first square on the board, 2 grains for the second, 4 grains for the third, 8 for the fourth, and so on. It is an interesting test of programming skills to write a program to compute exactly the number of grains of rice the king would have to pay. In fact, any C++ program written to compute this number in a single integer value causes an integer overflow to occur (although the run-time machine probably won't complain).

If we must draw a line between efficient and inefficient algorithms, therefore, it is natural to make this distinction be that between those algorithms running in polynomial time and those running in exponential time. That is, make the distinction between algorithms with a running time that is $O(n^c)$, for some constant $c > 1$, and those with a running time that is $O(b^n)$, for some constant $b > 1$. Like so many notions we have discussed in this section, this too should be taken with a "grain of salt," for an algorithm running in $O(n^{100})$ time should probably not be considered "efficient." Even so, the distinction between polynomial-time and exponential-time algorithms is considered a robust measure of tractability.

To summarize, the asymptotic notations of big-Oh, big-Omega, and big-Theta provide a convenient language for us to analyze data structures and algorithms. As mentioned earlier, these notations provide convenience because they let us concentrate on the "big picture" rather than low-level details.

Two Examples of Asymptotic Algorithm Analysis

We conclude this section by analyzing two algorithms that solve the same problem but have rather different running times. The problem we are interested in is the one of computing the so-called ***prefix averages*** of a sequence of numbers. Namely, given an array X storing n numbers, we want to compute an array A such that $A[i]$ is the average of elements $X[0],\ldots,X[i]$, for $i = 0,\ldots,n-1$, that is,

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$

Computing prefix averages has many applications in economics and statistics. For example, given the year-by-year returns of a mutual fund, an investor typically wants to see the fund's average annual returns for the last year, the last three years, the last five years, and the last ten years. Likewise, given a stream of daily Web usage logs, a Web site manager may wish to track average usage trends over various time periods.

A Quadratic-Time Algorithm

Our first algorithm for the prefix averages problem, called prefixAverages1, is shown in Code Fragment 4.2. It computes every element of A separately, following the definition.

Algorithm prefixAverages1(X):
Input: An n-element array X of numbers.
Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0],\ldots,X[i]$.
Let A be an array of n numbers.
for $i \leftarrow 0$ **to** $n-1$ **do**
  $a \leftarrow 0$
  for $j \leftarrow 0$ **to** i **do**
    $a \leftarrow a + X[j]$
  $A[i] \leftarrow a/(i+1)$
return array A

Code Fragment 4.2: Algorithm prefixAverages1.

Let us analyze the prefixAverages1 algorithm.

- Initializing and returning array A at the beginning and end can be done with a constant number of primitive operations per element and takes $O(n)$ time.
- There are two nested **for** loops that are controlled by counters i and j, respectively. The body of the outer loop, controlled by counter i, is executed n times for $i = 0,\ldots,n-1$. Thus, statements $a = 0$ and $A[i] = a/(i+1)$ are executed n times each. This implies that these two statements, plus the incrementing and testing of counter i, contribute a number of primitive operations proportional to n, that is, $O(n)$ time.
- The body of the inner loop, which is controlled by counter j, is executed $i+1$ times, depending on the current value of the outer loop counter i. Thus, statement $a = a + X[j]$ in the inner loop is executed $1+2+3+\cdots+n$ times. By recalling Proposition 4.3, we know that $1+2+3+\cdots+n = n(n+1)/2$, which implies that the statement in the inner loop contributes $O(n^2)$ time. A similar argument can be done for the primitive operations associated with the incrementing and testing counter j, which also take $O(n^2)$ time.

The running time of algorithm prefixAverages1 is given by the sum of three terms. The first and the second term are $O(n)$, and the third term is $O(n^2)$. By a simple application of Proposition 4.9, the running time of prefixAverages1 is $O(n^2)$.

A Linear-Time Algorithm

In order to compute prefix averages more efficiently, we can observe that two consecutive averages $A[i-1]$ and $A[i]$ are similar:

$$\begin{aligned} A[i-1] &= (X[0]+X[1]+\cdots+X[i-1])/i \\ A[i] &= (X[0]+X[1]+\cdots+X[i-1]+X[i])/(i+1). \end{aligned}$$

If we denote with S_i the ***prefix sum*** $X[0]+X[1]+\cdots+X[i]$, we can compute the prefix averages as $A[i] = S_i/(i+1)$. It is easy to keep track of the current prefix sum while scanning array X with a loop. We are now ready to present Algorithm prefixAverages2 in Code Fragment 4.3.

Algorithm prefixAverages2(X):
Input: An n-element array X of numbers.
Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0],\ldots,X[i]$.
Let A be an array of n numbers.
$s \leftarrow 0$
for $i \leftarrow 0$ **to** $n-1$ **do**
  $s \leftarrow s+X[i]$
  $A[i] \leftarrow s/(i+1)$
return array A

Code Fragment 4.3: Algorithm prefixAverages2.

