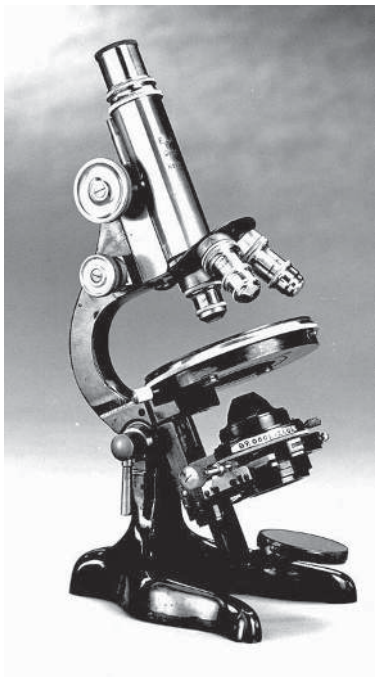
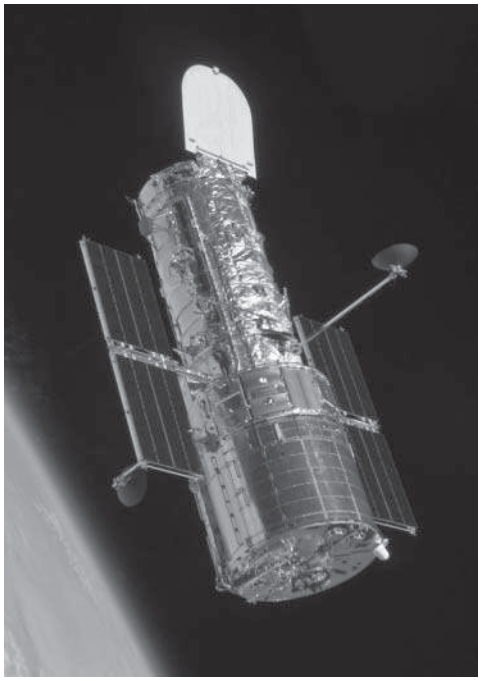
Chapter 1 Algorithm Analysis





Microscope: U.S. government image, from the N.I.H. Medical Instrument Gallery, DeWitt Stetten, Jr., Museum of Medical Research. Hubble Space Telescope: U.S. government image, from NASA, STS-125 Crew, May 25, 2009.

Contents

Scientists often have to deal with differences in scale, from the microscopically small to the astronomically large, and they have developed a wide range of tools for dealing with the differences in scale in the objects they study. Similarly, computer scientists must also deal with scale, but they deal with it primarily in terms of data volume rather than physical object size. In the world of information technology, ***scalability*** refers to the ability of a system to gracefully accommodate growing sizes of inputs or amounts of workload. Being able to achieve scalability for a computer system can mean the difference between a technological solution that can succeed in the marketplace or scientific application and one that becomes effectively unusable as data volumes increase. In this book, we are therefore interested in the design of scalable 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, and this book is dedicated to the discussion of paradigms and principles for the design and implementation of correct and efficient data structures and algorithms. But to be able to determine the degree to which algorithms and data structures are scalable, we must have precise ways of analyzing them.

The primary analysis tool we use in this book is to characterize the ***running time*** of an algorithm or data structure operation, with ***space usage*** also being of interest. Running time is a natural measure for the purposes of scalability, since time is a precious resource. It is an important consideration in economic and scientific applications, since everyone expects computer applications to run as fast as possible.

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 1.1.5, 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 1.2, 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 a case study on algorithm analysis in Section 1.3, focusing on a problem often used as a test question during job interviews. We follow this case study in Section 1.4 by presenting an interesting analysis technique, known as amortization, which allows us to account for the group behavior of many individual operations. Finally, we conclude the chapter with some exercises that include several problems inspired by questions commonly asked during job interviews at major software and Internet companies.

1.1 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.

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.

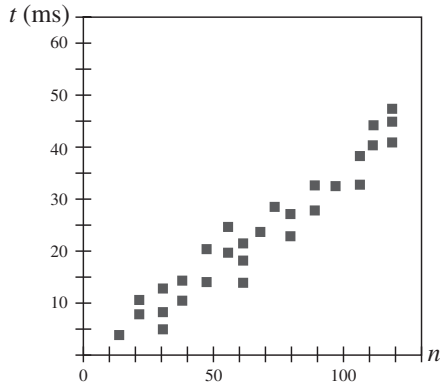


Figure 1.1: Results of an experimental study on the running time of an algorithm. A dot with coordinates (n, t) indicates that on an input of size n, the running time of the algorithm is t milliseconds (ms).

Requirements for a General Analysis Methodology

Experimental studies on running times are useful, 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 ***pseudocode***.

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

Using Pseudo-Code to Prove Algorithm Correctness

By inspecting the pseudocode, 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 pseudocode 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 ***pseudocode*** language, however, because of its reliance on natural language. At the same time, to help achieve clarity, pseudocode mixes natural language with standard programming language constructs. The programming language constructs we choose are those consistent with modern high-level languages such as Python, 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, and we assume Boolean operators allow for short-circuit evaluation.
- ***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. We usually index the cells of an array A of size n from 1 to n, as in mathematics, but sometimes we instead such an array from 0 to $n - 1$, 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 pseudocode, 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 pseudocode.

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 pseudocode. 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 pseudocode. 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 pseudocode 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 1 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), $A[i]$ 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 uphill.

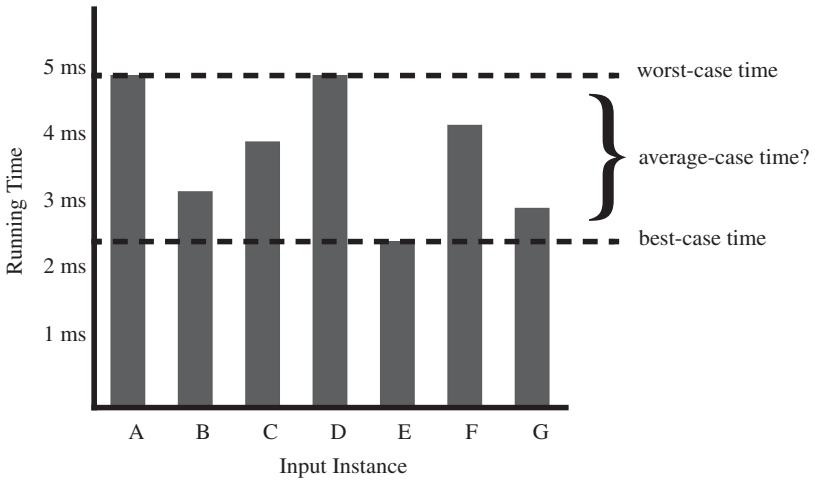


Figure 1.3: The difference between best-case and worst-case time. Each bar represents the running time of some algorithm on a different possible input.

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 $T(n) = 7(n-1) + 3 = 7n - 4$. 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 8, when we study some sorting and selection algorithms. We study methods for solving recurrence equations of a general form in Section 11.1.

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.

1.1.5 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 pseudocode description and each statement in a high-level language implementation tends to correspond 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 pseudocode 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.

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 pronounced as "$f(n)$ is ***big-Oh*** of $g(n)$" or "$f(n)$ is ***order*** $g(n)$." (See Figure 1.5.)

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

Proof: *We need 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 $n_0 = 1$, but there are other possibilities as well.* ■

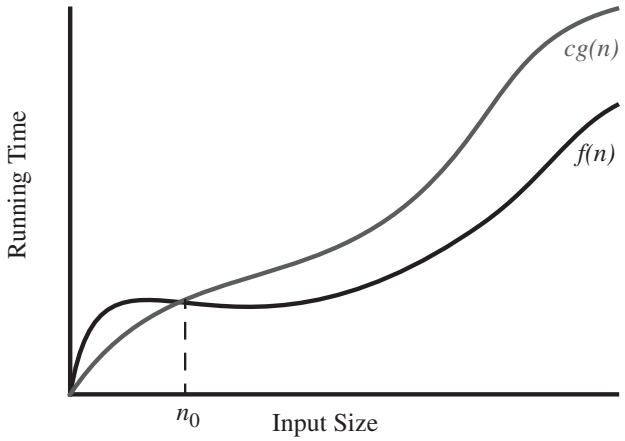


Figure 1.5: The function $f(n)$ is $O(g(n))$, for $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

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 of algorithm in terms of a parameter, n, which represents 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* $n = 1$, *but* $\log \log n < \log n$, *for* $n \geq 2$. *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: $5n \log n + 2n$ *is* $O(n \log n)$.

