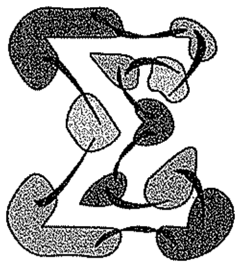
Chapter

1 Algorithm Analysis



Contents

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

The primary analysis tool we will 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. But focusing on running time as a primary measure of goodness implies that we will need to use at least a little mathematics to describe running times and compare algorithms.

We begin this chapter by describing the basic framework needed for analyzing algorithms, which includes the language for describing algorithms, the computational model that language is intended for, and the main factors we count when considering running time. We also include a brief discussion of how recursive algorithms are analyzed. In Section 1.2, we present the main notation we use to characterize running times—the so-called "big-Oh" notation. These tools comprise the main theoretical tools for designing and analyzing algorithms.

In Section 1.3, we take a short break from our development of the framework for algorithm analysis to review some important mathematical facts, including discussions of summations, logarithms, proof techniques, and basic probability. Given this background and our notation for algorithm analysis, we present some case studies on theoretical algorithm analysis in Section 1.4. We follow these examples in Section 1.5 by presenting an interesting analysis technique, known as amortization, which allows us to account for the group behavior of many individual operations. Finally, in Section 1.6, we conclude the chapter by discussing an important and practical analysis technique—experimentation. We discuss both the main principles of a good experimental framework as well as techniques for summarizing and characterizing data from an experimental analysis.

1.1 Methodologies for Analyzing Algorithms

The running time of an algorithm or data structure operation typically depends on a number of factors, so what should be the proper way of measuring it? 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. Such measurements can be taken in an accurate manner by using system calls that are built into the language or operating system for which the algorithm is written. In general, we are interested in determining the dependency of the running time on the size of the input. In order to determine this, we can perform several experiments on many different test inputs of various sizes. We can then 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 1.1.) 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, which is an approach we discuss in more detail in Section 1.6.

In general, the running time of an algorithm or data structure method increases with the input size, although it may also vary for distinct inputs of the same size. Also, the running time is affected by the hardware environment (processor, clock rate, memory, disk, etc.) and software environment (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 will be 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.

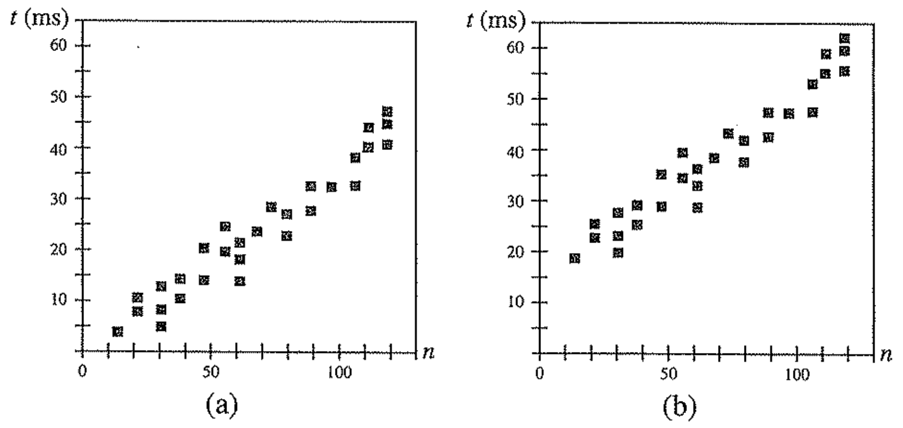


Figure 1.1: Results of an experimental study on the running time of an algorithm. A dot with coordinates (n,t) indicates that on an input of size n, the running time of the algorithm is t milliseconds (ms). (a) The algorithm executed on a fast computer; (b) the algorithm executed on a slow computer.

Requirements for a General Analysis Methodology

Experimental studies on running times are useful, as we explore in Section 1.6, but they have some limitations:

- Experiments can be done only on a limited set of test inputs, and care must be taken to make sure these are representative.
- It is difficult to compare the efficiency of two algorithms unless experiments on their running times have been performed in the same hardware and software environments.
- It is necessary to implement and execute an algorithm in order to study its running time experimentally.

Thus, while experimentation has an important role to play in algorithm analysis, it alone is not sufficient. Therefore, in addition to experimentation, we desire an analytic framework 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 in terms of the input size n. Typical functions that will be encountered include n and n^2. For example, we will write statements of the type "Algorithm A runs in time proportional to n," meaning that if we were to perform experiments, we would find that the actual running time of algorithm A on ***any*** input of size n never exceeds cn, where c is a constant that depends on the hardware and software environment used in the experiment. Given two algorithms A and B, where A runs in time proportional to n and B runs in time proportional to n^2, we will prefer A to B, since the function n grows at a smaller rate than the function n^2.

We are now ready to "roll up our sleeves" and start developing our methodology for algorithm analysis. There are several components to this methodology, including the following:

- A language for describing algorithms
- A computational model that algorithms execute within
- A metric for measuring algorithm running time
- An approach for characterizing running times, including those for recursive algorithms.

We describe these components in more detail in the remainder of this section.

1.1.1 Pseudo-Code

Programmers are often asked to describe algorithms in a way that is intended for human eyes only. Such descriptions are not computer programs, but are more structured than usual prose. They also facilitate the high-level analysis of a data structure or algorithm. We call these descriptions ***pseudo-code***.

An Example of Pseudo-Code

The array-maximum problem is the simple problem of finding the maximum element in an array A storing n integers. To solve this problem, we can use an algorithm called arrayMax, which scans through the elements of A using a **for** loop.

The pseudo-code description of algorithm arrayMax is shown in Algorithm 1.2.

```
Algorithm arrayMax($A, n$):
   Input: An array $A$ storing $n \geq 1$ integers.
   Output: The maximum element in $A$.
  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ to $n-1$ do
     if $currentMax < A[i]$ then
        $currentMax \leftarrow A[i]$
  return $currentMax$
```

Algorithm 1.2: Algorithm arrayMax.

Note that the pseudo-code is more compact than an equivalent actual software code fragment would be. In addition, the pseudo-code is easier to read and understand.

Using Pseudo-Code to Prove Algorithm Correctness

By inspecting the pseudo-code, we can argue about the correctness of algorithm arrayMax with a simple argument. Variable *currentMax* starts out being equal to the first element of A. We claim that at the beginning of the ith iteration of the loop, *currentMax* is equal to the maximum of the first i elements in A. Since we compare *currentMax* to $A[i]$ in iteration i, if this claim is true before this iteration, it will be true after it for $i+1$ (which is the next value of counter i). Thus, after $n-1$ iterations, *currentMax* will equal the maximum element in A. As with this example, we want our pseudo-code descriptions to always be detailed enough to fully justify the correctness of the algorithm they describe, while being simple enough for human readers to understand.