The analysis of the running time of algorithm prefixAverages2 follows:

- Initializing and returning array A at the beginning and end can be done with a constant number of primitive operations per element, and takes $O(n)$ time.
- Initializing variable s at the beginning takes $O(1)$ time.
- There is a single **for** loop, which is controlled by counter i. The body of the loop is executed n times, for $i = 0,\ldots,n-1$. Thus, statements $s = s+X[i]$ and $A[i] = s/(i+1)$ are executed n times each. This implies that these two statements plus the incrementing and testing of counter i contribute a number of primitive operations proportional to n, that is, $O(n)$ time.

The running time of algorithm prefixAverages2 is given by the sum of three terms. The first and the third term are $O(n)$, and the second term is $O(1)$. By a simple application of Proposition 4.9, the running time of prefixAverages2 is $O(n)$, which is much better than the quadratic-time algorithm prefixAverages1.

4.2.6 A Recursive Algorithm for Computing Powers

As a more interesting example of algorithm analysis, let us consider the problem of raising a number x to an arbitrary nonnegative integer, n. That is, we wish to compute the ***power function*** $p(x,n)$, defined as $p(x,n) = x^n$. This function has an immediate recursive definition based on linear recursion:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,n-1) & \text{otherwise} \end{cases}$$

This definition leads immediately to a recursive algorithm that uses $O(n)$ function calls to compute $p(x,n)$. We can compute the power function much faster than this, however, by using the following alternative definition, also based on linear recursion, which employs a squaring technique:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

To illustrate how this definition works, consider the following examples:

$$\begin{aligned} 2^4 &= 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16 \\ 2^5 &= 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32 \\ 2^6 &= 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64 \\ 2^7 &= 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128 \end{aligned}$$

This definition suggests the algorithm of Code Fragment 4.4.

Algorithm Power(x,n):
Input: A number x and integer $n \geq 0$
Output: The value x^n

```
if n = 0 then
    return 1
if n is odd then
    y ← Power(x, (n − 1)/2)
    return x · y · y
else
    y ← Power(x, n/2)
    return y · y
```

Code Fragment 4.4: Computing the power function using linear recursion.

To analyze the running time of the algorithm, we observe that each recursive call of function Power(x,n) divides the exponent, n, by two. Thus, there are $O(\log n)$ recursive calls, not $O(n)$. That is, by using linear recursion and the squaring technique, we reduce the running time for the computation of the power function from $O(n)$ to $O(\log n)$, which is a big improvement.

4.2.7 Some More Examples of Algorithm Analysis

Now that we have the big-Oh notation for doing algorithm analysis, let us give some more examples of simple algorithms that can have their running times characterized using this notation. Moreover, in keeping with our earlier promise, we illustrate below how each of the seven functions given earlier in this chapter can be used to characterize the running time of an example algorithm.

A Constant-Time Method

To illustrate a constant-time algorithm, consider the following C++ function, which returns the ***size*** of an STL vector, that is, the current number of cells in the array:

```
int capacity(const vector<int>& arr) {
  return arr.size();
}
```

This is a very simple algorithm, because the size of a vector is stored as a member variable in the vector object, so it takes only a constant-time lookup to return this value. Thus, the capacity function runs in $O(1)$ time; that is, the running time of this function is independent of the value of n, the size of the array.

Revisiting the Method for Finding the Maximum in an Array

For our next example, let us reconsider a simple problem studied earlier, finding the largest value in an array of integers. We assume that the array is stored as an STL vector. This can be done in C++ as follows:

```
int findMax(const vector<int>& arr) {
  int max = arr[0];
  for (int i = 1; i < arr.size(); i++) {
    if (max < arr[i]) max = arr[i];
  }
  return max;
}
```

This function, which amounts to a C++ implementation of the arrayMax algorithm of Section 4.2.3, compares each of the n elements in the input array to a current maximum, and each time it finds an element larger than the current maximum, it updates the current maximum to be this value. Thus, it spends a constant amount of time for each of the n elements in the array; hence, as with the pseudo-code version of the arrayMax algorithm, the running time of this algorithm is $O(n)$.

Further Analysis of the Maximum-Finding Algorithm

A more interesting question, with respect to the above maximum-finding algorithm, is to ask how many times we update the current maximum value. Note that this statement is executed only if we encounter a value of the array that is larger than our current maximum. In the worst case, this condition could be true each time we perform the test. For instance, this situation would occur if the input array is given to us in sorted order. Thus, in the worst-case, the statement $max = arr[i]$ is performed $n-1$ times, hence $O(n)$ times.

But what if the input array is given to us in random order, with all orders equally likely; what would be the expected number of times we updated the maximum value in this case? To answer this question, note that we update the current maximum in the ith iteration only if the ith element in the array is bigger than all the elements that precede it. But if the array is given to us in random order, the probability that the ith element is larger than all elements that precede it is $1/i$; hence, the expected number of times we update the maximum in this case is $H_n = \sum_{i=1}^{n} 1/i$, which is known as the nth ***Harmonic number***. It turns out (see Proposition A.16) that H_n is $O(\log n)$. Therefore, the expected number of times the maximum is updated when the above maximum-finding algorithm is run on a random array is $O(\log n)$.