Proof: $5n \log n + 2n \leq 7n \log n$, *for* $n \geq 2$ *(but not for* $n = 1$*).* ■

As mentioned above, we are typically interested in characterizing the running time or space usage of algorithm in terms of a function, $f(n)$, which we bound using the big-Oh notion. For this reason, we should use the big-Oh notation to characterize such a function, $f(n)$, using an asymptotically small and simple function, $g(n)$. For instance, while it is true that a function, $f(n) = 4n^3 + 3n^{4/3}$, is $O(n^5)$, it is more informative to say that such an $f(n)$ is $O(n^3)$. Moreover, it is

often difficult or cumbersome to characterize the running time or space usage of algorithm exactly, whereas characterizing such measures using the big-Oh notion is typically easier.

Instead of always applying the big-Oh definition directly to obtain a big-Oh characterization, we can often use the following rules to simplify our task of figuring out the simplest characterization.

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

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 $2n \log n + O(n)$.

Big-Omega and Big-Theta

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.

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.

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 $n_0 = (12 + 6)/c = 18/c$, then $18 \leq cn$, for $n \geq n_0$. Thus, if $n \geq n_0$,*

$$f(n) = 12n^2 + 6n \leq 12n^2 + 6n^2 = 18n^2 \leq cn^3.$$

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 $n \geq n_0$, $12n \geq c$. Thus, if $n \geq n_0$,

$$f(n) = 12n^2 + 6n \geq 12n^2 \geq cn.$$

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.

1.1.6 The Importance of Asymptotic Notation

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 functions by asymptotic growth rate, as we show in Table 1.9.

Some Functions Ordered by Growth Rate	Common Name
$\log n$	logarithmic
$\log^2 n$	polylogarithmic
$\sqrt{n}$	square root
n	linear
$n \log n$	linearithmic
n^2	quadratic
n^3	cubic
2^n	exponential

Table 1.9: An ordered list of simple functions such that if a function $f(n)$ precedes a function $g(n)$ in the list, then $f(n)$ is $o(g(n))$. Using common terminology, the function, $\log^c n$, for any $c > 0$, is also polylogarithmic, and the functions, n^2 and n^3, are also polynomial.

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

n	$\log n$	$\log^2 n$	$\sqrt{n}$	$n \log n$	n^2	n^3	2^n
4	2	4	2	8	16	64	16
16	4	16	4	64	256	4,096	65,536
64	6	36	8	384	4,096	262,144	1.84×10^{19}
256	8	64	16	2,048	65,536	16,777,216	1.15×10^{77}
1,024	10	100	32	10,240	1,048,576	1.07×10^9	1.79×10^{308}
4,096	12	144	64	49,152	16,777,216	6.87×10^{10}	10^{1233}
16,384	14	196	128	229,376	268,435,456	4.4×10^{12}	10^{4932}
65,536	16	256	256	1,048,576	4.29×10^9	2.81×10^{14}	10^{19728}
262,144	18	324	512	4,718,592	6.87×10^{10}	1.8×10^{16}	10^{78913}

Table 1.10: Growth rates of several functions. Note the point at which the function $\sqrt{n}$ dominates $\log^2 n$.

1.2 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.

1.2.1 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 Carl 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. ■

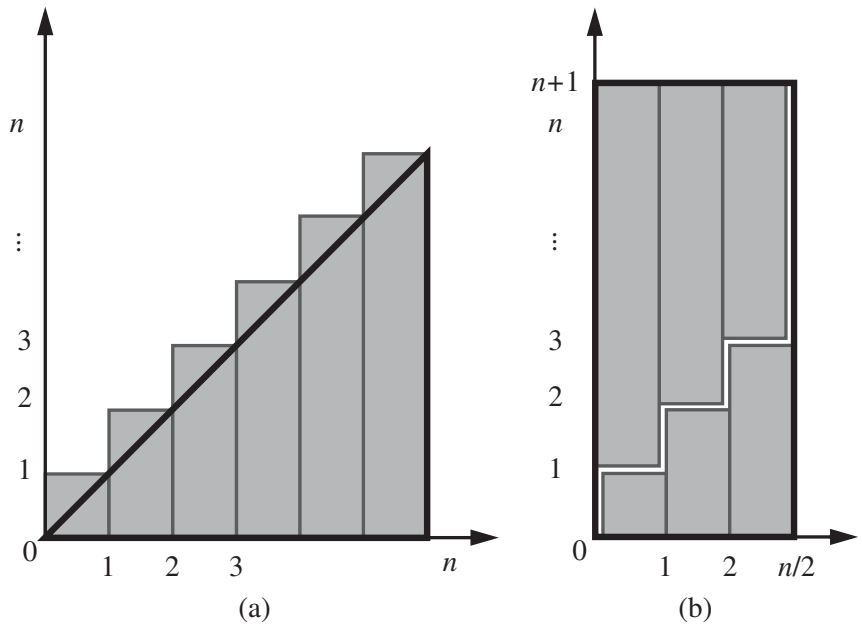


Figure 1.11: Visual justifications of Theorem 1.13. Both illustrations visualize the identity in terms of the total area covered by n unit-width rectangles with heights $1, 2, \ldots, n$. In (a) the rectangles are shown to cover a big triangle of area $n^2/2$ (base n and height n) plus n small triangles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

1.2.2 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]

1.2.3 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 counterexample. 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 1.1.5, for which the use of the ceiling function would usually be redundant anyway. (See Exercise R-1.25.)

Contrapositives and Contradiction

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***. 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 and b is odd.*

Proof: *To justify this claim, let us prove the contrapositive, "If a is even or b is even, then ab is even." So, suppose $a = 2i$ or $b = 2i$, for some integer i. Then we have $ab = (2i)b = 2ib$, or we have $ab = a(2i) = 2ai$; hence, in either case, ab is even. Since this establishes the contrapositive, it proves the original statement.* ■

Besides showing a use of the contrapositive proof technique, the above 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, "p or q," is "not p and not q," and that the negation of a statement, "p and q," is "not p or not q."

Another 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 even, then a is even or b is even.*

Proof: *Let ab be even. We wish to show that a is even or b is even. So, with the hope of leading to a contradiction, let us assume the opposite, namely, suppose a is odd and b is odd. Then $a = 2i + 1$ and $b = 2j + 1$, for some integers i and j. Hence, $ab = (2i + 1)(2j + 1) = 4ij + 2i + 2j + 1 = 2(2ij + i + j) + 1$; that is, ab is odd. But this is a contradiction: ab cannot simultaneously be odd and even. Therefore, a is even 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,*

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

■

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

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

■

It is useful to think about 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 proofs.