What Is Pseudo-Code?

Pseudo-code is a mixture of natural language and high-level programming constructs that describe the main ideas behind a generic implementation of a data structure or algorithm. There really is no precise definition of the ***pseudo-code*** language, however, because of its reliance on natural language. At the same time, to help achieve clarity, pseudo-code mixes natural language with standard programming language constructs. The programming language constructs we choose are those consistent with modern high-level languages such as C, C++, and Java. These constructs include the following:

- ***Expressions:*** We use standard mathematical symbols to express numeric and Boolean expressions. We use the left arrow sign ($\leftarrow$) as the assignment operator in assignment statements (equivalent to the $=$ operator in C, C++, and Java) and we use the equal sign ($=$) as the equality relation in Boolean expressions (equivalent to the "==" relation in C, C++, and Java).
- ***Method declarations:*** **Algorithm** name($param1, param2, \ldots$) declares a new method "name" and its parameters.
- ***Decision structures:*** **if** condition **then** true-actions [**else** false-actions]. We use indentation to indicate what actions should be included in the true-actions and false-actions.
- ***While-loops:*** **while** condition **do** actions. We use indentation to indicate what actions should be included in the loop actions.
- ***Repeat-loops:*** **repeat** actions **until** condition. We use indentation to indicate what actions should be included in the loop actions.
- ***For-loops:*** **for** variable-increment-definition **do** actions. We use indentation to indicate what actions should be included among the loop actions.
- ***Array indexing:*** $A[i]$ represents the ith cell in the array A. The cells of an n-celled array A are indexed from $A[0]$ to $A[n-1]$ (consistent with C, C++, and Java).
- ***Method calls:*** object.method(args) (object is optional if it is understood).
- ***Method returns:*** **return** value. This operation returns the value specified to the method that called this one.

When we write pseudo-code, we must keep in mind that we are writing for a human reader, not a computer. Thus, we should strive to communicate high-level ideas, not low-level implementation details. At the same time, we should not gloss over important steps. Like many forms of human communication, finding the right balance is an important skill that is refined through practice.

Now that we have developed a high-level way of describing algorithms, let us next discuss how we can analytically characterize algorithms written in pseudo-code.

1.1.2 The Random Access Machine (RAM) Model

As we 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 take the following more analytic approach directly on the high-level code or pseudo-code. We define a set of high-level ***primitive operations*** that are largely independent from the programming language used and can be identified also in the pseudo-code. Primitive operations include the following:

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

Specifically, a primitive operation corresponds to a low-level instruction with an execution time that depends on the hardware and software environment but is nevertheless constant. Instead of trying to determine the specific execution time of each primitive operation, we will simply ***count*** how many primitive operations are executed, and use this number t as a high-level estimate of the running time of the algorithm. This operation count will correlate to an actual running time in a specific hardware and software environment, for 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 will be fairly similar. Thus, the number, t, of primitive operations an algorithm performs will be proportional to the actual running time of that algorithm.

RAM Machine Model Definition

This approach of simply counting primitive operations gives rise to a computational model called the ***Random Access Machine*** (RAM). This model, which should not be confused with "random access memory," views a computer simply as a CPU connected to a bank of memory cells. Each memory cell stores a word, which can be a number, a character string, or an address, that is, the value of a base type. The term "random access" refers to the ability of the CPU to access an arbitrary memory cell with one primitive operation. To keep the model simple, we do not place any specific limits on the size of numbers that can be stored in words of memory. We assume the CPU in the RAM model can perform any primitive operation in a constant number of steps, which do not depend on the size of the input. Thus, an accurate bound on the number of primitive operations an algorithm performs corresponds directly to the running time of that algorithm in the RAM model.

1.1.3 Counting Primitive Operations

We now show how to count the number of primitive operations executed by an algorithm, using as an example algorithm arrayMax, whose pseudo-code was given back in Algorithm 1.2. We do this analysis by focusing on each step of the algorithm and counting the primitive operations that it takes, taking into consideration that some operations are repeated, because they are enclosed in the body of a loop.

- Initializing the variable *currentMax* to $A[0]$ corresponds to two primitive operations (indexing into an array and assigning a value to a variable) and is executed only once at the beginning of the algorithm. Thus, it contributes two units to the count.
- At the beginning of the for loop, counter i is initialized to 1. This action corresponds to executing one primitive operation (assigning a value to a variable).
- Before entering the body of the for loop, condition $i < n$ is verified. This action corresponds to executing one primitive instruction (comparing two numbers). Since counter i starts at 0 and is incremented by 1 at the end of each iteration of the loop, the comparison $i < n$ is performed n times. Thus, it contributes n units to the count.
- The body of the for loop is executed $n-1$ times (for values $1, 2, \ldots, n-1$ of the counter). At each iteration, $A[i]$ is compared with *currentMax* (two primitive operations, indexing and comparing), $A[currentMax]$ is possibly assigned to *currentMax* (two primitive operations, indexing and assigning), and the counter i is incremented (two primitive operations, summing and assigning). Hence, at each iteration of the loop, either four or six primitive operations are performed, depending on whether $A[i] \leq currentMax$ or $A[i] > currentMax$. Therefore, the body of the loop contributes between $4(n-1)$ and $6(n-1)$ units to the count.
- Returning the value of variable *currentMax* corresponds to one primitive operation, and is executed only once.

To summarize, the number of primitive operations $t(n)$ executed by algorithm arrayMax is at least

$$2+1+n+4(n-1)+1=5n$$

and at most

$$2+1+n+6(n-1)+1=7n-2.$$

The best case ($t(n)=5n$) occurs when $A[0]$ is the maximum element, so that variable *currentMax* is never reassigned. The worst case ($t(n)=7n-2$) occurs when the elements are sorted in increasing order, so that variable *currentMax* is reassigned at each iteration of the for loop.

Average-Case and Worst-Case Analysis

Like the arrayMax method, an algorithm may run faster on some inputs than it does on others. In such cases we may wish to express the running time of such an algorithm as an average taken over all possible inputs. Although such an ***average case*** analysis would often be valuable, it is typically quite challenging. It requires us to define a probability distribution on the set of inputs, which is typically a difficult task. Figure 1.3 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"?

An average-case analysis also typically requires that we calculate expected running times based on a given input distribution. Such an analysis often requires heavy mathematics and probability theory.

Therefore, except for experimental studies or the analysis of algorithms that are themselves randomized, we will, for the remainder of this book, typically characterize running times in terms of the ***worst case***. We say, for example, that algorithm arrayMax executes $t(n) = 7n - 2$ primitive operations ***in the worst case***, meaning that the maximum number of primitive operations executed by the algorithm, taken over all inputs of size n, is $7n - 2$.