Three-Way Set Disjointness

Suppose we are given three sets, A, B, and C, with these sets stored in three different integer arrays, a, b, and c, respectively. The ***three-way set disjointness*** problem is to determine if these three sets are disjoint, that is, whether there is no element x such that $x \in A$, $x \in B$, and $x \in C$. A simple C++ function to determine this property is given below:

```
bool areDisjoint(const vector<int>& a, const vector<int>& b,
                 const vector<int>& c) {
  for (int i = 0; i < a.size(); i++)
    for (int j = 0; j < b.size(); j++)
      for (int k = 0; k < c.size(); k++)
        if ((a[i] == b[j]) && (b[j] == c[k])) return false;
  return true;
}
```

This simple algorithm loops through each possible triple of indices i, j, and k to check if the respective elements indexed in a, b, and c are equal. Thus, if each of these arrays is of size n, then the worst-case running time of this function is $O(n^3)$. Moreover, the worst case is achieved when the sets are disjoint, since in this case we go through all n^3 triples of valid indices, i, j, and k. Such a running time would generally not be considered very efficient, but, fortunately, there is a better way to solve this problem, which we explore in Exercise C-4.3.

Recursion Run Amok

The next few example algorithms we study are for solving the ***element uniqueness problem***, in which we are given a range, $i, i+1, \ldots, j$, of indices for an array, A, which we assume is given as an STL vector. We want to determine if the elements of this range, $A[i], A[i+1], \ldots, A[j]$, are all unique, that is, there is no repeated element in this group of array entries. The first algorithm we give for solving the element uniqueness problem is a recursive one. But it uses recursion in a very inefficient manner, as shown in the following C++ implementation.

```
bool isUnique(const vector<int>& arr, int start, int end) {
  if (start >= end) return true;
  if (!isUnique(arr, start, end-1))
    return false;
  if (!isUnique(arr, start+1, end))
    return false;
  return (arr[start] != arr[end]);
}
```

You should first convince yourself that the function is correct. To analyze this recursive algorithm's running time, let us first determine how much time we spend outside of recursive calls in any invocation of this function. Note, in particular, that there are no loops—just comparisons, arithmetic operations, array element references, and function returns. Thus, the nonrecursive part of each function invocation runs in constant time, that is, $O(1)$ time; hence, to determine the worst-case running time of this function we only need to determine the worst-case total number of calls we make to the isUnique function.

Let n denote the number of entries under consideration, that is, let

$$n = end - start + 1.$$

If $n = 1$, then the running time of the isUnique is $O(1)$, since there are no recursive calls for this case. To characterize the running time of the general case, the important observation to make is that in order to solve a problem of size n, the isUnique function makes two recursive calls on problems of size $n-1$. Thus, in the worst case, a call for a range of size n makes two calls on ranges of size $n-1$, which each make two calls on ranges of size $n-2$, which each make two calls on ranges of size $n-3$, and so on. Thus, in the worst case, the total number of function calls is given by the geometric summation

$$1 + 2 + 4 + \cdots + 2^{n-1},$$

which is equal to $2^n - 1$ by Proposition 4.5. Thus, the worst-case running time of function isUnique is $O(2^n)$. This is an incredibly inefficient method for solving the element uniqueness problem. Its inefficiency comes not from the fact that it uses recursion—it comes from the fact that it uses recursion poorly, which is something we address in Exercise C-4.2.

An Iterative Method for Solving the Element Uniqueness Problem

We can do much better than the above exponential-time method by using the following iterative algorithm:

```
bool isUniqueLoop(const vector<int>& arr, int start, int end) {
  if (start >= end) return true;
  for (int i = start; i < end; i++)
    for (int j = i+1; j <= end; j++)
      if (arr[i] == arr[j]) return false;
  return true;
}
```

This function solves the element uniqueness problem by looping through all distinct pairs of indices, i and j, and checking if any of them indexes a pair of elements that are equal to each other. It does this using two nested **for** loops, such that the first iteration of the outer loop causes $n-1$ iterations of the inner loop, the second iteration of the outer loop causes $n-2$ iterations of the inner loop, the third iteration of the outer loop causes $n-3$ iterations of the inner loop, and so on. Thus, the worst-case running time of this function is proportional to

$$1+2+3+\cdots+(n-1),$$

which is $O(n^2)$ as we saw earlier in this chapter (Proposition 4.3).

Using Sorting as a Problem-Solving Tool

An even better algorithm for the element uniqueness problem is based on using sorting as a problem-solving tool. In this case, by sorting an array of elements, we are guaranteed that any duplicate elements will be placed next to each other. Thus, it suffices to sort the array and look for duplicates among consecutive elements. A C++ implementation of this algorithm follows.

```
bool isUniqueSort(const vector<int>& arr, int start, int end) {
  if (start >= end) return true;
  vector<int> buf(arr);                    // duplicate copy of arr
  sort(buf.begin()+start, buf.begin()+end); // sort the subarray
  for (int i = start; i < end; i++)        // check for duplicates
    if (buf[i] == buf[i+1]) return false;
  return true;
}
```

The function sort is provided by the STL. On most systems, it runs in $O(n \log n)$ time. Since the other steps run in $O(n)$ time, the entire algorithm runs in $O(n \log n)$ time. Incidentally, we can solve the element uniqueness problem even faster, at least in terms of its average-case running time, by using the hash table data structure we explore in Section 9.2.