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 ← 0
while i < n do
    if x = A[i] then
        return i
    else
        i ← i + 1
return −1
```

Algorithm 1.12: Algorithm arrayFind.

1.2.4 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. $0 \leq \Pr(A) \leq 1$, 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 prove the following later in this book (Section 19.5).

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 $0 < \delta \leq 1$,

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

1.3 A Case Study in Algorithm Analysis

Having presented the general framework for describing and analyzing algorithms, we now present a case study in algorithm analysis to make this discussion more concrete. Specifically, we show how to use the big-Oh notation to analyze three algorithms that solve the same problem but have different running times.

The problem we focus on is one that is reportedly often used as a job interview question by major software and Internet companies—the ***maximum subarray problem***. Here, we are given an array of integers and asked to find the subarray whose elements have the largest sum. See the example of Figure 1.13. That is, given array $A = [a_1, a_2, \ldots, a_n]$, find indices j and k that maximize the sum

$$s_{j,k} = a_j + a_{j+1} + \cdots + a_k = \sum_{i=j}^{k} a_i.$$

Note that each element of the array could have a positive, negative, or zero value. Thus, in the special case where all array elements are negative, the solution is an empty subarray of conventional zero sum.

To define the problem more formally, we conventionally define the special array element $A[0] = 0$ and let $A[j : k]$ denote the sequence of elements of A from index j to index k ($0 \leq j \leq k \leq n$). The maximum subarray problem consists of finding the sequence $A[j : k]$ ($0 \leq j \leq k \leq n$) that maximizes $s_{j,k}$, the sum of its values. Such a maximum sum is referred to as the ***maximum subarray sum*** of array A.

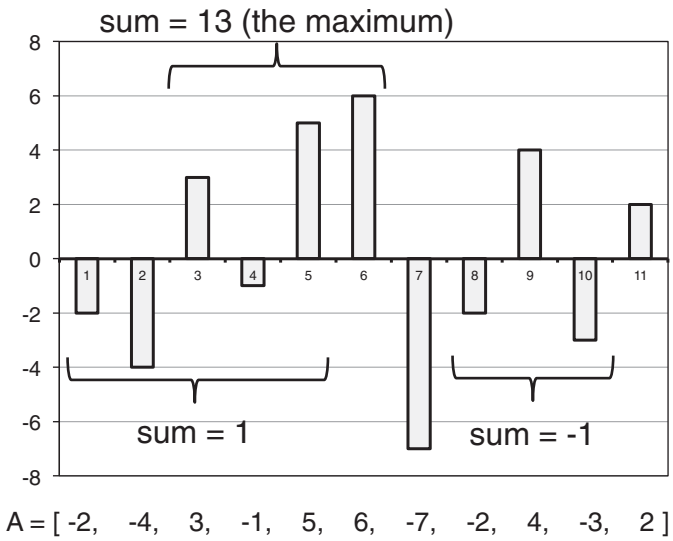


Figure 1.13: An instance of the maximum subarray problem. In this case, the maximum subarray is $A[3 : 6]$, that is, the maximum sum is $s_{3,6} = 13$.

In addition to being a good problem for testing the thinking skills of prospective employees, the maximum subarray problem also has applications in pattern analysis in digitized images.

1.3.1 A First Solution to the Maximum Subarray Problem

Our first algorithm for the maximum subarray problem, which we call MaxsubSlow, is shown in Algorithm 1.14. It computes the maximum of every possible subarray summation, $s_{j,k}$, of A separately.

Algorithm MaxsubSlow(A):
Input: An n-element array A of numbers, indexed from 1 to n.
Output: The maximum subarray sum of array A.

```
m ← 0    // the maximum found so far
for j ← 1 to n do
    for k ← j to n do
        s ← 0    // the next partial sum we are computing
        for i ← j to k do
            s ← s + A[i]
        if s > m then
            m ← s
return m
```

Algorithm 1.14: Algorithm MaxsubSlow.

It isn't hard to see that the MaxsubSlow algorithm is correct. This algorithm calculates the partial sum, $s_{j,k}$, of every possible subarray, by adding up the values in the subarray $A[j : k]$. Moreover, for every such subarray sum, it compares that sum to a running maximum and if the new value is greater than the old, it updates that maximum to the new value. In the end, this will be maximum subarray sum.

Incidentally, both the calculating of subarray summations and the computing of the maximum so far are examples of an ***accumulator*** pattern, where we incrementally accumulate values into a single variable to compute a sum or maximum (or minimum). This is a pattern that is used in a lot of algorithms, but in this case it is not being used in the most efficient way possible.

Analyzing the running time of the MaxsubSlow algorithm is easy. In particular, the outer loop, for index j, will iterate n times, its inner loop, for index k, will iterate at most n times, and the inner-most loop, for index i, will iterate at most n times. Thus, the running time of the MaxsubSlow algorithm is $O(n^3)$. Unfortunately, in spite of its use of the accumulator design pattern, giving the MaxsubSlow algorithm as a solution to the maximum subarray problem would be a bad idea during a job interview. This is a slow algorithm for the maximum subarray problem.

1.3.2 An Improved Maximum Subarray Algorithm

We can design an improved algorithm for the maximum subarray problem by observing that we are wasting a lot of time by recomputing all the subarray summations from scratch in the inner loop of the MaxsubSlow algorithm. There is a much more efficient way to calculate these summations. The crucial insight is to consider all the ***prefix sums***, which are the sums of the first t integers in A for $t = 1, 2, \ldots, n$. That is, consider each prefix sum, S_t, which is defined as

$$S_t = a_1 + a_2 + \cdots + a_t = \sum_{i=1}^{t} a_i.$$

If we are given all such prefix sums, then we can compute any subarray summation, $s_{j,k}$, in constant time using the formula

$$s_{j,k} = S_k - S_{j-1},$$

where we use the notational convention that $S_0 = 0$. To see this, note that

$$\begin{aligned} S_k - S_{j-1} &= \sum_{i=1}^{k} a_i - \sum_{i=1}^{j-1} a_i \\ &= \sum_{i=j}^{k} a_i = s_{j,k}, \end{aligned}$$

where we use the notational convention that $\sum_{i=1}^{0} a_i = 0$. We can incorporate the above observations into an improved algorithm for the maximum subarray problem, called MaxsubFaster, which we show in Algorithm 1.15.

Algorithm MaxsubFaster(A):
Input: An n-element array A of numbers, indexed from 1 to n.
Output: The maximum subarray sum of array A.

```
S_0 ← 0    // the initial prefix sum
for i ← 1 to n do
    S_i ← S_{i−1} + A[i]
m ← 0    // the maximum found so far
for j ← 1 to n do
    for k ← j to n do
        s = S_k − S_{j−1}
        if s > m then
            m ← s
return m
```

Algorithm 1.15: Algorithm MaxsubFaster.

Analyzing the MaxsubFaster Algorithm

The correctness of the MaxsubFaster algorithm follows along the same arguments as for the MaxsubSlow algorithm, but it is much faster. In particular, the outer loop, for index j, will iterate n times, its inner loop, for index k, will iterate at most n times, and the steps inside that loop will only take $O(1)$ time in each iteration. Thus, the total running time of the MaxsubFaster algorithm is $O(n^2)$, which improves the running time of the MaxsubSlow algorithm by a linear factor.

True story: A former student of one of the authors gave this very algorithm during a job interview for a major software company, when asked about the maximum subarray problem, correctly observing that this algorithm beats the running time of the naive $O(n^3)$-time algorithm by a linear factor. Sadly, this student did not get a job offer, however, and one reason could have been because there is an even better solution to the maximum subarray problem, which the student didn't give.

1.3.3 A Linear-Time Maximum Subarray Algorithm

We can improve the running time for solving the maximum subarray further by applying the intuition behind the prefix sums idea to the computation of the maximum itself. Recall our notation of letting $s_{j,k}$ denote the partial sum of the values in $A[j : k]$. Instead of computing prefix sum $S_t = s_{1,t}$, what if, we compute a ***maximum suffix sum***, M_t, which is the maximum of $s_{j,t}$ for $j = 1, \cdots, t$?

Such a definition is an interesting idea, but it is not quite right, because it doesn't include the boundary case where we wouldn't want any subarray that ends at t, in the event that all such subarrays sum up to a negative number. Instead, let us define

$$M_t = \max\{0, \max_{j=1,\cdots,t}\{s_{j,t}\}\}.$$

In other words, M_t is the maximum of 0 and the maximum $s_{j,k}$ value where we restrict k to equal t. For example, in the array shown in Figure 1.13, we have $M_2 = 0$, $M_3 = 3$, and $M_4 = 2$.

This definition implies that if $M_t > 0$, then it is the summation value for a maximum subarray that ends at t, and if $M_t = 0$, then we can safely ignore any subarray that ends at t.

Note that if we know all the M_t values, for $t = 1, 2, \ldots, n$, then the solution to the maximum subarray problem would simply be the maximum of all these values. So let us consider how we could compute these M_t values.

The crucial observation is that, for $t \geq 2$, if we have a maximum subarray that ends at t, and it has a positive sum, then it is either $A[t : t]$ or it is made up of the maximum subarray that ends at $t - 1$ plus $A[t]$. If this were not the case, then we could make a subarray of even larger sum by swapping out the one we chose to end at $t - 1$ with the maximum one that ends at $t - 1$, which would contradict the fact

that we have the maximum subarray that ends at t. In addition, if taking the value of maximum subarray that ends at $t-1$ and adding $A[t]$ makes this sum no longer be positive, then $M_t = 0$, for there is no subarray that ends at t with a positive summation. In other words, we can define $M_0 = 0$ as a boundary condition, and use the following formula to compute M_t, for $t = 1, 2, \ldots, n$:

$$M_t = \max\{0,\ M_{t-1} + A[t]\}.$$

Therefore, we can solve the maximum subarray problem using the algorithm, MaxsubFastest, shown in Algorithm 1.16.

Algorithm MaxsubFastest(A):
Input: An n-element array A of numbers, indexed from 1 to n.
Output: The maximum subarray sum of array A.

