CHAPTER
1

Algorithm Basics

Before you jump into the study of algorithms, you need a little background. To begin with, you need to know that, simply stated, an *algorithm* is a recipe for getting something done. *It defines the steps for performing a task in a certain way.*

That definition seems simple enough, but no one writes algorithms for performing extremely simple tasks. No one writes instructions for how to access the fourth element in an array. It is just assumed that this is part of the definition of an array and that you know how to do it (if you know how to use the programming language in question).

Normally people write algorithms only for difficult tasks. Algorithms explain how to find the solution to a complicated algebra problem, how to find the shortest path through a network containing thousands of streets, or how to find the best mix of hundreds of investments to optimize profits.

This chapter explains some of the basic algorithmic concepts you should understand if you want to get the most out of your study of algorithms.

It may be tempting to skip this chapter and jump to studying specific algorithms, but you should at least skim this material. Pay close attention to the section "Big O Notation," because a good understanding of runtime performance can mean the difference between an algorithm performing its task in seconds, hours, or not at all.

Approach

To get the most out of an algorithm, you must be able to do more than simply follow its steps. You need to understand the following:

- **The algorithm's behavior.** Does it find the best possible solution, or does it just find a good solution? Could there be multiple best solutions? Is there a reason to pick one "best" solution over the others?
- **The algorithm's speed.** Is it fast? Slow? Is it usually fast but sometimes slow for certain inputs?
- **The algorithm's memory requirements.** How much memory will the algorithm need? Is this a reasonable amount? Does the algorithm require billions of terabytes more memory than a computer could possibly have (at least today)?
- **The main techniques the algorithm uses.** Can you reuse those techniques to solve similar problems?

This book covers all these topics. It does not, however, attempt to cover every detail of every algorithm with mathematical precision. It uses an intuitive approach to explain algorithms and their performance, but it does not analyze performance in rigorous detail. Although that kind of proof can be interesting, it can also be confusing and take up a lot of space, providing a level of detail that is unnecessary for most programmers. This book, after all, is intended primarily for programming professionals who need to get a job done.

This book's chapters group algorithms that have related themes. Sometimes the theme is the task they perform (sorting, searching, network algorithms), sometimes it's the data structures they use (linked lists, arrays, hash tables, trees), and sometimes it's the techniques they use (recursion, decision trees, distributed algorithms). At a high level, these groupings may seem arbitrary, but when you read about the algorithms, you'll see that they fit together.

In addition to those categories, many algorithms have underlying themes that cross chapter boundaries. For example, tree algorithms (Chapters 10, 11, and 12) tend to be highly recursive (Chapter 9). Linked lists (Chapter 3) can be used to build arrays (Chapter 4), hash tables (Chapter 8), stacks (Chapter 5), and queues (Chapter 5). The ideas of references and pointers are used to build linked lists (Chapter 3), trees (Chapters 10, 11, and 12), and networks (Chapters 13 and 14). As you read, watch for these common threads. Appendix A summarizes common strategies programs use to make these ideas easier to follow.

Algorithms and Data Structures

An *algorithm* is a recipe for performing a certain task. A *data structure* is a way of arranging data to make solving a particular problem easier. A data structure could be a way of arranging values in an array, a linked list that connects items in a certain pattern, a tree, a graph, a network, or something even more exotic.

Often algorithms are closely tied to data structures. For example, the edit distance algorithm described in Chapter 15 uses a network to determine how similar two strings are. The algorithm is tied closely to the network and won't work without it.

Often an algorithm says, "Build a certain data structure and then use it in a certain way." The algorithm can't exist without the data structure, and there's no point in building the data structure if you don't plan to use it with the algorithm.

Pseudocode

To make the algorithms described in this book as useful as possible, they are first described in intuitive English terms. From this high-level explanation, you should be able to implement the algorithm in most programming languages.

Often, however, an algorithm's implementation contains niggling little details that can make implementation hard. To make handling those details easier, the algorithms are also described in pseudocode. *Pseudocode* is text that is a lot like a programming language but that is not really a programming language. The idea is to give you the structure and details you would need to implement the algorithm in code without tying the algorithm to a particular programming language. Hopefully you can translate the pseudocode into actual code to run on your computer.