This type of analysis is much easier than an average-case analysis, as it does not require probability theory; it just requires the ability to identify the worst-case input, which is often straightforward. In addition, taking a worst-case approach can actually lead to better algorithms. Making the standard of success that of having an algorithm perform well in the worst case necessarily requires that it perform well on ***every*** input. That is, designing for the worst case can lead to stronger algorithmic "muscles," much like a track star who always practices by running up hill.

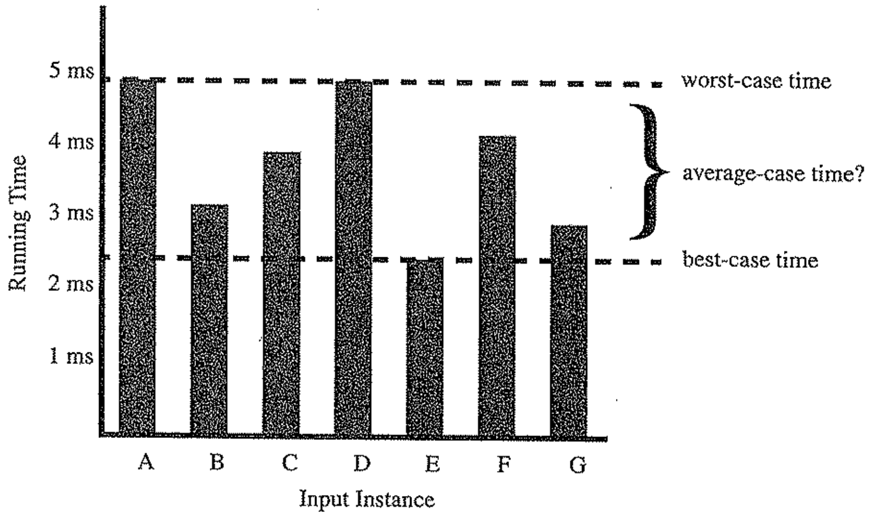


Figure 1.3: The difference between best-case and worst-case time. Each bar represents the running time of some algorithm on a different possible input.

1.1.4 Analyzing Recursive Algorithms

Iteration is not the only interesting way of solving a problem. Another useful technique, which is employed by many algorithms, is to use ***recursion***. In this technique, we define a procedure P that is allowed to make calls to itself as a subroutine, provided those calls to P are for solving subproblems of smaller size. The subroutine calls to P on smaller instances are called "recursive calls." A recursive procedure should always define a ***base case***, which is small enough that the algorithm can solve it directly without using recursion.

We give a recursive solution to the array maximum problem in Algorithm 1.4. This algorithm first checks if the array contains just a single item, which in this case must be the maximum; hence, in this simple base case we can immediately solve the problem. Otherwise, the algorithm recursively computes the maximum of the first $n-1$ elements in the array and then returns the maximum of this value and the last element in the array.

As with this example, recursive algorithms are often quite elegant. Analyzing the running time of a recursive algorithm takes a bit of additional work, however. In particular, to analyze such a running time, we use a ***recurrence equation***, which defines mathematical statements that the running time of a recursive algorithm must satisfy. We introduce a function $T(n)$ that denotes the running time of the algorithm on an input of size n, and we write equations that $T(n)$ must satisfy. For example, we can characterize the running time, $T(n)$, of the recursiveMax algorithm as

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1)+7 & \text{otherwise,} \end{cases}$$

assuming that we count each comparison, array reference, recursive call, max calculation, or **return** as a single primitive operation. Ideally, we would like to characterize a recurrence equation like that above in ***closed form***, where no references to the function T appear on the righthand side. For the recursiveMax algorithm, it isn't too hard to see that a closed form would be $T(n) = 7(n-1)+3 = 7n-2$. In general, determining closed form solutions to recurrence equations can be much more challenging than this, and we study some specific examples of recurrence equations in Chapter 4, when we study some sorting and selection algorithms. We study methods for solving recurrence equations of a general form in Section 5.2.

```
Algorithm recursiveMax($A,n$):
    Input: An array $A$ storing $n \geq 1$ integers.
    Output: The maximum element in $A$.
  if $n = 1$ then
      return $A[0]$
  return max{recursiveMax($A, n-1$), $A[n-1]$}
```

Algorithm 1.4: Algorithm recursiveMax.

1.2 Asymptotic Notation

We have clearly gone into laborious detail for evaluating the running time of such a simple algorithm as arrayMax and its recursive cousin, recursiveMax. Such an approach would clearly prove cumbersome if we had to perform it for more complicated algorithms. In general, each step in a pseudo-code description and each statement in a high-level language implementation corresponds to a small number of primitive operations that does not depend on the input size. Thus, we can perform a simplified analysis that estimates the number of primitive operations executed up to a constant factor, by counting the steps of the pseudo-code or the statements of the high-level language executed. Fortunately, there is a notation that allows us to characterize the main factors affecting an algorithm's running time without going into all the details of exactly how many primitive operations are performed for each constant-time set of instructions.

1.2.1 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 $f(n) \leq cg(n)$ for every integer $n \geq n_0$. 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*** $g(n)$." (This definition is illustrated in Figure 1.5.)

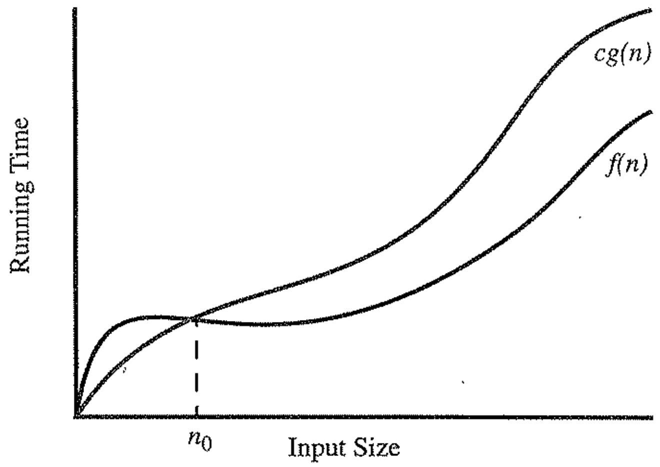


Figure 1.5: Illustrating the "big-Oh" notation. The function $f(n)$ is $O(g(n))$, for $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

Example 1.1: *$7n-2$ is $O(n)$.*

Proof: *By the big-Oh definition, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $7n-2 \leq cn$ for every integer $n \geq n_0$. It is easy to see that a possible choice is $c = 7$ and $n_0 = 1$. Indeed, this is one of infinitely many choices available because any real number greater than or equal to 7 will work for c, and any integer greater than or equal to 1 will work for n_0.* ■

The big-Oh notation allows us to say that a function of n is "less than or equal to" another function (by the inequality "$\leq$" in the definition), up to a constant factor (by the constant c in the definition) and in the ***asymptotic*** sense as n grows toward infinity (by the statement "$n \geq n_0$" in the definition).