4.3 Simple Justification Techniques

Sometimes, we want to make claims about an algorithm, such as showing that it is correct or that it runs fast. In order to rigorously make such claims, we must use mathematical language, and in order to back up such claims, we must justify or ***prove*** our statements. Fortunately, there are several simple ways to do this.

4.3.1 By Example

Some claims are of the generic form, "There is an element x in a set S that has property P." To justify such a claim, we only need to produce a particular x in S that has property P. Likewise, some hard-to-believe claims are of the generic form, "Every element x in a set S has property P." To justify that such a claim is false, we only need to produce a particular x from S that does not have property P. Such an instance is called a ***counterexample***.

Example 4.17: *Professor Amongus claims that every number of the form $2^i - 1$ is a prime, when i is an integer greater than 1. Professor Amongus is wrong.*

Justification: To prove Professor Amongus is wrong, we find a counter-example. Fortunately, we need not look too far, for $2^4 - 1 = 15 = 3 \cdot 5$. ■

4.3.2 The "Contra" Attack

Another set of justification techniques involves the use of the negative. The two primary such methods are the use of the ***contrapositive*** and the ***contradiction***. The use of the contrapositive method is like looking through a negative mirror. To justify the statement "if p is true, then q is true" we establish that "if q is not true, then p is not true" instead. Logically, these two statements are the same, but the latter, which is called the ***contrapositive*** of the first, may be easier to think about.

Example 4.18: *Let a and b be integers. If ab is even, then a is even or b is even.*

Justification: To justify this claim, consider the contrapositive, "If a is odd and b is odd, then ab is odd." So, suppose $a = 2i + 1$ and $b = 2j + 1$, for some integers i and j. Then $ab = 4ij + 2i + 2j + 1 = 2(2ij + i + j) + 1$; hence, ab is odd. ■

Besides showing a use of the contrapositive justification technique, the previous example also contains an application of ***DeMorgan's Law***. This law helps us deal with negations, for it states that the negation of a statement of the form "p or q" is "not p and not q." Likewise, it states that the negation of a statement of the form "p and q" is "not p or not q."

Contradiction

Another negative justification technique is justification by ***contradiction***, which also often involves using DeMorgan's Law. In applying the justification by contradiction technique, we establish that a statement q is true by first supposing that q is false and then showing that this assumption leads to a contradiction (such as $2 \neq 2$ or $1 > 3$). By reaching such a contradiction, we show that no consistent situation exists with q being false, so q must be true. Of course, in order to reach this conclusion, we must be sure our situation is consistent before we assume q is false.

Example 4.19: *Let a and b be integers. If ab is odd, then a is odd and b is odd.*

Justification: Let ab be odd. We wish to show that a is odd and b is odd. So, with the hope of leading to a contradiction, let us assume the opposite, namely, suppose a is even or b is even. In fact, without loss of generality, we can assume that a is even (since the case for b is symmetric). Then $a = 2i$ for some integer i. Hence, $ab = (2i)b = 2(ib)$, that is, ab is even. But this is a contradiction: ab cannot simultaneously be odd and even. Therefore a is odd and b is odd. ■

4.3.3 Induction and Loop Invariants

Most of the claims we make about a running time or a space bound involve an integer parameter n (usually denoting an intuitive notion of the "size" of the problem). Moreover, most of these claims are equivalent to saying some statement $q(n)$ is true "for all $n \geq 1$." Since this is making a claim about an infinite set of numbers, we cannot justify this exhaustively in a direct fashion.

Induction

We can often justify claims such as those above as true, however, by using the technique of ***induction***. This technique amounts to showing that, for any particular $n \geq 1$, there is a finite sequence of implications that starts with something known to be true and ultimately leads to showing that $q(n)$ is true. Specifically, we begin a justification by induction by showing that $q(n)$ is true for $n = 1$ (and possibly some other values $n = 2, 3, \ldots, k$, for some constant k). Then we justify that the inductive "step" is true for $n > k$, namely, we show "if $q(i)$ is true for $i < n$, then $q(n)$ is true." The combination of these two pieces completes the justification by induction.