The following snippet shows an example of pseudocode for an algorithm that calculates the greatest common divisor (GCD) of two integers:

```
// Find the greatest common divisor of a and b.
// GCD(a, b) = GCD(b, a Mod b).
Integer: Gcd(Integer: a, Integer: b)
    While (b != 0)
        // Calculate the remainder.
        Integer: remainder = a Mod b

        // Calculate GCD(b, remainder).
        a = b
```

```
        b = remainder
    End While

    // GCD(a, 0) is a.
    Return a
End Gcd
```

THE MOD OPERATOR

The modulus operator, which is written `Mod` in the pseudocode, means the remainder after division. For example, 13 `Mod` 4 is 1 because 13 divided by 4 is 3 with a remainder of 1.

The equation 13 `Mod` 4 is usually pronounced "13 mod 4" or "13 modulo 4."

The pseudocode starts with a comment. Comments begin with the characters `//` and extend to the end of the line.

The first actual line of code is the algorithm's declaration. This algorithm is called `Gcd` and returns an integer result. It takes two parameters named `a` and `b`, both of which are integers.

NOTE Chunks of code that perform a task, optionally returning a result, are variously called routines, subroutines, methods, procedures, subprocedures, or functions.

The code after the declaration is indented to show that it is part of the method. The first line in the method's body begins a `While` loop. The code indented below the `While` statement is executed as long as the condition in the `While` statement remains true.

The `While` loop ends with an `End While` statement. This statement isn't strictly necessary, because the indentation shows where the loop ends, but it provides a reminder of what kind of block of statements is ending.

The method exits at the `Return` statement. This algorithm returns a value, so this `Return` statement indicates which value the algorithm should return. If the algorithm doesn't return any value, such as if its purpose is to arrange values or build a data structure, the `Return` statement isn't followed by a return value.

The code in this example is fairly close to actual programming code. Other examples may contain instructions or values described in English. In those cases, the instructions are enclosed in angle brackets (`<>`) to indicate that you need to translate the English instructions into program code.

Normally when a parameter or variable is declared (in the `Gcd` algorithm, this includes the parameters `a` and `b` and the variable `remainder`), its data type is given before it, followed by a colon, as in `Integer: remainder`. The data type may be omitted for simple integer looping variables, as in `For i = 1 To 10`.

One other feature that is different from some programming languages is that a pseudocode `For` loop may include a `Step` statement indicating the value by which the looping variable is changed each trip through the loop. A `For` loop ends with a `Next i` statement (where `i` is the looping variable) to remind you which loop is ending.

For example, consider the following pseudocode:

```
For i = 100 To 0 Step -5
    // Do something...
Next i
```

This code is equivalent to the following C# code:

```
for (int i = 100; i >= 0; i -= 5)
{
    // Do something...
}
```

The pseudocode used in this book uses `If-Then-Else` statements, `Case` statements, and other statements as needed. These should be familiar to you from your knowledge of real programming languages. Anything else that the code needs is spelled out in English.

One basic data structure that may be unfamiliar to you depending on which programming languages you know is a `List`. A `List` is similar to a self-expanding array. It provides an `Add` method that lets you add an item to the end of the list. For example, the following pseudocode creates a `List Of Integer` that contains the numbers 1 through 10:

```
List Of Integer: numbers
For i = 1 To 10
    numbers.Add(i)
Next i
```

After a list is initialized, the pseudocode can use it as if it were a normal array and access items anywhere in the list. Unlike arrays, lists also let you add and remove items from any position.

Many algorithms in this book are written as methods or functions that return a result. The method's declaration begins with the result's data type. If a method performs some task and doesn't return a result, it has no data type.

The following pseudocode contains two methods:

```
// Return twice the input value.
Integer: DoubleIt(Integer: value)
    Return 2 * value
End DoubleIt

// The following method does something and doesn't return a value.
DoSomething(Integer: values[])
    // Some code here.
    ...
End DoSomething
```

The `DoubleIt` method takes an integer as a parameter and returns an integer. The code doubles the input value and returns the result.