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 usually defined as an intuitive notion of the "size" of the problem. For example, if we are interested in finding the largest element in an array of integers (see arrayMax given in Algorithm 1.2), it would be most natural to let n denote the number of elements of the array. For example, we can write the following precise statement on the running time of algorithm arrayMax from Algorithm 1.2.

Theorem 1.2: *The running time of algorithm* arrayMax *for computing the maximum element in an array of n integers is $O(n)$.*

Proof: As shown in Section 1.1.3, the number of primitive operations executed by algorithm arrayMax is at most $7n-2$. We may therefore apply the big-Oh definition with $c = 7$ and $n_0 = 1$ and conclude that the running time of algorithm arrayMax is $O(n)$. ■

Let us consider a few additional examples that illustrate the big-Oh notation.

Example 1.3: *$20n^3 + 10n\log n + 5$ is $O(n^3)$.*

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

In fact, any polynomial $a_k n^k + a_{k-1}n^{k-1} + \cdots + a_0$ will always be $O(n^k)$.

Example 1.4: *$3\log n + \log\log n$ is $O(\log n)$.*

Proof: *$3\log n + \log\log n \leq 4\log n$, for $n \geq 2$. Note that $\log\log n$ is not even defined for $n = 1$. That is why we use $n \geq 2$.* ■

Example 1.5: *2^{100} is $O(1)$.*

Proof: *$2^{100} \leq 2^{100} \cdot 1$, for $n \geq 1$. Note that variable n does not appear in the inequality, since we are dealing with constant-valued functions.* ■

Example 1.6: *$5/n$ is $O(1/n)$.*

Proof: *$5/n \leq 5(1/n)$, for $n \geq 1$ (even though this is actually a* **decreasing** *function).* ■

In general, we should use the big-Oh notation to characterize a function as closely as possible. While it is true that $f(n) = 4n^3 + 3n^{4/3}$ is $O(n^5)$, 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."

Instead of always applying the big-Oh definition directly to obtain a big-Oh characterization, we can use the following rules to simplify notation.

Theorem 1.7: *Let $d(n)$, $e(n)$, $f(n)$, and $g(n)$ be functions mapping nonnegative integers to nonnegative reals. Then*

1. *If $d(n)$ is $O(f(n))$, then $ad(n)$ is $O(f(n))$, for any constant $a > 0$.*
2. *If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)+e(n)$ is $O(f(n)+g(n))$.*
3. *If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)e(n)$ is $O(f(n)g(n))$.*
4. *If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $d(n)$ is $O(g(n))$.*
5. *If $f(n)$ is a polynomial of degree d (that is, $f(n) = a_0 + a_1 n + \cdots + a_d n^d$), then $f(n)$ is $O(n^d)$.*
6. *n^x is $O(a^n)$ for any fixed $x > 0$ and $a > 1$.*
7. *$\log n^x$ is $O(\log n)$ for any fixed $x > 0$.*
8. *$\log^x n$ is $O(n^y)$ for any fixed constants $x > 0$ and $y > 0$.*

It is 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***.

Example 1.8: $2n^3 + 4n^2 \log n$ *is* $O(n^3)$.

Proof: *We can apply the rules of Theorem 1.7 as follows:*

- *$\log n$ is $O(n)$ (Rule 8).*
- *$4n^2 \log n$ is $O(4n^3)$ (Rule 3).*
- *$2n^3 + 4n^2 \log n$ is $O(2n^3 + 4n^3)$ (Rule 2).*
- *$2n^3 + 4n^3$ is $O(n^3)$ (Rule 5 or Rule 1).*
- *$2n^3 + 4n^2 \log n$ is $O(n^3)$ (Rule 4).* ■

Some functions appear often in the analysis of algorithms and data structures, and we often use special terms to refer to them. Table 1.6 shows some terms commonly used in algorithm analysis.

logarithmic	**linear**	**quadratic**	**polynomial**	**exponential**
$O(\log n)$	$O(n)$	$O(n^2)$	$O(n^k)$ $(k \geq 1)$	$O(a^n)$ $(a > 1)$

Table 1.6: Terminology for classes of functions.

Using the Big-Oh Notation

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 completely correct to say "$f(n) = O(g(n))$" (with the usual understanding of the "$=$" relation), and it is actually incorrect to say "$f(n) \geq O(g(n))$" or "$f(n) > O(g(n))$." It is best to say "$f(n)$ ***is*** $O(g(n))$." For the more mathematically inclined, it is also correct to say,

$$\text{“}f(n) \in O(g(n)),\text{”}$$

for the big-Oh notation is, technically speaking, denoting a whole collection of functions.

Even with this interpretation, there is considerable freedom in how we can use arithmetic operations with the big-Oh notation, provided the connection to the definition of the big-Oh is clear. For instance, we can say,

$$\text{“}f(n) \text{ is } g(n) + O(h(n)),\text{”}$$

which would mean that there are constants $c > 0$ and $n_0 \geq 1$ such that $f(n) \leq g(n) + ch(n)$ for $n \geq n_0$. As in this example, we may sometimes wish to give the exact leading term in an asymptotic characterization. In that case, we would say that "$f(n)$ is $g(n) + O(h(n))$," where $h(n)$ grows slower than $g(n)$. For example, we could say that $2n\log n + 4n + 10\sqrt{n}$ is $2n\log n + O(n)$.

1.2.2 "Relatives" of the Big-Oh

Just as the big-Oh notation provides an asymptotic way of saying that a function is "less than or equal to" another function, there are other notations that provide asymptotic ways of making other types of comparisons.

Big-Omega and Big-Theta

Let $f(n)$ and $g(n)$ be functions mapping 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 $f(n) \geq cg(n)$, for $n \geq n_0$. This definition allows us to say asymptotically that one function is greater than or equal to another, up to a constant factor. Likewise, 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 $c'g(n) \leq f(n) \leq c''g(n)$, for $n \geq n_0$.

The big-Theta allows us to say that two functions are asymptotically equal, up to a constant factor. We consider some examples of these notations below.

Example 1.9: *$3\log n + \log\log n$ is $\Omega(\log n)$.*

Proof: *$3\log n + \log\log n \geq 3\log n$, for $n \geq 2$.* ▪

This example shows that lower order terms are not dominant in establishing lower bounds with the big-Omega notation. Thus, as the next example sums up, lower order terms are not dominant in the big-Theta notation either.