```
M_0 ← 0    // the initial prefix maximum
for t ← 1 to n do
    M_t ← max{0, M_{t-1} + A[t]}
m ← 0    // the maximum found so far
for t ← 1 to n do
    m ← max{m, M_t}
return m
```

Algorithm 1.16: Algorithm MaxsubFastest.

Analyzing the MaxsubFastest Algorithm

The MaxsubFastest algorithm consists of two loops, which each iterate exactly n times and take $O(1)$ time in each iteration. Thus, the total running time of the MaxsubFastest algorithm is $O(n)$. Incidentally, in addition to using the accumulator pattern, to calculate the M_t and m variables based on previous values of these variables, it also can be viewed as a simple application of the ***dynamic programming*** technique, which we discuss in Chapter 12.

Given all these positive aspects of this algorithm, even though we can't guarantee that a prospective employee will get a job offer by describing the MaxsubFastest algorithm when asked about the maximum subarray problem, we can at least guarantee that this is the way to nail this question. Still, we are nonetheless leaving one small detail as an exercise (C-1.1), which is to modify the description of the MaxsubFastest algorithm so that, in addition to the value of the maximum subarray summation, it also outputs the indices j and k that identify the maximum subarray $A[j : k]$.

1.4 Amortization

An important analysis tool useful for understanding the running times of algorithms that have steps with widely varying performance is ***amortization***. The term "amortization" itself comes from the field of accounting, which provides an intuitive monetary metaphor for algorithm analysis, as we shall see in this section.

The typical data structure usually supports a wide variety of different methods for accessing and updating the elements it stores. Likewise, some algorithms operate iteratively, with each iteration performing a varying amount of work. In some cases, we can effectively analyze the performance of these data structures and algorithms on the basis of the worst-case running time of each individual operation. Amortization takes a different viewpoint. Rather than focusing on each operation separately, it considers the interactions between all the operations by studying the running time of a series of these operations.

The Clearable Table Data Structure

As an example of amortized analysis, let us introduce a simple data structure, the ***clearable table***. This structure stores a table of elements, which can be accessed by their indices in the table. In addition, the clearable table supports the following two methods:

add(e): Add the element e to the next available cell in the table.

clear(): Empty the table by removing all its elements.

Let S be a clearable table with n elements implemented by means of an array, with a fixed upper bound, N, on its size. Operation clear takes $\Theta(n)$ time, since we should dereference all the elements in the table in order to really empty it.

Now consider a series of n operations on an initially empty clearable table S. If we take a worst-case viewpoint, we may say that the running time of this series of operations is $O(n^2)$, since the worst case of a single clear operation in the series is $O(n)$, and there may be as many as $O(n)$ clear operations in this series. While this analysis is correct, it is also an overstatement, since an analysis that takes into account the interactions between the operations shows that the running time of the entire series is actually $O(n)$.

Theorem 1.30: *A series of n operations on an initially empty clearable table implemented with an array takes $O(n)$ time.*

Proof: Let $M_0, \ldots, M_{n-1}$ be the series of operations performed on S, and let $M_{i_0}, \ldots, M_{i_{k-1}}$ be the k clear operations within the series. We have

$$0 \leq i_0 < \ldots < i_{k-1} \leq n-1.$$

Let us also define $i_{-1} = -1$. The running time of operation M_{i_j} (a clear operation) is $O(i_j - i_{j-1})$, because at most $i_j - i_{j-1} - 1$ elements could have been added into the table (using the add operation) since the previous clear operation $M_{i_{j-1}}$ or since the beginning of the series. Thus, the running time for the clear operations is

$$O\left(\sum_{j=0}^{k-1}(i_j - i_{j-1})\right).$$

A summation such as this is known as a ***telescoping sum***, for all terms other than the first and last cancel each other out. That is, this summation is $O(i_{k-1} - i_{-1})$, which is $O(n)$. All the remaining operations of the series take $O(1)$ time each. Thus, we conclude that a series of n operations performed on an initially empty clearable table takes $O(n)$ time. ■

Theorem 1.30 indicates that the average running time of any operation on a clearable table is $O(1)$, where the average is taken over an arbitrary series of operations, starting with an initially empty clearable table.

Amortizing an Algorithm's Running Time

The above example provides a motivation for the amortization technique, which gives us a worst-case way of performing an average-case analysis. Formally, we define the ***amortized running time*** of an operation within a series of operations as the worst-case running time of the series of operations divided by the number of operations. When the series of operations is not specified, it is usually assumed to be a series of operations from the repertoire of a certain data structure, starting from an empty structure. Thus, by Theorem 1.30, we can say that the amortized running time of each operation for a clearable table structure is $O(1)$ when we implement that clearable table with an array. Note that the actual running time of an operation may be much higher than its amortized running time (for example, a particular clear operation may take $O(n)$ time).

The advantage of using amortization is that it gives us a way to do a robust average-case analysis without using any probability. It simply requires that we have some way of characterizing the worst-case running time for performing a series of operations. We can even extend the notion of amortized running time so as to assign each individual operation in a series of operations its own amortized running time, provided the total actual time taken to process the entire series of operations is no more than the sum of amortized bounds given to the individual operations.

There are several ways of doing an amortized analysis. The most obvious way is to use a direct argument to derive bounds on the total time needed to perform a series of operations, which is what we did in the proof of Theorem 1.30. While direct arguments can often be found for a simple series of operations, performing an amortized analysis of a nontrivial series of operations is often easier using special techniques for amortized analysis.

1.4.1 Amortization Techniques

There are two fundamental techniques for performing an amortized analysis, one based on a financial model—the accounting method—and the other based on an energy model—the potential function method.

The Accounting Method

The ***accounting method*** for performing an amortized analysis is to use a scheme of credits and debits for keeping track of the running time of the different operations in the series. The basis of the accounting method is simple. We view the computer as a coin-operated appliance that requires the payment of 1 ***cyber-dollar*** for a constant amount of computing time. We also view an operation as a sequence of constant time ***primitive operations***, which each cost 1 ***cyber-dollar*** to be executed. When an operation is executed, we should have enough cyber-dollars available to pay for its running time. Of course, the most obvious approach is to charge an operation a number of cyber-dollars equal to the number of primitive operations performed. However, the interesting aspect of using the accounting method is that we do not have to be fair in the way we charge the operations. Namely, we can overcharge some operations that execute few primitive operations and use the profit made on them to help out other operations that execute many primitive operations. This mechanism may allow us to charge the same amount a of cyber-dollars to each operation in the series, without ever running out of cyber-dollars to pay for the computer time. Hence, if we can set up such a scheme, called an ***amortization scheme***, we can say that each operation in the series has an amortized running time that is $O(a)$. When designing an amortization scheme, it is often convenient to think of the unspent cyber-dollars as being "stored" in certain places of the data structure, for example, at the elements of a table.