Proposition 4.20: *Consider the Fibonacci function $F(n)$, where we define $F(1) = 1$, $F(2) = 2$, and $F(n) = F(n-1) + F(n-2)$ for $n > 2$. (See Section 2.2.3.) We claim that $F(n) < 2^n$.*

Justification: We show our claim is right by induction.
Base cases: ($n \leq 2$). $F(1) = 1 < 2 = 2^1$ and $F(2) = 2 < 4 = 2^2$.
Induction step: ($n > 2$). Suppose our claim is true for $n' < n$. Consider $F(n)$. Since $n > 2$, $F(n) = F(n-1) + F(n-2)$. Moreover, since $n-1 < n$ and $n-2 < n$, we can apply the inductive assumption (sometimes called the "inductive hypothesis") to imply that $F(n) < 2^{n-1} + 2^{n-2}$, since

$$2^{n-1} + 2^{n-2} < 2^{n-1} + 2^{n-1} = 2 \cdot 2^{n-1} = 2^n.$$

■

Let us do another inductive argument, this time for a fact we have seen before.

Proposition 4.21: *(which is the same as Proposition 4.3)*

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

Justification: We justify this equality by induction.
Base case: $n = 1$. Trivial, for $1 = n(n+1)/2$, if $n = 1$.
Induction step: $n \geq 2$. Assume the claim is true for $n' < n$. Consider n.

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i.$$

By the induction hypothesis, then

$$\sum_{i=1}^{n} i = n + \frac{(n-1)n}{2},$$

which we can simplify as

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

■

We may sometimes feel overwhelmed by the task of justifying something true for *all* $n \geq 1$. We should remember, however, the concreteness of the inductive technique. It shows that, for any particular n, there is a finite step-by-step sequence of implications that starts with something true and leads to the truth about n. In short, the inductive argument is a formula for building a sequence of direct justifications.

Loop Invariants

The final justification technique we discuss in this section is the ***loop invariant***. To prove some statement $\mathcal{S}$ about a loop is correct, define $\mathcal{S}$ in terms of a series of smaller statements $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_k$, where:

1. The ***initial*** claim, $\mathcal{S}_0$, is true before the loop begins.
2. If $\mathcal{S}_{i-1}$ is true before iteration i, then $\mathcal{S}_i$ is true after iteration i.
3. The final statement, $\mathcal{S}_k$, implies the statement $\mathcal{S}$ that we wish to be true.

Let us give a simple example of using a loop-invariant argument to justify the correctness of an algorithm. In particular, let us consider using a loop invariant to justify the correctness of arrayFind, shown in Code Fragment 4.5, for finding an element x in an array A.

Algorithm arrayFind(x,A):
  Input: An element x and an n-element array, A.
  Output: The index i such that $x = A[i]$ or -1 if no element of A is equal to x.
```
i ← 0
while i < n do
  if x = A[i] then
    return i
  else
    i ← i + 1
return −1
```

Code Fragment 4.5: Algorithm arrayFind for finding a given element in an array.

To show that arrayFind is correct, we inductively define a series of statements, $\mathcal{S}_i$, that lead to the correctness of our algorithm. Specifically, we claim the following is true at the beginning of iteration i of the **while** loop:

$\mathcal{S}_i$: x is not equal to any of the first i elements of A.

This claim is true at the beginning of the first iteration of the loop, since there are no elements among the first 0 in A (this kind of a trivially true claim is said to hold ***vacuously***). In iteration i, we compare element x to element $A[i]$ and return the index i if these two elements are equal, which is clearly correct and completes the algorithm in this case. If the two elements x and $A[i]$ are not equal, then we have found one more element not equal to x and we increment the index i. Thus, the claim $\mathcal{S}_i$ is true for this new value of i; hence, it is true at the beginning of the next iteration. If the while-loop terminates without ever returning an index in A, then we have $i = n$. That is, $\mathcal{S}_n$ is true—there are no elements of A equal to x. Therefore, the algorithm correctly returns -1 to indicate that x is not in A.

4.4 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-4.1 There is a well-known city (which will go nameless here) whose inhabitants have the reputation of enjoying a meal only if that meal is the best they have ever experienced in their life. Otherwise, they hate it. Assuming meal quality is distributed uniformly across a person's life, what is the expected number of times inhabitants of this city are happy with their meals?

R-4.2 Give a pseudo-code description of the $O(n)$-time algorithm for computing the power function $p(x,n)$. Also, draw the recursion trace of this algorithm for the computation of $p(2,5)$.

R-4.3 Give a C++ description of Algorithm Power for computing the power function $p(x,n)$ (Code Fragment 4.4).

R-4.4 Draw the recursion trace of the Power algorithm (Code Fragment 4.4, which computes the power function $p(x,n)$) for computing $p(2,9)$.

R-4.5 Analyze the running time of Algorithm BinarySum (Code Fragment 3.41) for arbitrary values of the input parameter n.

R-4.6 Graph the functions $8n$, $4n\log n$, $2n^2$, n^3, and 2^n using a logarithmic scale for the x- and y-axes. That is, if the function is $f(n)$ is y, plot this as a point with x-coordinate at $\log n$ and y-coordinate at $\log y$.

R-4.7 The number of operations executed by algorithms A and B is $8n\log n$ and $2n^2$, respectively. Determine n_0 such that A is better than B for $n \geq n_0$.

R-4.8 The number of operations executed by algorithms A and B is $40n^2$ and $2n^3$, respectively. Determine n_0 such that A is better than B for $n \geq n_0$.