The `DoSomething` method takes as a parameter an array of integers named values. It performs a task and doesn't return a result. For example, it might randomize or sort the items in the array. (Note that this book assumes that arrays start with the index 0. For example, an array containing three items has indices 0, 1, and 2.)

Pseudocode should be intuitive and easy to understand, but if you find something that doesn't make sense to you, feel free to post a question on the book's discussion forum at `www.wiley.com/go/essentialalgorithms` or e-mail me at `RodStephens@CSharpHelper.com`. I'll point you in the right direction.

One problem with pseudocode is that it has no compiler to detect errors. As a check of the basic algorithm, and to give you some actual code to use for a reference, C# implementations of most of the algorithms and many of the exercises are available for download on the book's website.

Algorithm Features

A good algorithm must have three features: correctness, maintainability, and efficiency.

Obviously if an algorithm doesn't solve the problem for which it was designed, it's not much use. If it doesn't produce correct answers, there's little point in using it.

NOTE Interestingly, some algorithms produce correct answers only some of the time but are still useful. For example, an algorithm may be able to give you some information with a certain probability. In that case you may be able to rerun the algorithm many times to increase your confidence that the answer is correct. Fermat's primality test, described in Chapter 2, is this kind of algorithm.

If an algorithm isn't maintainable, it's dangerous to use in a program. If an algorithm is simple, intuitive, and elegant, you can be confident that it is producing correct results, and you can fix it if it doesn't. If the algorithm is intricate, confusing, and convoluted, you may have a lot of trouble implementing it, and you will have even more trouble fixing it if a bug arises. If it's hard to understand, how can you know if it is producing correct results?

NOTE This doesn't mean it isn't worth studying confusing and difficult algorithms. Even if you have trouble implementing an algorithm, you may learn a lot in the attempt. Over time your algorithmic intuition and skill will increase, so algorithms you once thought were confusing will seem easier to handle. You must always test all algorithms thoroughly, however, to make sure they are producing correct results.

Most developers spend a lot of effort on efficiency, and efficiency is certainly important. If an algorithm produces a correct result and is simple to implement and debug, it's still not much use if it takes seven years to finish or if it requires more memory than a computer can possibly hold.

In order to study an algorithm's performance, computer scientists ask how its performance changes as the size of the problem changes. If you double the number of values the algorithm is processing, does the runtime double? Does it increase by a factor of 4? Does it increase exponentially so that it suddenly takes years to finish?

You can ask the same questions about memory usage or any other resource that the algorithm requires. If you double the size of the problem, does the amount of memory required double?

You can also ask the same questions with respect to the algorithm's performance under different circumstances. What is the algorithm's worst-case performance? How likely is the worst case to occur? If you run the algorithm on a large set of random data, what is its average-case performance?

To get a feeling for how problem size relates to performance, computer scientists use Big O notation, described in the following section.

Big O Notation

Big O notation uses a function to describe how the algorithm's worst-case performance relates to the problem size as the size grows very large. (This is sometimes called the program's *asymptotic performance*.) The function is written within parentheses after a capital letter O.

For example, $O(N^2)$ means an algorithm's runtime (or memory usage or whatever you're measuring) increases as the square of the number of inputs N. If you double the number of inputs, the runtime increases by roughly a factor of 4. Similarly, if you triple the number of inputs, the runtime increases by a factor of 9.

NOTE Often $O(N^2)$ is pronounced "order N squared." For example, you might say, "The quicksort algorithm described in Chapter 6 has a worst-case performance of order N squared."

There are five basic rules for calculating an algorithm's Big O notation:

1. If an algorithm performs a certain sequence of steps f(N) times for a mathematical function f, it takes O(f(N)) steps.
2. If an algorithm performs an operation that takes O(f(N)) steps and then performs a second operation that takes O(g(N)) steps for functions f and g, the algorithm's total performance is O(f(N) + g(N)).
3. If an algorithm takes O(f(N) + g(N)) and the function f(N) is greater than g(N) for large N, the algorithm's performance can be simplified to O(f(N)).