An alternative amortization scheme charges different amounts to the various operations. In this case, the amortized running time of an operation is proportional to the total charges made divided by the number of operations.

We now go back to the clearable table example and present an amortization scheme for it that yields an alternative proof of Theorem 1.30. Let us assume that one cyber-dollar is enough to pay for the execution of operation of an index access or an add operation, and for the time spent by operation clear to dereference one element. We shall charge each operation 2 cyber-dollars. This means undercharging operation clear and overcharging all the other operations by 1 cyber-dollar. The cyber-dollar profited in an add operation will be stored at the element inserted by the operation. (See Figure 1.17.) When a clear operation is executed, the cyber-dollar stored at each element in the table is used to pay for the time spent dereferencing it. Hence, we have a valid amortization scheme, where each operation is charged 2 cyber-dollars, and all the computing time is paid for. This simple amortization scheme implies the result of Theorem 1.30.

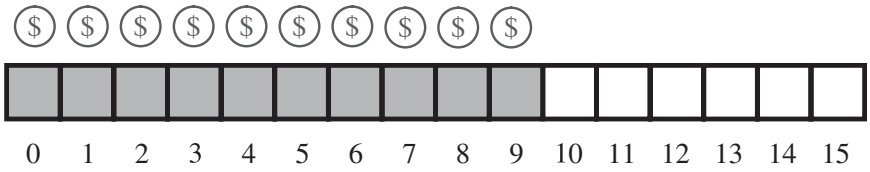


Figure 1.17: Cyber-dollars stored at the elements of a clearable table S in the amortized analysis of a series of operations on S.

Notice that the worst case for the running time occurs for a series of add operations followed by a single clear operation. In other cases, at the end of the series of operations, we may end up with some unspent cyber-dollars, which are those profited from index access operations and those stored at the elements still in the sequence. Indeed, the computing time for executing a series of n operations can be paid for with the amount of cyber-dollars between n and $2n$. Our amortization scheme accounts for the worst case by always charging 2 cyber-dollars per operation.

At this point, we should stress that the accounting method is simply an analysis tool. It does not require that we modify a data structure or the execution of an algorithm in any way. In particular, it does not require that we add objects for keeping track of the cyber-dollars spent.

Potential Functions

Another useful technique for performing an amortized analysis is based on an energy model. In this approach, we associate with our structure a value, Φ, which represents the current energy state of our system. Each operation that we perform will contribute some additional amount, known as the amortized time, to Φ, but then also extracts value from Φ in proportion to the amount of time actually spent. Formally, we let $\Phi_0 \geq 0$ denote the initial value of Φ, before we perform any operations, and we use Φ_i to denote the value of the potential function, Φ, after we perform the ith operation. The main idea of using the potential function argument is to use the change in potential for the ith operation, $\Phi_i - \Phi_{i-1}$, to characterize the amortized time needed for that operation.

Let us focus more closely on the action of the ith operation, letting t_i denote its actual running time. We define the amortized running time of the ith operation as

$$t'_i = t_i + \Phi_i - \Phi_{i-1}.$$

That is, the amortized cost of the ith operation is the actual running time plus the net change in potential that operation causes (which may be positive or negative). Or, put another way,

$$t_i = t'_i + \Phi_{i-1} - \Phi_i,$$

that is, the actual time spent is the amortized cost plus the net drop in potential.

Denote by T' the total amortized time for performing n operations on our structure. That is,

$$T' = \sum_{i=1}^{n} t'_i.$$

Then the total actual time, T, taken by our n operations can be bounded as

$$\begin{aligned} T &= \sum_{i=1}^{n} t_i \\ &= \sum_{i=1}^{n} \left(t'_i + \Phi_{i-1} - \Phi_i\right) \\ &= \sum_{i=1}^{n} t'_i + \sum_{i=1}^{n} (\Phi_{i-1} - \Phi_i) \\ &= T' + \sum_{i=1}^{n} (\Phi_{i-1} - \Phi_i) \\ &= T' + \Phi_0 - \Phi_n, \end{aligned}$$

since the second term above forms a telescoping sum. In other words, the total actual time spent is equal to the total amortized time plus the net drop in potential over the entire sequence of operations. Thus, so long as $\Phi_n \geq \Phi_0$, then $T \leq T'$, the actual time spent is no more than the amortized time.

To make this concept more concrete, let us repeat our analysis of the clearable table using a potential argument. In this case, we choose the potential Φ of our system to be the actual number of elements in our clearable table. We claim that the amortized time for any operation is 2, that is, $t'_i = 2$, for $i = 1, \ldots, n$. To justify this, let us consider the two possible methods for the ith operation.

- add(e): inserting the element e into the table increases Φ by 1 and the actual time needed is 1 unit of time. So, in this case,
$$1 = t_i = t'_i + \Phi_{i-1} - \Phi_i = 2 - 1,$$
which is clearly true.
- clear(): removing all m elements from the table requires no more than $m+2$ units of time—m units to do the removal plus at most 2 units for the method call and its overhead. But this operation also drops the potential Φ of our system from m to 0 (we even allow for $m = 0$). So, in this case
$$m + 2 = t_i = t'_i + \Phi_{i-1} - \Phi_i = 2 + m,$$
which clearly holds.

Therefore, the amortized time to perform any operation on a clearable table is $O(1)$. Moreover, since $\Phi_i \geq \Phi_0$, for any $i \geq 1$, the actual time, T, to perform n operations on an initially empty clearable table is $O(n)$.

1.4.2 Analyzing an Extendable Array Implementation

A major weakness of the simple array implementation for a clearable table given above is that it requires advance specification of a fixed capacity, N, for the total number of elements that may be stored in the table. If the actual number of elements, n, of the table is much smaller than N, then this implementation will waste space. Worse, if n increases past N, then this implementation will crash.

Let us provide a means to grow the array A that stores the elements of a table S. Of course, in any conventional programming language, such as C, C++, and Java, we cannot actually grow the array A; its capacity is fixed at some number N. Instead, when an ***overflow*** occurs, that is, when $n = N$ and method add is called, we perform the following steps:

1. Allocate a new array B of capacity $2N$.
2. Copy $A[i]$ to $B[i]$, for $i = 0, \ldots, N-1$.
3. Let $A = B$, that is, we use B as the array supporting S.

This array replacement strategy is known as an ***extendable array***. (See Figure 1.18.) Intuitively, this strategy is much like that of the hermit crab, which moves into a larger shell when it outgrows its previous one.

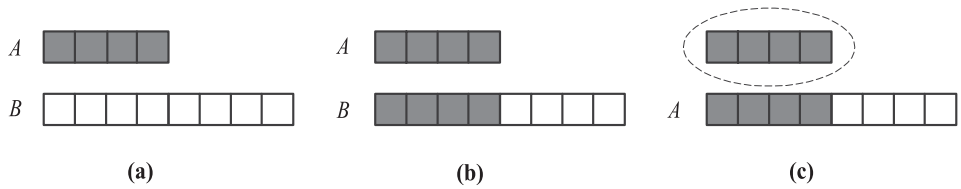


Figure 1.18: An illustration of the three steps for "growing" an extendable array: (a) create new array B; (b) copy elements from A to B; (c) reassign reference A to the new array. Not shown is the future garbage collection of the old array.

In terms of efficiency, this array replacement strategy might at first seem slow, for performing a single array replacement of size n required by some element insertion takes $\Theta(n)$ time. Still, notice that after we perform an array replacement, our new array allows us to add n new elements to the table before the array must be replaced again. This simple fact allows us to show that the running time of a series of operations performed on an initially empty extendable table is actually quite efficient. As a shorthand notation, let us refer to the insertion of an element to be the last element in a vector as an "add" operation. Using ***amortization***, we can show that performing a sequence of such add operations on a table implemented with an extendable array is actually quite efficient.