R-4.9 Give an example of a function that is plotted the same on a log-log scale as it is on a standard scale.

R-4.10 Explain why the plot of the function n^c is a straight line with slope c on a log-log scale.

R-4.11 What is the sum of all the even numbers from 0 to $2n$, for any positive integer n?

R-4.12 Show that the following two statements are equivalent:

(a) The running time of algorithm A is always $O(f(n))$.

(b) In the worst case, the running time of algorithm A is $O(f(n))$.

R-4.13 Order the following functions by asymptotic growth rate.

$$\begin{array}{lll} 4n\log n + 2n & 2^{10} & 2^{\log n} \\ 3n + 100\log n & 4n & 2^n \\ n^2 + 10n & n^3 & n\log n \end{array}$$

R-4.14 Show that if $d(n)$ is $O(f(n))$, then $ad(n)$ is $O(f(n))$, for any constant $a > 0$.

R-4.15 Show that if $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then the product $d(n)e(n)$ is $O(f(n)g(n))$.

R-4.16 Give a big-Oh characterization, in terms of n, of the running time of the Ex1 function shown in Code Fragment 4.6.

R-4.17 Give a big-Oh characterization, in terms of n, of the running time of the Ex2 function shown in Code Fragment 4.6.

R-4.18 Give a big-Oh characterization, in terms of n, of the running time of the Ex3 function shown in Code Fragment 4.6.

R-4.19 Give a big-Oh characterization, in terms of n, of the running time of the Ex4 function shown in Code Fragment 4.6.

R-4.20 Give a big-Oh characterization, in terms of n, of the running time of the Ex5 function shown in Code Fragment 4.6.

R-4.21 Bill has an algorithm, find2D, to find an element x in an $n \times n$ array A. The algorithm find2D iterates over the rows of A, and calls the algorithm arrayFind, of Code Fragment 4.5, on each row, until x is found or it has searched all rows of A. What is the worst-case running time of find2D in terms of n? What is the worst-case running time of find2D in terms of N, where N is the total size of A? Would it be correct to say that Find2D is a linear-time algorithm? Why or why not?

R-4.22 For each function $f(n)$ and time t in the following table, determine the largest size n of a problem P that can be solved in time t if the algorithm for solving P takes $f(n)$ microseconds (one entry is already completed).

	1 Second	1 Hour	1 Month	1 Century
$\log n$	$\approx 10^{300000}$			
n				
$n\log n$				
n^2				
2^n				

R-4.23 Show that if $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n) + e(n)$ is $O(f(n) + g(n))$.

Algorithm Ex1(A):
  Input: An array A storing $n \geq 1$ integers.
  Output: The sum of the elements in A.
  $s \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $s \leftarrow s + A[i]$
  return s

Algorithm Ex2(A):
  Input: An array A storing $n \geq 1$ integers.
  Output: The sum of the elements at even cells in A.
  $s \leftarrow A[0]$
  for $i \leftarrow 2$ **to** $n-1$ **by** increments of 2 **do**
    $s \leftarrow s + A[i]$
  return s

Algorithm Ex3(A):
  Input: An array A storing $n \geq 1$ integers.
  Output: The sum of the prefix sums in A.
  $s \leftarrow 0$
  for $i \leftarrow 0$ **to** $n-1$ **do**
    $s \leftarrow s + A[0]$
    for $j \leftarrow 1$ **to** i **do**
      $s \leftarrow s + A[j]$
  return s

Algorithm Ex4(A):
  Input: An array A storing $n \geq 1$ integers.
  Output: The sum of the prefix sums in A.
  $s \leftarrow A[0]$
  $t \leftarrow s$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $s \leftarrow s + A[i]$
    $t \leftarrow t + s$
  return t

Algorithm Ex5(A, B):
  Input: Arrays A and B each storing $n \geq 1$ integers.
  Output: The number of elements in B equal to the sum of prefix sums in A.
  $c \leftarrow 0$
  for $i \leftarrow 0$ **to** $n-1$ **do**
    $s \leftarrow 0$
    for $j \leftarrow 0$ **to** $n-1$ **do**
      $s \leftarrow s + A[0]$
      for $k \leftarrow 1$ **to** j **do**
        $s \leftarrow s + A[k]$
    if $B[i] = s$ **then**
      $c \leftarrow c + 1$
  return c

Code Fragment 4.6: Some algorithms.

R-4.24 Show that if $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n) - e(n)$ is ***not necessarily*** $O(f(n) - g(n))$.

R-4.25 Show that if $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $d(n)$ is $O(g(n))$.

R-4.26 Show that $O(\max\{f(n), g(n)\}) = O(f(n) + g(n))$.

R-4.27 Show that $f(n)$ is $O(g(n))$ if and only if $g(n)$ is $\Omega(f(n))$.

R-4.28 Show that if $p(n)$ is a polynomial in n, then $\log p(n)$ is $O(\log n)$.

R-4.29 Show that $(n+1)^5$ is $O(n^5)$.