Example 1.10: *$3\log n + \log\log n$ is $\Theta(\log n)$.*

Proof: *This follows from Examples 1.4 and 1.9.* ▪

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 $\Theta(n)$, if this is the running time of an algorithm being compared to one whose running time is $10n\log n$, we should prefer the $\Theta(n\log n)$ 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 above observation 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 $\Theta(2^n)$ time should never be considered efficient. This fact is illustrated by 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. But try to imagine the sight of 2^{64} grains stacked on the last square! In fact, this number cannot even be represented as a standard long integer in most programming languages.

Therefore, if we must draw a line between efficient and inefficient algorithms, it is natural to make this distinction be that between those algorithms running in polynomial time and those requiring exponential time. That is, make the distinction between algorithms with a running time that is $O(n^k)$, for some constant $k \geq 1$, and those with a running time that is $\Theta(c^n)$, for some constant $c > 1$. 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 $\Theta(n^{100})$ 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.

"Distant Cousins" of the Big-Oh: Little-Oh and Little-Omega

There are also some ways of saying that one function is strictly less than or strictly greater than another asymptotically, but these are not used as often as the big-Oh, big-Omega, and big-Theta. Nevertheless, for the sake of completeness, we give their definitions as well.

Let $f(n)$ and $g(n)$ be functions mapping integers to real numbers. We say that $f(n)$ is $o(g(n))$ (pronounced "$f(n)$ is little-oh of $g(n)$") if, for any constant $c > 0$, there is a constant $n_0 > 0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$. Likewise, we say that $f(n)$ is $\omega(g(n))$ (pronounced "$f(n)$ is little-omega of $g(n)$") if $g(n)$ is $o(f(n))$, that is, if, for any constant $c > 0$, there is a constant $n_0 > 0$ such that $g(n) \leq cf(n)$ for $n \geq n_0$. Intuitively, $o(\cdot)$ is analogous to "less than" in an asymptotic sense, and $\omega(\cdot)$ is analogous to "greater than" in an asymptotic sense.

Example 1.11: *The function $f(n) = 12n^2 + 6n$ is $o(n^3)$ and $\omega(n)$.*

Proof: *Let us first show that $f(n)$ is $o(n^3)$. Let $c > 0$ be any constant. If we take $n_0 = (12+6)/c$, then, for $n \geq n_0$, we have*

$$cn^3 \geq 12n^2 + 6n^2 \geq 12n^2 + 6n.$$

Thus, $f(n)$ is $o(n^3)$.

To show that $f(n)$ is $\omega(n)$, let $c > 0$ again be any constant. If we take $n_0 = c/12$, then, for $n \geq n_0$, we have

$$12n^2 + 6n \geq 12n^2 \geq cn.$$

Thus, $f(n)$ is $\omega(n)$. ■

For the reader familiar with limits, we note that $f(n)$ is $o(g(n))$ if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0,$$

provided this limit exists. The main difference between the little-oh and big-Oh notions is that $f(n)$ is $O(g(n))$ if ***there exist*** constants $c > 0$ and $n_0 \geq 1$ such that $f(n) \leq cg(n)$, for $n \geq n_0$; whereas $f(n)$ is $o(g(n))$ if ***for all*** constants $c > 0$ there is a constant n_0 such that $f(n) \leq cg(n)$, for $n \geq n_0$. Intuitively, $f(n)$ is $o(g(n))$ if $f(n)$ becomes insignificant compared to $g(n)$ as n grows toward infinity. As previously mentioned, asymptotic notation is useful because it allows us to concentrate on the main factor determining a function's growth.

To summarize, the asymptotic notations of big-Oh, big-Omega, and big-Theta, as well as little-oh and little-omega, 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.

1.2.3 The Importance of Asymptotics

Asymptotic notation has many important benefits, which might not be immediately obvious. Specifically, we illustrate one important aspect of the asymptotic viewpoint in Table 1.7. This table explores the maximum size allowed for an input instance for various running times to be solved in 1 second, 1 minute, and 1 hour, assuming each operation can be processed in 1 microsecond (1 μs). It also shows the importance of algorithm design, because an algorithm with an asymptotically slow running time (for example, one that is $O(n^2)$) is beaten in the long run by an algorithm with an asymptotically faster running time (for example, one that is $O(n \log n)$), even if the constant factor for the faster algorithm is worse.

Running Time	Maximum Problem Size (n)		
	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$20n\lceil \log n \rceil$	4,096	166,666	7,826,087
$2n^2$	707	5,477	42,426
n^4	31	88	244
2^n	19	25	31

Table 1.7: Maximum size of a problem that can be solved in one second, one minute, and one 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 1.8, even if we achieve a dramatic speedup 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
$400n$	$256m$
$20n\lceil \log n \rceil$	approx. $256((\log m)/(7+\log m))m$
$2n^2$	$16m$
n^4	$4m$
2^n	$m+8$

Table 1.8: Increase in the maximum size of a problem that can be solved in a certain fixed amount of time, by using a computer that is 256 times faster than the previous one, for various running times of the algorithm. Each entry is given as a function of m, the previous maximum problem size.

Ordering Functions by Their Growth Rates

Suppose two algorithms solving the same problem are available: an algorithm A, which has a running time of $\Theta(n)$, and an algorithm B, which has a running time of $\Theta(n^2)$. Which one is better? The little-oh notation says that n is $o(n^2)$, which implies that algorithm A is ***asymptotically better*** than algorithm B, although for a given (small) value of n, it is possible for algorithm B to have lower running time than algorithm A. Still, in the long run, as shown in the above tables, the benefits of algorithm A over algorithm B will become clear.

In general, we can use the little-oh notation to order classes of functions by asymptotic growth rate. In Table 1.9, we show a list of functions ordered by increasing growth rate, that is, if a function $f(n)$ precedes a function $g(n)$ in the list, then $f(n)$ is $o(g(n))$.

Functions Ordered by Growth Rate
$\log n$
$\log^2 n$
$\sqrt{n}$
n
$n \log n$
n^2
n^3
2^n

Table 1.9: An ordered list of simple functions. Note that, using common terminology, one of the above functions is logarithmic, two are polylogarithmic, three are sublinear, one is linear, one is quadratic, one is cubic, and one is exponential.

In Table 1.10, we illustrate the difference in the growth rate of all but one of the functions shown in Table 1.9.

n	$\log n$	$\sqrt{n}$	n	$n\log n$	n^2	n^3	2^n
2	1	1.4	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	2.8	8	24	64	512	256
16	4	4	16	64	256	4,096	65,536
32	5	5.7	32	160	1,024	32,768	4,294,967,296
64	6	8	64	384	4,096	262,144	1.84×10^{19}
128	7	11	128	896	16,384	2,097,152	3.40×10^{38}
256	8	16	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	23	512	4,608	262,144	134,217,728	1.34×10^{154}
1,024	10	32	1,024	10,240	1,048,576	1,073,741,824	1.79×10^{308}

Table 1.10: Growth of several functions.