Theorem 1.31: *Let S be a table implemented by means of an extendable array A, as described above. The total time to perform a series of n add operations in S, starting from S being empty and A having size $N = 1$, is $O(n)$.*

Proof: We justify this theorem using the accounting method for ***amortization***. To perform this analysis, we again view the computer as a coin-operated appliance that requires the payment of 1 ***cyber-dollar*** for a constant amount of computing time. When an operation is executed, we should have enough cyber-dollars available in our current "bank account" to pay for that operation's running time. Thus, the total amount of cyber-dollars spent for any computation will be proportional to the total time spent on that computation. The beauty of this analysis is that we can over charge some operations to save up cyber-dollars to pay for others.

Let us assume that 1 cyber-dollar is enough to pay for the execution of each add operation in S, excluding the time for growing the array. Also, let us assume that growing the array from size k to size $2k$ requires k cyber-dollars for the time spent copying the elements. We shall charge each add operation 3 cyber-dollars. Thus, we over charge each add operation not causing an overflow by 2 cyber-dollars. Think of the 2 cyber-dollars profited in an insertion that does not grow the array as being "stored" at the element inserted. An overflow occurs when the table S has 2^i elements, for some integer $i \geq 0$, and the size of the array used by S is 2^i. Thus, doubling the size of the array will require 2^i cyber-dollars. Fortunately, these cyber-dollars can be found at the elements stored in cells 2^{i-1} through $2^i - 1$. (See Figure 1.19.) Note that the previous overflow occurred when the number of elements became larger than 2^{i-1} for the first time, and thus the cyber-dollars stored in cells 2^{i-1} through $2^i - 1$ were not previously spent. Therefore, we have a valid amortization scheme in which each operation is charged 3 cyber-dollars and all the computing time is paid for. That is, we can pay for the execution of n add operations using $3n$ cyber-dollars. ■

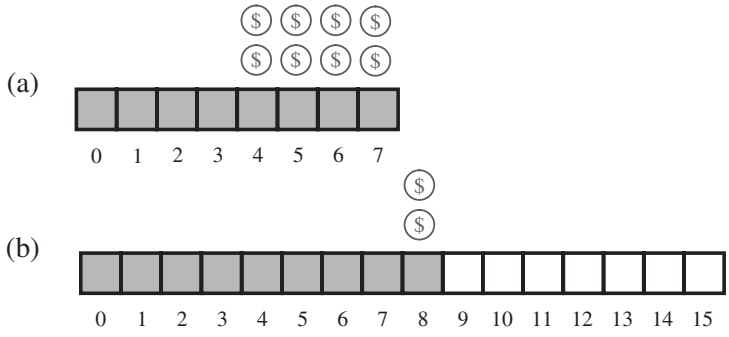


Figure 1.19: A costly add operation: (a) a full 8-cell with 2 cyber-dollars for cells 4 through 7; (b) an add doubles the capacity. Copying elements spends the cyber-dollars in the table, inserting the new element spends 1 cyber-dollar charged to the add, and 2 cyber-dollars "of profit" are stored at cell 8.

A table can be doubled in size with each extension, as we have described it, or we can specify an explicit capacityIncrement parameter that determines the fixed amount an array should grow with each expansion. That is, this parameter is set to a value, k, then the array adds k new cells when it grows. We must utilize such a parameter with caution, however. For most applications, doubling in size is the right choice, as the following theorem shows.

Theorem 1.32: *If we create an initially empty table with a fixed positive* capacityIncrement *value, then performing a series of n* add *operations on this vector takes $\Omega(n^2)$ time.*

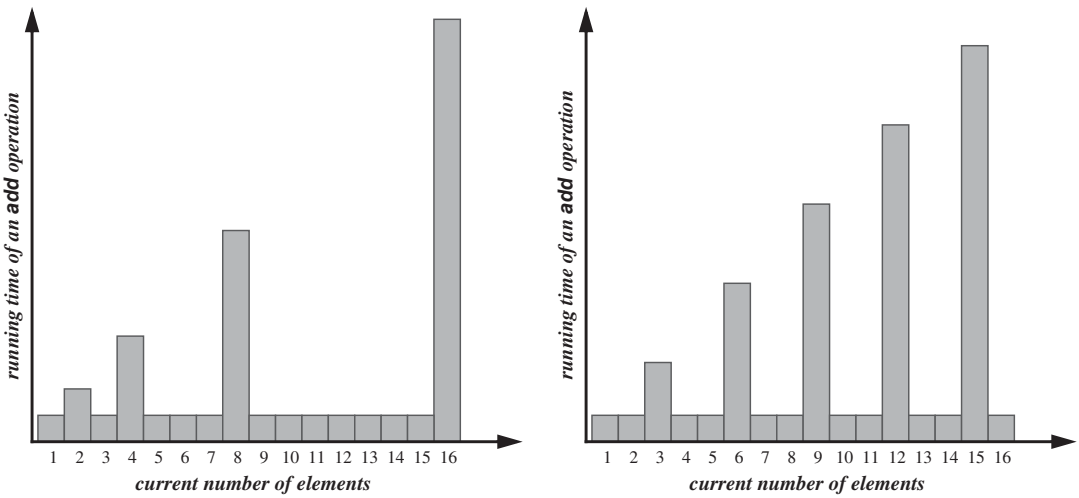
Proof: Let $c > 0$ be the capacityIncrement value, and let $c_0 > 0$ denote the initial size of the array. An overflow will be caused by an add operation when the current number of elements in the table is $c_0 + ic$, for $i = 0, \ldots, m-1$, where $m = \lfloor (n - c_0)/c \rfloor$. Hence, by Theorem 1.13, the total time for handling the overflows is proportional to

$$\sum_{i=0}^{m-1}(c_0 + ci) \;=\; c_0 m + c\sum_{i=0}^{m-1} i \;=\; c_0 m + c\frac{m(m-1)}{2},$$

which is $\Omega(n^2)$. Thus, performing the n add operations takes $\Omega(n^2)$ time. ■

Figure 1.20 compares the running times of a series of add operations on an initially empty table, for two initial values of capacityIncrement.

We discuss applications of amortization further when we discuss splay trees (Section 4.5) and a tree structure for set partitions (Section 7.1).



(a) (b)

Figure 1.20: Running times of a series of add operations on an extendable table. In (a) the size is doubled with each expansion, and in (b) it is simply incremented by capacityIncrement $= 3$.

1.5 Exercises

Reinforcement

R-1.1 Graph the functions $12n$, $6n \log n$, n^2, n^3, and 2^n using a logarithmic scale for the x- and y-axes; that is, if the function value $f(n)$ is y, plot this as a point with x-coordinate at $\log n$ and y-coordinate at $\log y$.

R-1.2 Show that the MaxsubSlow algorithm runs in $\Omega(n^3)$ time.

R-1.3 Algorithm A uses $10n \log n$ operations, while algorithm B uses n^2 operations. Determine the value n_0 such that A is better than B for $n \geq n_0$.

R-1.4 Repeat the previous problem assuming B uses $n\sqrt{n}$ operations.

R-1.5 Show that $\log^3 n$ is $o(n^{1/3})$.

R-1.6 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-1.7 Order the following list of functions by the big-Oh notation. Group together (for example, by underlining) those functions that are big-Theta of one another.

$6n \log n$	2^{100}	$\log \log n$	$\log^2 n$	$2^{\log n}$
2^{2^n}	$\lceil \sqrt{n} \rceil$	$n^{0.01}$	$1/n$	$4n^{3/2}$
$3n^{0.5}$	$5n$	$\lfloor 2n \log^2 n \rfloor$	2^n	$n \log_4 n$
4^n	n^3	$n^2 \log n$	$4^{\log n}$	$\sqrt{\log n}$

Hint: When in doubt about two functions $f(n)$ and $g(n)$, consider $\log f(n)$ and $\log g(n)$ or $2^{f(n)}$ and $2^{g(n)}$.

R-1.8 For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t assuming that the algorithm to solve the problem takes $f(n)$ microseconds. Recall that $\log n$ denotes the logarithm in base 2 of n. Some entries have already been completed to get you started.

	1 Second	1 Hour	1 Month	1 Century
$\log n$	$\approx 10^{300000}$			
$\sqrt{n}$				
n				
$n \log n$				
n^2				
n^3				
2^n				
$n!$		12		

R-1.9 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 Algorithm 1.12, on each one, 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? Is this a linear-time algorithm? Why or why not?

R-1.10 Consider the following recurrence equation, defining $T(n)$, as

$$T(n) = \begin{cases} 4 & \text{if } n = 1 \\ T(n-1) + 4 & \text{otherwise.} \end{cases}$$

Show, by induction, that $T(n) = 4n$.

R-1.11 Give a big-Oh characterization, in terms of n, of the running time of the Loop1 method shown in Algorithm 1.21.

R-1.12 Perform a similar analysis for method Loop2 shown in Algorithm 1.21.

R-1.13 Perform a similar analysis for method Loop3 shown in Algorithm 1.21.

R-1.14 Perform a similar analysis for method Loop4 shown in Algorithm 1.21.

R-1.15 Perform a similar analysis for method Loop5 shown in Algorithm 1.21.

Algorithm Loop1(n):
  $s \leftarrow 0$
  for $i \leftarrow 1$ **to** n **do**
    $s \leftarrow s + i$

Algorithm Loop2(n):
  $p \leftarrow 1$
  for $i \leftarrow 1$ **to** $2n$ **do**
    $p \leftarrow p \cdot i$

Algorithm Loop3(n):
  $p \leftarrow 1$
  for $i \leftarrow 1$ **to** n^2 **do**
    $p \leftarrow p \cdot i$

Algorithm Loop4(n):
  $s \leftarrow 0$
  for $i \leftarrow 1$ **to** $2n$ **do**
    for $j \leftarrow 1$ **to** i **do**
      $s \leftarrow s + i$

Algorithm Loop5(n):
  $s \leftarrow 0$
  for $i \leftarrow 1$ **to** n^2 **do**
    for $j \leftarrow 1$ **to** i **do**
      $s \leftarrow s + i$

Algorithm 1.21: A collection of loop methods.

R-1.16 Show that if $f(n)$ is $O(g(n))$ and $d(n)$ is $O(h(n))$, then the summation $f(n) + d(n)$ is $O(g(n) + h(n))$.

R-1.17 Show that $O(\max\{f(n), g(n)\}) = O(f(n) + g(n))$.

R-1.18 Show that $f(n)$ is $O(g(n))$ if and only if $g(n)$ is $\Omega(f(n))$.

R-1.19 Show that if $p(n)$ is a polynomial in n, then $\log p(n)$ is $O(\log n)$.

R-1.20 Show that $(n + 1)^5$ is $O(n^5)$.