4. If an algorithm performs an operation that takes O(f(N)) steps, and for every step in that operation it performs another O(g(N)) steps, the algorithm's total performance is O(f(N) × g(N)).
5. Ignore constant multiples. If C is a constant, O(C × f(N)) is the same as O(f(N)), and O(f(C × N)) is the same as O(f(N)).

These rules may seem a bit formal, with all the f(N) and g(N), but they're fairly easy to apply. If they seem confusing, a few examples should make them easier to understand.

Rule 1

If an algorithm performs a certain sequence of steps f(N) times for a mathematical function f, it takes O(f(N)) steps.

Consider the following algorithm, written in pseudocode, for finding the largest integer in an array:

```
Integer: FindLargest(Integer: array[])
    Integer: largest = array[0]
    For i = 1 To <largest index>
        If (array[i] > largest) Then largest = array[i]
    Next i
    Return largest
End FindLargest
```

The `FindLargest` algorithm takes as a parameter an array of integers and returns an integer result. It starts by setting the variable `largest` equal to the first value in the array.

It then loops through the remaining values in the array, comparing each to `largest`. If it finds a value that is larger than `largest`, the program sets `largest` equal to that value.

After it finishes the loop, the algorithm returns `largest`.

This algorithm examines each of the N items in the array once, so it has O(N) performance.

NOTE Often algorithms spend most of their time in loops. There's no way an algorithm can execute more than N steps with a fixed number of code lines unless it contains some sort of loop.

Study an algorithm's loops to figure out how much time it takes.

Rule 2

If an algorithm performs an operation that takes O(f(N)) steps and then performs a second operation that takes O(g(N)) steps for functions f and g, the algorithm's total performance is O(f(N) + g(N)).

If you look again at the `FindLargest` algorithm shown in the preceding section, you'll see that a few steps are not actually inside the loop. The following

pseudocode shows the same steps, with their runtime order shown to the right in comments:

```
Integer: FindLargest(Integer: array[])
    Integer: largest = array[0]                              // O(1)
    For i = 1 To <largest index>                             // O(N)
        If (array[i] > largest) Then largest = array[i]
    Next i
    Return largest                                           // O(1)
End FindLargest
```

This algorithm performs one setup step before it enters its loop and then performs one more step after it finishes the loop. Both of those steps have performance O(1) (they're each just a single step), so the total runtime for the algorithm is really O(1 + N + 1). You can use normal algebra to combine terms to rewrite this as O(2 + N).

Rule 3

If an algorithm takes O(f(N) + g(N)) and the function f(N) is greater than g(N) for large N, the algorithm's performance can be simplified to O(f(N)).

The preceding example showed that the `FindLargest` algorithm has runtime O(2 + N). When N grows large, the function N is larger than the constant value 2, so O(2 + N) simplifies to O(N).

Ignoring the smaller function lets you focus on the algorithm's asymptotic behavior as the problem size becomes very large. It also lets you ignore relatively small setup and cleanup tasks. If an algorithm spends some time building simple data structures and otherwise getting ready to perform a big computation, you can ignore the setup time as long as it's small compared to the length of the main calculation.

Rule 4

If an algorithm performs an operation that takes O(f(N)) steps, and for every step in that operation it performs another O(g(N)) steps, the algorithm's total performance is O(f(N) × g(N)).

Consider the following algorithm that determines whether an array contains any duplicate items. (Note that this isn't the most efficient way to detect duplicates.)

```
Boolean: ContainsDuplicates(Integer: array[])
    // Loop over all of the array's items.
    For i = 0 To <largest index>
        For j = 0 To <largest index>
            // See if these two items are duplicates.
            If (i != j) Then
                If (array[i] == array[j]) Then Return True
            End If
        Next j
```

```
    Next i

    // If we get to this point, there are no duplicates.
    Return False
End ContainsDuplicates
```

This algorithm contains two nested loops. The outer loop iterates over all the array's N items, so it takes O(N) steps.

For each trip through the outer loop, the inner loop also iterates over the N items in the array, so it also takes O(N) steps.

Because one loop is nested inside the other, the combined performance is $O(N \times N) = O(N^2)$.

Rule 5

Ignore constant multiples. If C is a constant, $O(C \times f(N))$ is the same as $O(f(N))$, and $O(f(C \times N))$ is the same as $O(f(N))$.