1.3 A Quick Mathematical Review

In this section, we briefly review some of the fundamental concepts from discrete mathematics that will arise in several of our discussions. In addition to these fundamental concepts, Appendix A includes a list of other useful mathematical facts that apply in the context of data structure and algorithm analysis.

1.3.1 Summations

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

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

Summations arise in data structure and algorithm analysis because the running times of loops naturally give rise to summations. For example, a summation that often arises in data structure and algorithm analysis is the geometric summation.

Theorem 1.12: *For any integer $n \geq 0$ and any real number $0 < a \neq 1$, consider*

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

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

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

Summations as shown in Theorem 1.12 are called ***geometric*** summations, because each term is geometrically larger than the previous one if $a > 1$. That is, the terms in such a geometric summation exhibit exponential growth. For example, everyone working in computing should know that

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

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

Another summation that arises in several contexts is

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n.$$

This summation often arises in the analysis of loops in cases where the number of operations performed inside the loop increases by a fixed, constant amount with each iteration. This summation also has an interesting history. In 1787, a German elementary schoolteacher decided to keep his 9- and 10-year-old pupils occupied with the task of adding up all the numbers from 1 to 100. But almost immediately after giving this assignment, one of the children claimed to have the answer—5,050.

That elementary school student was none other than Karl Gauss, who would grow up to be one of the greatest mathematicians of the 19th century. It is widely suspected that young Gauss derived the answer to his teacher's assignment using the following identity.

Theorem 1.13: *For any integer $n \geq 1$, we have*

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

Proof: We give two "visual" justifications of Theorem 1.13 in Figure 1.11, both of which are based on computing the area of a collection of rectangles representing the numbers 1 through n. In Figure 1.11a we draw a big triangle over an ordering of the rectangles, noting that the area of the rectangles is the same as that of the big triangle ($n^2/2$) plus that of n small triangles, each of area $1/2$. In Figure 1.11b, which applies when n is even, we note that 1 plus n is $n+1$, as is 2 plus $n-1$, 3 plus $n-2$, and so on. There are $n/2$ such pairings. ▨

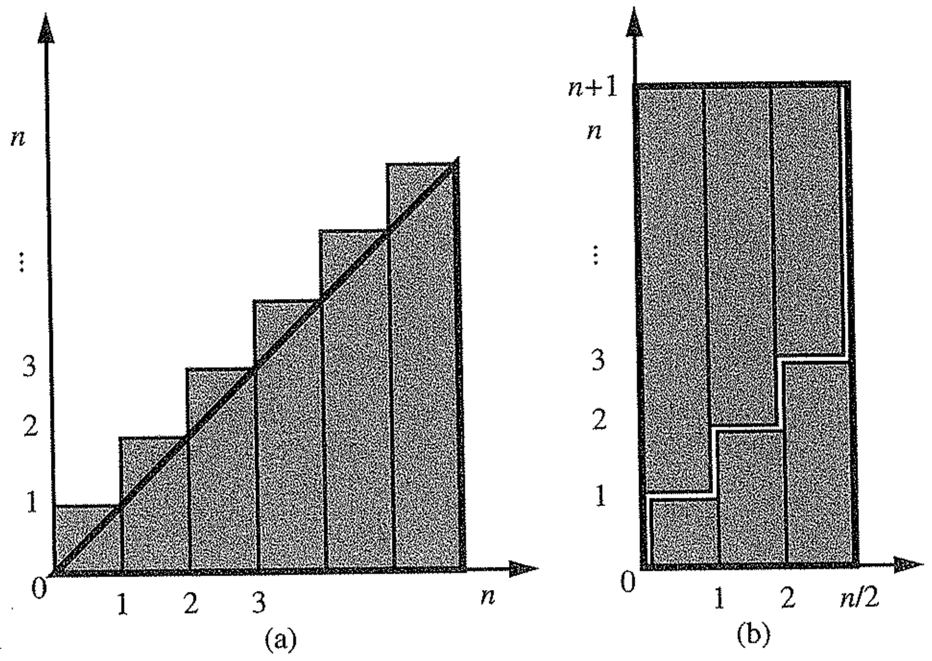


Figure 1.11: Visual justifications of Theorem 1.13. Both illustrations visualize the identity in terms of the total area covered by n unit-width rectangles with heights $1, 2, \ldots, n$. In (a) the rectangles are shown to cover a big triangle of area $n^2/2$ (base n and height n) plus n small triangles 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 $n+1$.

1.3.2 Logarithms and Exponents

One of the interesting and sometimes even surprising aspects of the analysis of data structures and algorithms is the ubiquitous presence of logarithms and exponents, where we say

$$\log_b a = c \qquad \text{if} \qquad a = b^c.$$

As is the custom in the computing literature, we omit writing the base b of the logarithm when $b = 2$. For example, $\log 1024 = 10$.

There are a number of important rules for logarithms and exponents, including the following:

Theorem 1.14: *Let a, b, and c be positive real numbers. 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. $\log_b a = (\log_c a)/\log_c b$
5. $b^{\log_c a} = a^{\log_c b}$
6. $(b^a)^c = b^{ac}$
7. $b^a b^c = b^{a+c}$
8. $b^a/b^c = b^{a-c}$.

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

Example 1.15: *We illustrate some interesting cases when the base of a logarithm or exponent is 2. The rules cited refer to Theorem 1.14.*

- $\log(2n\log n) = 1 + \log n + \log\log n$, *by rule 1 (twice)*
- $\log(n/2) = \log n - \log 2 = \log n - 1$, *by rule 2*
- $\log\sqrt{n} = \log(n)^{1/2} = (\log n)/2$, *by rule 3*
- $\log\log\sqrt{n} = \log(\log n)/2 = \log\log n - 1$, *by rules 2 and 3*
- $\log_4 n = (\log n)/\log 4 = (\log n)/2$, *by rule 4*
- $\log 2^n = n$, *by rule 3*
- $2^{\log n} = n$, *by rule 5*
- $2^{2\log n} = (2^{\log n})^2 = n^2$, *by rules 5 and 6*
- $4^n = (2^2)^n = 2^{2n}$, *by rule 6*
- $n^2 2^{3\log n} = n^2 \cdot n^3 = n^5$, *by rules 5, 6, and 7*
- $4^n/2^n = 2^{2n}/2^n = 2^{2n-n} = 2^n$, *by rules 6 and 8*

The Floor and Ceiling Functions

One additional comment concerning logarithms is in order. The value of a logarithm is typically not an integer, yet the running time of an algorithm is typically expressed by means of an integer quantity, such as the number of operations performed. Thus, an algorithm analysis may sometimes involve the use of the so-called "floor" and "ceiling" functions, 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.