R-1.21 Show that 2^{n+1} is $O(2^n)$.

R-1.22 Show that n is $o(n \log n)$.

R-1.23 Show that n^2 is $\omega(n)$.

R-1.24 Show that $n^3 \log n$ is $\Omega(n^3)$.

R-1.25 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-1.26 Justify the fact 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-1.27 Given the values of the maximum suffix sums, M_t ($t = 1, \cdots, 11$), for the array $A = [-2, -4, 3, -1, 5, 6, -7, -2, 4, -3, 2]$.

R-1.28 What is the amortized running time of an operation in a series of n add operations on an initially empty extendable table implemented with an array such that the capacityIncrement parameter is always maintained to be $\lceil \log(m+1) \rceil$, where m is the number of elements of the stack? That is, each time the table is expanded by $\lceil \log(m + 1) \rceil$ cells, its capacityIncrement is reset to $\lceil \log(m' + 1) \rceil$ cells, where m is the old size of the table and m' is the new size (in terms of actual elements present).

R-1.29 Describe a recursive algorithm for finding both the minimum and the maximum elements in an array A of n elements. Your method should return a pair (a, b), where a is the minimum element and b is the maximum. What is the running time of your method?

R-1.30 Suppose you have an array of n numbers and you select each one independently with probability $1/n^{1/2}$. Use the Chernoff bound to determine an upper bound on the probability that you would have more than $4n^{1/2}$ elements in this random sample.

R-1.31 Rewrite the proof of Theorem 1.31 under the assumption that the cost of growing the array from size k to size $2k$ is $3k$ cyber-dollars. How much should each add operation be charged to make the amortization work?

R-1.32 Suppose we have a set of n balls and we choose each one independently with probability $1/n^{1/2}$ to go into a basket. Derive an upper bound on the probability that there are more than $3n^{1/2}$ balls in the basket.

Creativity

C-1.1 Describe how to modify the description of the MaxsubFastest algorithm so that, in addition to the value of the maximum subarray summation, it also outputs the indices j and k that identify the maximum subarray $A[j : k]$.

C-1.2 Describe how to modify the MaxsubFastest algorithm so that it uses just a single loop and, instead of computing $n + 1$ different M_t values, it maintains just a single variable M.

C-1.3 What is the amortized running time of the operations in a sequence of n operations $P = p_1 p_2 \ldots p_n$ if the running time of p_i is $\Theta(i)$ if i is a multiple of 3, and is constant otherwise?

C-1.4 What is the total running time of counting from 1 to n in binary if the time needed to add 1 to the current number i is proportional to the number of bits in the binary expansion of i that must change in going from i to $i + 1$?

C-1.5 Consider the following recurrence equation, defining a function $T(n)$:

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

Show, by induction, that $T(n) = n(n + 1)/2$.

C-1.6 Consider the following recurrence equation, defining a function $T(n)$:

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n-1) + 2^n & \text{otherwise,} \end{cases}$$

Show, by induction, that $T(n) = 2^{n+1} - 1$.

C-1.7 Consider the following recurrence equation, defining a function $T(n)$:

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2T(n-1) & \text{otherwise,} \end{cases}$$

Show, by induction, that $T(n) = 2^n$.

C-1.8 Al and Bill are arguing about the performance of their sorting algorithms. Al claims that his $O(n \log n)$-time algorithm is *always* faster than Bill's $O(n^2)$-time algorithm. To settle the issue, they implement and run the two algorithms on many randomly generated data sets. To Al's dismay, they find that if $n < 100$, the $O(n^2)$-time algorithm actually runs faster, and only when $n \geq 100$ is the $O(n \log n)$-time algorithm better. Explain why this scenario is possible. You may give numerical examples.

C-1.9 Give an example of a positive function $f(n)$ such that $f(n)$ is neither $O(n)$ nor $\Omega(n)$.

C-1.10 Show that $\sum_{i=1}^{n} i^2$ is $O(n^3)$.

C-1.11 Show that $\sum_{i=1}^{n} i/2^i < 2$.

Hint: Try to bound this sum term by term with a geometric progression.

C-1.12 Show that $\log_b f(n)$ is $\Theta(\log f(n))$ if $b > 1$ is a constant.

C-1.13 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-1.14 An n-degree ***polynomial*** $p(x)$ is an equation of the form

$$p(x) = \sum_{i=0}^{n} a_i x^i,$$

where x is a real number and each a_i is a constant.

a. Describe a simple $O(n^2)$-time method for computing $p(x)$ for a particular value of x.
b. Consider now 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 multiplications and additions this method of evaluation uses.

C-1.15 Consider the following induction "proof" 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 of the flock. The remaining $n - 1$ are all the same color by induction. Now put sheep a back in the flock, and take out a different sheep, b. By induction, the $n - 1$ sheep (now with a in their group) are all the same color. Therefore, a is the same color as all the other sheep; hence, all the sheep in the flock are the same color.

What is wrong with this "proof"?

C-1.16 Consider the following "proof" that the Fibonacci function, $F(n)$, defined as $F(1) = 1$, $F(2) = 2$, $F(n) = F(n-1) + F(n-2)$, is $O(n)$:
Base case ($n \leq 2$): $F(1) = 1$, which is $O(1)$, and $F(2) = 2$, which is $O(2)$.
Induction step ($n > 2$): Assume the claim is 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-1.16. Therefore, $F(n)$ is $O(n)$, since $O((n-1) + (n-2))$ is $O(n)$.
What is wrong with this "proof"?

C-1.17 Consider the Fibonacci function, $F(n)$, from the previous exercise. Show by induction that $F(n)$ is $\Omega((3/2)^n)$.

C-1.18 Draw a visual justification of Theorem 1.13 analogous to that of Figure 1.11b for the case when n is odd.

C-1.19 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 allowed to use only $O(1)$ additional space besides the array A itself.

C-1.20 Show that the summation $\sum_{i=1}^{n} \lceil \log_2 i \rceil$ is $O(n \log n)$.

C-1.21 Show that the summation $\sum_{i=1}^{n} \lceil \log_2 i \rceil$ is $\Omega(n \log n)$.

C-1.22 Show that the summation $\sum_{i=1}^{n} \lceil \log_2 (n/i) \rceil$ is $O(n)$. You may assume that n is a power of 2.

Hint: Use induction to reduce the problem to that for $n/2$.

C-1.23 Let S be a set of n lines 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-1.24 Suppose that each row of an $n \times n$ array A consists of 1's and 0's such that, in any row of A, all the 1's come before any 0's in that row. Assuming A is already in memory, describe a method running in $O(n)$ time (not $O(n^2)$ time) for finding the row of A that contains the most 1's.

C-1.25 Suppose that 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 in that row. Suppose further 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 $O(n^2)$ time) for counting the number of 1's in the array A.

C-1.26 Describe, using pseudocode, 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-1.27 Give a recursive algorithm to compute the product of two positive integers m and n using only addition.

C-1.28 Give complete pseudocode for a new class, ShrinkingTable, that performs the add method of the extendable table, as well as methods, remove(), which removes the last (actual) element of the table, and shrinkToFit(), which replaces the underlying array with an array whose capacity is exactly equal to the number of elements currently in the table.

C-1.29 Consider an extendable table that supports both add and remove methods, as defined in the previous exercise. Moreover, suppose we grow the underlying array implementing the table by doubling its capacity any time we need to increase the size of this array, and we shrink the underlying array by half any time the number of (actual) elements in the table dips below $N/4$, where N is the current capacity of the array. Show that a sequence of n add and remove methods, starting from an array with capacity $N = 1$, takes $O(n)$ time.