R-4.30 Show that 2^{n+1} is $O(2^n)$.

R-4.31 Show that n is $O(n \log n)$.

R-4.32 Show that n^2 is $\Omega(n \log n)$.

R-4.33 Show that $n \log n$ is $\Omega(n)$.

R-4.34 Show that $\lceil f(n) \rceil$ is $O(f(n))$, if $f(n)$ is a positive nondecreasing function that is always greater than 1.

R-4.35 Algorithm A executes an $O(\log n)$-time computation for each entry of an n-element array. What is the worst-case running time of Algorithm A?

R-4.36 Given an n-element array X, Algorithm B chooses $\log n$ elements in X at random and executes an $O(n)$-time calculation for each. What is the worst-case running time of Algorithm B?

R-4.37 Given an n-element array X of integers, Algorithm C executes an $O(n)$-time computation for each even number in X, and an $O(\log n)$-time computation for each odd number in X. What are the best-case and worst-case running times of Algorithm C?

R-4.38 Given an n-element array X, Algorithm D calls Algorithm E on each element $X[i]$. Algorithm E runs in $O(i)$ time when it is called on element $X[i]$.What is the worst-case running time of Algorithm D?

R-4.39 Al and Bob are arguing about their algorithms. Al claims his $O(n \log n)$-time method is ***always*** faster than Bob's $O(n^2)$-time method. To settle the issue, they perform a set of experiments. To Al's dismay, they find that if $n < 100$, the $O(n^2)$-time algorithm runs faster, and only when $n \geq 100$ is the $O(n \log n)$-time one better. Explain how this is possible.

Creativity

C-4.1 Describe a recursive algorithm to compute the integer part of the base-2 logarithm of n using only addition and integer division.

C-4.2 Describe an efficient recursive method for solving the element uniqueness problem, which runs in time that is at most $O(n^2)$ in the worst case without using sorting.

C-4.3 Assuming it is possible to sort n numbers in $O(n \log n)$ time, show that it is possible to solve the three-way set disjointness problem in $O(n \log n)$ time.

C-4.4 Describe an efficient algorithm for finding the 10 largest elements in an array of size n. What is the running time of your algorithm?

C-4.5 Suppose you are given an n-element array A containing distinct integers that are listed in increasing order. Given a number k, describe a recursive algorithm to find two integers in A that sum to k, if such a pair exists. What is the running time of your algorithm?

C-4.6 Given an n-element unsorted array A of n integers and an integer k, describe a recursive algorithm for rearranging the elements in A so that all elements less than or equal to k come before any elements larger than k. What is the running time of your algorithm?

C-4.7 Communication security is extremely important in computer networks, and one way many network protocols achieve security is to encrypt messages. Typical *cryptographic* schemes for the secure transmission of messages over such networks are based on the fact that no efficient algorithms are known for factoring large integers. Hence, if we can represent a secret message by a large prime number p, we can transmit, over the network, the number $r = p \cdot q$, where $q > p$ is another large prime number that acts as the *encryption key*. An eavesdropper who obtains the transmitted number r on the network would have to factor r in order to figure out the secret message p.

Using factoring to figure out a message is very difficult without knowing the encryption key q. To understand why, consider the following naive factoring algorithm:

```
for p = 2, ..., r − 1 do
  if p divides r then
    return "The secret message is p!"
```

a. Suppose that the eavesdropper uses the above algorithm and has a computer that can carry out in 1 microsecond (1 millionth of a second) a division between two integers of up to 100 bits each. Give an estimate of the time that it will take in the worst case to decipher the secret message p if the transmitted message r has 100 bits.

b. What is the worst-case time complexity of the above algorithm? Since the input to the algorithm is just one large number r, assume that the input size n is the number of bytes needed to store r, that is, $n = \lfloor (\log_2 r)/8 \rfloor + 1$, and that each division takes time $O(n)$.

C-4.8 Give an example of a positive function $f(n)$ such that $f(n)$ is neither $O(n)$ nor $\Omega(n)$.

C-4.9 Show that $\sum_{i=1}^{n} i^2$ is $O(n^3)$.

C-4.10 Show that $\sum_{i=1}^{n} i/2^i < 2$.
(Hint: Try to bound this sum term by term with a geometric progression.)

C-4.11 Show that $\log_b f(n)$ is $\Theta(\log f(n))$ if $b > 1$ is a constant.

C-4.12 Describe a method for finding both the minimum and maximum of n numbers using fewer than $3n/2$ comparisons.
(Hint: First construct a group of candidate minimums and a group of candidate maximums.)

C-4.13 Bob built a Web site and gave the URL only to his n friends, which he numbered from 1 to n. He told friend number i that he/she can visit the Web site at most i times. Now Bob has a counter, C, keeping track of the total number of visits to the site (but not the identities of who visits). What is the minimum value for C such that Bob should know that one of his friends has visited his/her maximum allowed number of times?