These functions give us a way to convert real-valued functions into integer-valued functions. Even so, functions used to analyze data structures and algorithms are often expressed simply as real-valued functions (for example, $n \log n$ or $n^{3/2}$). We should read such a running time as having a "big" ceiling function surrounding it.[1]

1.3.3 Simple Justification Techniques

We will sometimes wish to make strong claims about a certain data structure or algorithm. We may, for example, wish to show that our algorithm 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.

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 need only 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 need to only produce a particular x from S that does not have property P. Such an instance is called a ***counterexample***.

Example 1.16: *A certain 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.*

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

[1]Real-valued running-time functions are almost always used in conjunction with the asymptotic notation described in Section 1.2, for which the use of the ceiling function would usually be redundant anyway. (See Exercise R-1.24.)

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 instead establish that "if q is not true, then p is not true." 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 1.17: *If ab is odd, then a is odd or b is even.*

Proof: *To justify this claim, consider the contrapositive, "If a is even and b is odd, then ab is even." So, suppose $a = 2i$, for some integer i. Then $ab = (2i)b = 2(ib)$; hence, ab is even.* ■

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."

Another negative justification technique is proof by ***contradiction***, which also often involves using DeMorgan's Law. In applying the proof 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 1.18: *If ab is odd, then a is odd or b is even.*

Proof: *Let ab be odd. We wish to show that a is odd or b is even. So, with the hope of leading to a contradiction, let us assume the opposite, namely, suppose a is even and b is odd. 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 or b is even.* ■

Induction

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.

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 proof 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 proof by induction.

Example 1.19: *Consider the Fibonacci sequence: $F(1) = 1$, $F(2) = 2$, and $F(n) = F(n-1) + F(n-2)$ for $n > 2$. We claim that $F(n) < 2^n$.*

Proof: *We will 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 $F(n) < 2^{n-1} + 2^{n-2}$. In addition,*

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

This completes the proof. ■

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

Theorem 1.20: *(which is the same as Theorem 1.13)*

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

Proof: We will 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

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

This completes the proof. ■

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 begins, then one can show that $\mathcal{S}_i$ will be true after iteration i is over.
> 3. The final statement, $\mathcal{S}_k$, implies the statement $\mathcal{S}$ that we wish to justify as being true.

We have, in fact, already seen the loop-invariant justification technique at work in Section 1.1.1 (for the correctness of arrayMax), but let us nevertheless give one more example here. In particular, let us consider applying the loop invariant method to justify the correctness of Algorithm arrayFind, shown in Algorithm 1.12, which searches for an element x in an array A.

To show arrayFind to be correct, we use a loop invariant argument. That is, we inductively define statements, $\mathcal{S}_i$, for $i = 0, 1, \ldots, n$, that lead to the correctness of arrayFind. Specifically, we claim the following to be true at the beginning of iteration i:

$\mathcal{S}_i$: 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. 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 $\mathcal{S}_i$ will be true for this new value of i, for the beginning of the next iteration. If the while-loop terminates without ever returning an index in A, then $\mathcal{S}_n$ is true—there are no elements of A equal to x. Therefore, the algorithm is correct to return the nonindex value -1, as required.

```
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 $x$.
  $i \leftarrow 0$
  while $i < n$ do
    if $x = A[i]$ then
      return $i$
    else
      $i \leftarrow i + 1$
  return $-1$
```

Algorithm 1.12: Algorithm arrayFind.

1.3.4 Basic Probability

When we analyze algorithms that use randomization or if we wish to analyze the average-case performance of an algorithm, then we need to use some basic facts from probability theory. The most basic is that any statement about a probability is defined upon a ***sample space*** S, which is defined as the set of all possible outcomes from some experiment. We leave the terms "outcomes" and "experiment" undefined in any formal sense, however.

Example 1.21: *Consider an experiment that consists of the outcome from flipping a coin five times. This sample space has 2^5 different outcomes, one for each different ordering of possible flips that can occur.*

Sample spaces can also be infinite, as the following example illustrates.

Example 1.22: *Consider an experiment that consists of flipping a coin until it comes up heads. This sample space is infinite, with each outcome being a sequence of i tails followed by a single flip that comes up heads, for $i \in \{0,1,2,3,\ldots\}$.*

A ***probability space*** is a sample space S together with a probability function, Pr, that maps subsets of S to real numbers in the interval $[0,1]$. It captures mathematically the notion of the probability of certain "events" occurring. Formally, each subset A of S is called an ***event***, and the probability function Pr is assumed to possess the following basic properties with respect to events defined from S:

1. $\Pr(\emptyset) = 0$.
2. $\Pr(S) = 1$.
3. $0 \le \Pr(A) \le 1$, for any $A \subseteq S$.
4. If $A, B \subseteq S$ and $A \cap B = \emptyset$, then $\Pr(A \cup B) = \Pr(A) + \Pr(B)$.

Independence

Two events A and B are ***independent*** if

$$\Pr(A \cap B) = \Pr(A) \cdot \Pr(B).$$

A collection of events $\{A_1, A_2, \ldots, A_n\}$ is ***mutually independent*** if

$$\Pr(A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}) = \Pr(A_{i_1})\Pr(A_{i_2})\cdots\Pr(A_{i_k}).$$

for any subset $\{A_{i_1}, A_{i_2}, \ldots, A_{i_k}\}$.

Example 1.23: *Let A be the event that the roll of a die is a 6, let B be the event that the roll of a second die is a 3, and let C be the event that the sum of these two dice is a 10. Then A and B are independent events, but C is not independent with either A or B.*

Conditional Probability

The ***conditional probability*** that an event A occurs, given an event B, is denoted as $\Pr(A|B)$, and is defined as

$$\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)},$$

assuming that $\Pr(B) > 0$.

Example 1.24: *Let A be the event that a roll of two dice sums to 10, and let B be the event that the roll of the first die is a 6. Note that $\Pr(B) = 1/6$ and that $\Pr(A \cap B) = 1/36$, for there is only one way two dice can sum to 10 if the first one is a 6 (namely, if the second is a 4). Thus, $\Pr(A|B) = (1/36)/(1/6) = 1/6$.*

Random Variables and Expectation

An elegant way for dealing with events is in terms of ***random variables***. Intuitively, random variables are variables whose values depend upon the outcome of some experiment. Formally, a ***random variable*** is a function X that maps outcomes from some sample space S to real numbers. An ***indicator random variable*** is a random variable that maps outcomes to the set $\{0,1\}$. Often in algorithm analysis we use a random variable X that has a discrete set of possible outcomes to characterize the running time of a randomized algorithm. In this case, the sample space S is defined by all possible outcomes of the random sources used in the algorithm. We are usually most interested in the typical, average, or "expected" value of such a random variable. The ***expected value*** of a discrete random variable X is defined as

$$E(X) = \sum_x x \Pr(X = x),$$

where the summation is defined over the range of X.