C-1.30 Consider an implementation of the extendable table, but instead of copying the elements of the table into an array of double the size (that is, from N to $2N$) when its capacity is reached, we copy the elements into an array with $\lceil \sqrt{N} \rceil$ additional cells, going from capacity N to $N + \lceil \sqrt{N} \rceil$. Show that performing a sequence of n add operations (that is, insertions at the end) runs in $\Theta(n^{3/2})$ time in this case.

Applications

A-1.1 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 every integer p such that $1 < p < r$, check whether p divides r.
> If so, print "The secret message is p!" and stop; if not, continue.

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 if 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, $n = (\log_2 r)/8$, and that each division takes time $O(n)$.

A-1.2 Program the three algorithms given in the chapter for the maximum subarray problem, from Section 1.3, and perform a careful experimental analysis of their running times. Plot their running times as a function of their input sizes as scatter plots on both a linear-linear scale and a log-log scale. Choose representative values of the size n, and run at least five tests for each size value n in your tests. Note that the slope of a line plotted on a log-log scale is based on the exponent of a function, since $\log n^c = c \log n$.

A-1.3 Implement an extendable table using arrays that can increase in size as elements are added. Perform an experimental analysis of each of the running times for performing a sequence of n add methods, assuming the array size is increased from N to the following possible values:

a. $2N$
b. $N + \lceil \sqrt{N} \rceil$
c. $N + \lceil \log N \rceil$
d. $N + 100$.

A-1.4 An evil king has a cellar containing n bottles of expensive wine, and his guards have just caught a spy trying to poison the king's wine. Fortunately, the guards caught the spy after he succeeded in poisoning only one bottle. Unfortunately, they don't know which one. To make matters worse, the poison the spy used was

very deadly; just one drop diluted even a billion to one will still kill someone. Even so, the poison works slowly; it takes a full month for the person to die. Design a scheme that allows the evil king to determine exactly which one of his wine bottles was poisoned in just one month's time while expending at most $O(\log n)$ of his taste testers.

Note: All the remaining problems are inspired by questions reported to have been asked in job interviews for major software and Internet companies.

A-1.5 Suppose you are given a set of small boxes, numbered 1 to n, identical in every respect except that each of the first i contain a pearl whereas the remaining $n - i$ are empty. You also have two magic wands that can each test whether a box is empty or not in a single touch, except that a wand disappears if you test it on an empty box. Show that, without knowing the value of i, you can use the two wands to determine all the boxes containing pearls using at most $o(n)$ wand touches. Express, as a function of n, the asymptotic number of wand touches needed.

A-1.6 Repeat the previous problem assuming that you now have k magic wands, with $k > 2$ and $k < \log n$. Express, as a function of n and k, the asymptotic number of wand touches needed to identify all the magic boxes containing pearls.

A-1.7 Suppose you are given an integer c and an array, A, indexed from 1 to n, of n integers in the range from 1 to $5n$ (possibly with duplicates). Describe an efficient algorithm for determining if there are two integers, $A[i]$ and $A[j]$, in A that sum to c, that is, such that $c = A[i] + A[j]$, for $1 \leq i < j \leq n$. What is the running time of your algorithm?

A-1.8 Given an array, A, describe an efficient algorithm for reversing A. For example, if $A = [3, 4, 1, 5]$, then its reversal is $A = [5, 1, 4, 3]$. You can only use $O(1)$ memory in addition to that used by A itself. What is the running time of your algorithm?

A-1.9 Given a string, S, of n digits in the range from 0 to 9, describe an efficient algorithm for converting S into the integer it represents. What is the running time of your algorithm?

A-1.10 Given an array, A, of n integers, find the longest subarray of A such that all the numbers in that subarray are in sorted order. What is the running time of your method?

A-1.11 Given an array, A, of n positive integers, each of which appears in A exactly twice, except for one integer, x, describe an $O(n)$-time method for finding x using only a single variable besides A.

A-1.12 Given an array, A, of $n - 2$ unique integers in the range from 1 to n, describe an $O(n)$-time method for finding the two integers in the range from 1 to n that are not in A. You may use only $O(1)$ space in addition to the space used by A.

A-1.13 Suppose you are writing a simulator for a single-elimination sports tournament (like in NCAA Division-1 basketball). There are n teams at the beginning of the tournament and in each round of the tournament teams are paired up and the games for each pair are simulated. Winners progress to the next round and losers are sent home. This continues until a grand champion team is the final winner.

Suppose your simulator takes $O(\log n)$ time to process each game. How much time does your simulator take in total?

A-1.14 Suppose you are given an array, A, of n positive integers. Describe an $O(n)$ algorithm for removing all the even numbers from A. That is, if A has m odd numbers, then, after you are done, these odd numbers should occupy the first m cells of A in the same relative order they were in originally.

A-1.15 Given an integer $k > 0$ and an array, A, of n bits, describe an efficient algorithm for finding the shortest subarray of A that contains k 1's. What is the running time of your method?

A-1.16 A certain town has exactly n married heterosexual couples. Every wife knows whether every other wife's husband is cheating on his wife or not, but no wife knows if her own husband is cheating or not. In fact, if a wife ever learns that her husband is cheating on her, then she will poison him that very night. So no husband will ever confess that he is cheating. One day, the mayor (who is not married) announces that there is at least one cheating husband in the town. What happens next?

A-1.17 Imagine that a magician has just given you a biased coin. It looks just like a normal coin, with a "heads" side and a "tails" side, but each time this coin is flipped, it is more likely to come up heads than tails. How can you use this coin to generate an unbiased sequence of independent random bits, that is, a random sequence of 0's and 1's where each bit has an independent equal probability of being a 0 or 1?

A-1.18 Suppose you are processing a stream of bytes, one at a time, but you don't know in advance how many there will be, as the last byte is a special EOF character. You only get to consider each byte once. Describe a scheme for choosing a byte in this stream at random so that every byte in the stream has an equal chance of being chosen. You may use only an $O(1)$ amount of space.

Chapter Notes

The big-Oh notation has prompted several discussions over its proper use [36, 95, 128]. Knuth [129, 128], for example, defines it using the notation $f(n) = O(g(n))$, but refers to this "equality" as being 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 [36]. The linear-time algorithm we gave for the maximum subarray problem is due to Kadane [26]. For more information on amortization, please see Tarjan [207, 208]. We include a number of useful mathematical facts in Appendix A. The reader interested in further study into the analysis of algorithms is referred to the books by Graham, Knuth, and Patashnik [93], and Sedgewick and Flajolet [190]. Finally, for more information about using experimentation to estimate the running time of algorithms, we refer the interested reader to papers by McGeoch and coauthors [151, 152, 153].