C-4.14 Al says he can prove that all sheep in a flock are the same color:
Base case: One sheep. It is clearly the same color as itself.
Induction step: A flock of n sheep. Take a sheep, a, out. The remaining $n-1$ are all the same color by induction. Now put sheep a back in and take out a different sheep, b. By induction, the $n-1$ sheep (now with a) are all the same color. Therefore, all the sheep in the flock are the same color.
What is wrong with Al's "justification"?

C-4.15 Consider the following "justification" that the Fibonacci function, $F(n)$ (see Proposition 4.20) is $O(n)$:
Base case ($n \leq 2$): $F(1) = 1$ and $F(2) = 2$.
Induction step ($n > 2$): Assume the claim true for $n' < n$. Consider n. $F(n) = F(n-1) + F(n-2)$. By induction, $F(n-1)$ is $O(n-1)$ and $F(n-2)$ is $O(n-2)$. Then, $F(n)$ is $O((n-1) + (n-2))$, by the identity presented in Exercise R-4.23. Therefore, $F(n)$ is $O(n)$.
What is wrong with this "justification"?

C-4.16 Let $p(x)$ be a polynomial of degree n, that is, $p(x) = \sum_{i=0}^{n} a_i x^i$.
(a) Describe a simple $O(n^2)$ time method for computing $p(x)$.
(b) Now consider a rewriting of $p(x)$ as

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + xa_n)\cdots))),$$

which is known as ***Horner's method***. Using the big-Oh notation, characterize the number of arithmetic operations this method executes.

C-4.17 Consider the Fibonacci function, $F(n)$ (see Proposition 4.20). Show by induction that $F(n)$ is $\Omega((3/2)^n)$.

C-4.18 Given a set $A = \{a_1, a_2, \ldots, a_n\}$ of n integers, describe, in pseudo-code, an efficient method for computing each of partial sums $s_k = \sum_{i=1}^{k} a_i$, for $k = 1, 2, \ldots, n$. What is the running time of this method?

C-4.19 Draw a visual justification of Proposition 4.3 analogous to that of Figure 4.1(b) for the case when n is odd.

C-4.20 An array A contains $n-1$ unique integers in the range $[0, n-1]$, that is, there is one number from this range that is not in A. Design an $O(n)$-time algorithm for finding that number. You are only allowed to use $O(1)$ additional space besides the array A itself.

C-4.21 Let S be a set of n lines in the plane such that no two are parallel and no three meet in the same point. Show, by induction, that the lines in S determine $\Theta(n^2)$ intersection points.

C-4.22 Show that the summation $\sum_{i=1}^{n} \lceil \log_2 i \rceil$ is $O(n \log n)$.

C-4.23 An evil king has n bottles of wine, and a spy has just poisoned one of them. Unfortunately, they don't know which one it is. The poison is very deadly; just one drop diluted even a billion to one will still kill. Even so, it takes a full month for the poison to take effect. Design a scheme for determining exactly which one of the wine bottles was poisoned in just one month's time while expending $O(\log n)$ taste testers.

C-4.24 An array A contains n integers taken from the interval $[0, 4n]$, with repetitions allowed. Describe an efficient algorithm for determining an integer value k that occurs the most often in A. What is the running time of your algorithm?

C-4.25 Describe, in pseudo-code, a method for multiplying an $n \times m$ matrix A and an $m \times p$ matrix B. Recall that the product $C = AB$ is defined so that $C[i][j] = \sum_{k=1}^{m} A[i][k] \cdot B[k][j]$. What is the running time of your method?

C-4.26 Suppose each row of an $n \times n$ array A consists of 1's and 0's such that, in any row i of A, all the 1's come before any 0's. Also suppose that the number of 1's in row i is at least the number in row $i+1$, for $i = 0, 1, \ldots, n-2$. Assuming A is already in memory, describe a method running in $O(n)$ time (not $O(n^2)$) for counting the number of 1's in A.

C-4.27 Describe a recursive function for computing the nth ***Harmonic number***, $H_n = \sum_{i=1}^{n} 1/i$.

Projects

P-4.1 Implement prefixAverages1 and prefixAverages2 from Section 4.2.5, and perform an experimental analysis of their running times. Visualize their running times as a function of the input size with a log-log chart.

P-4.2 Perform a careful experimental analysis that compares the relative running times of the functions shown in Code Fragments 4.6.

P-4.3 Perform an experimental analysis to test the hypothesis that the STL function, sort, runs in $O(n \log n)$ time on average.

P-4.4 Perform an experimental analysis to determine the largest value of n for each of the three algorithms given in the chapter for solving the element uniqueness problem such that the given algorithm runs in one minute or less.

Chapter Notes

The big-Oh notation has prompted several comments about its proper use [15, 43, 58]. Knuth [59, 58] defines it using the notation $f(n) = O(g(n))$, but says this "equality" is only "one way." We have chosen to take a more standard view of equality and view the big-Oh notation as a set, following Brassard [15]. The reader interested in studying average-case analysis is referred to the book chapter by Vitter and Flajolet [101]. We found the story about Archimedes in [78]. For some additional mathematical tools, please refer to Appendix A.