Theorem 1.25 (The Linearity of Expectation): *Let X and Y be two arbitrary random variables. Then $E(X+Y) = E(X) + E(Y)$.*

Proof:

$$\begin{aligned}
E(X+Y) &= \sum_x \sum_y (x+y) \Pr(X = x \cap Y = y) \\
&= \sum_x \sum_y x \Pr(X = x \cap Y = y) + \sum_x \sum_y y \Pr(X = x \cap Y = y) \\
&= \sum_x \sum_y x \Pr(X = x \cap Y = y) + \sum_y \sum_x y \Pr(Y = y \cap X = x) \\
&= \sum_x x \Pr(X = x) + \sum_y y \Pr(Y = y) \\
&= E(X) + E(Y).
\end{aligned}$$

Note that this proof does not depend on any independence assumptions about the events when X and Y take on their respective values. ▩

Example 1.26: *Let X be a random variable that assigns the outcome of the roll of two fair dice to the sum of the number of dots showing. Then $E(X) = 7$.*

Proof: *To justify this claim, let X_1 and X_2 be random variables corresponding to the number of dots on each die, respectively. Thus, $X_1 = X_2$ (that is, they are two instances of the same function) and $E(X) = E(X_1 + X_2) = E(X_1) + E(X_2)$. Each outcome of the roll of a fair die occurs with probability $1/6$. Thus*

$$E(X_i) = \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6} = \frac{7}{2},$$

for $i = 1, 2$. Therefore, $E(X) = 7$. ■

Two random variables X and Y are ***independent*** if

$$\Pr(X = x | Y = y) = \Pr(X = x),$$

for all real numbers x and y.

Theorem 1.27: *If two random variables X and Y are independent, then*

$$E(XY) = E(X)E(Y).$$

Example 1.28: *Let X be a random variable that assigns the outcome of a roll of two fair dice to the product of the number of dots showing. Then $E(X) = 49/4$.*

Proof: *Let X_1 and X_2 be random variables denoting the number of dots on each die. The variables X_1 and X_2 are clearly independent; hence*

$$E(X) = E(X_1 X_2) = E(X_1)E(X_2) = (7/2)^2 = 49/4.$$

■

Chernoff Bounds

It is often necessary in the analysis of randomized algorithms to bound the sum of a set of random variables. One set of inequalities that makes this tractable is the set of Chernoff Bounds. Let $X_1, X_2, \ldots, X_n$ be a set of mutually independent indicator random variables, such that each X_i is 1 with some probability $p_i > 0$ and 0 otherwise. Let $X = \sum_{i=1}^{n} X_i$ be the sum of these random variables, and let μ denote the mean of X, that is, $\mu = E(X) = \sum_{i=1}^{n} p_i$. We give the following without proof.

Theorem 1.29: *Let X be as above. Then, for $\delta > 0$,*

$$\Pr(X > (1+\delta)\mu) < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu},$$

and, for $0 < \delta \le 1$,

$$\Pr(X < (1-\delta)\mu) < e^{-\mu\delta^2/2}.$$

1.4 Case Studies in Algorithm Analysis

Having presented the general framework for describing and analyzing algorithms, we now consider some case studies in algorithm analysis. Specifically, we show how to use the big-Oh notation to analyze two algorithms that solve the same problem but have different running times.

The problem we focus on in this section 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 will typically want 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. The prefix average is also useful as a "smoothing" function for a parameter that is quickly changing, as illustrated in Figure 1.13.

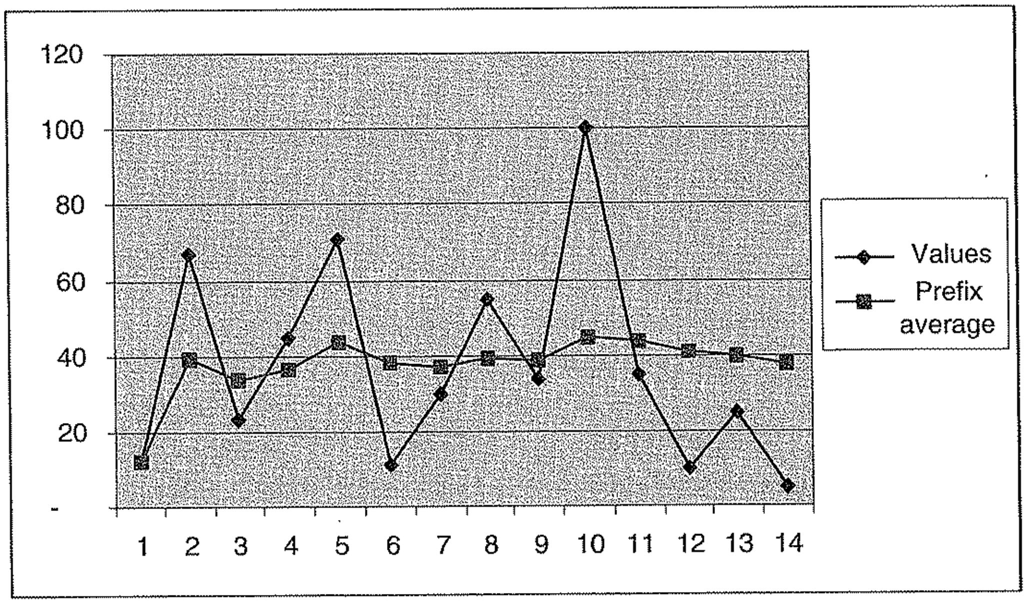


Figure 1.13: An illustration of the prefix average function and how it is useful for smoothing a quickly changing sequence of values.

1.4.1 A Quadratic-Time Prefix Averages Algorithm

Our first algorithm for the prefix averages problem, called prefixAverages1, is shown in Algorithm 1.14. 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 $X[0],\dots,X[i]$.
  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 $A$
```

Algorithm 1.14: 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, which are controlled by counters i and j, respectively. The body of the outer loop, controlled by counter i, is executed n times, for $i = 0,\dots,n-1$. 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 Theorem 1.13, 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 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 Theorem 1.7, the running time of prefixAverages1 is $O(n^2)$.

1.4.2 A Linear-Time Prefix Averages 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 present the details in Algorithm 1.15 (prefixAverages2).

```
**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 $A$
```

Algorithm 1.15: 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 Theorem 1.7, the running time of prefixAverages2 is $O(n)$, which is much better than the quadratic-time algorithm prefixAverages1.