If you look again at the `ContainsDuplicates` algorithm shown in the preceding section, you'll see that the inner loop actually performs one or two steps. It performs an `If` test to see if the indices `i` and `j` are the same. If they are different, it compares `array[i]` and `array[j]`. It may also return the value `True`.

If you ignore the extra step for the `Return` statement (it happens at most only once), and you assume that the algorithm performs both the `If` statements (as it does most of the time), the inner loop takes $O(2 \times N)$ steps. Therefore, the algorithm's total performance is $O(N \times 2 \times N) = O(2 \times N^2)$.

Rule 5 lets you ignore the factor of 2, so the runtime is $O(N^2)$.

This rule really goes back to the purpose of Big O notation. The idea is to get a feeling for the algorithm's behavior as N increases. In this case, suppose you increase N by a factor of 2.

If you plug the value $2 \times N$ into the equation $2 \times N^2$, you get the following:

$$2 \times (2 \times N)^2 = 2 \times 4 \times N^2 = 8 \times N^2$$

This is 4 times the original value $2 \times N^2$, so the runtime has increased by a factor of 4.

Now try the same thing with the runtime simplified by Rule 5 to $O(N^2)$. Plugging $2 \times N$ into this equation gives the following:

$$(2 \times N)^2 = 4 \times N^2$$

This is 4 times the original value N^2, so this also means that the runtime has increased by a factor of 4.

Whether you use the formula $2 \times N^2$ or just N^2, the result is the same: Increasing the size of the problem by a factor of 2 increases the runtime by a factor of 4. The important thing here isn't the constant; it's the fact that the runtime increases as the square of the number of inputs N.

NOTE It's important to remember that Big O notation is just intended to give you an idea of an algorithm's theoretical behavior. Your results in practice may be different. For example, suppose an algorithm's performance is O(N), but if you don't ignore the constants, the actual number of steps executed is something like 100,000,000 + N. Unless N is really big, you may not be able to safely ignore the constant.

Common Runtime Functions

When you study the runtime of algorithms, some functions occur frequently. The following sections give some examples of a few of the most common functions. They also give you some perspective so that you'll know, for example, whether an algorithm with $O(N^3)$ performance is reasonable.

1

An algorithm with O(1) performance takes a constant amount of time no matter how big the problem is. These sorts of algorithms tend to perform relatively trivial tasks because they cannot even look at all the inputs in O(1) time.

For example, at one point the quicksort algorithm needs to pick a number that is in an array of values. Ideally, that number should be somewhere in the middle of all the values in the array, but there's no easy way to tell which number might fall nicely in the middle. (For example, if the numbers are evenly distributed between 1 and 100, 50 would make a good dividing number.) The following algorithm shows one common approach for solving this problem:

```
Integer: DividingPoint(Integer: array[])
    Integer: number1 = array[0]
    Integer: number2 = array[<last index of array>]
    Integer: number3 = array[<last index of array> / 2]

    If (<number1 is between number2 and number3>) Return number1
    If (<number2 is between number1 and number3>) Return number2
    Return number3
End MiddleValue
```

This algorithm picks the values at the beginning, end, and middle of the array, compares them, and returns whichever item lies between the other two. This may not be the best item to pick out of the whole array, but there's a decent chance that it's not too terrible a choice.

Because this algorithm performs only a few fixed steps, it has O(1) performance and its runtime is independent of the number of inputs N. (Of course, this algorithm doesn't really stand alone. It's just a small part of a more complicated algorithm.)

Log N

An algorithm with O(log N) performance typically divides the number of items it must consider by a fixed fraction at every step.

LOGARITHMS

The logarithm of a number in a certain log base is the power to which the base must be raised to get a certain result. For example, $\log_2(8)$ is 3 because $2^3 = 8$. Here, 2 is the log base.

Often in algorithms the base is 2 because the inputs are being divided into two groups repeatedly. As you'll see shortly, the log base isn't really important in Big O notation, so it is usually omitted.

For example, Figure 1-1 shows a sorted complete binary tree. It's a *binary tree* because every node has at most two branches. It's a *complete tree* because every level (except possibly the last) is completely full and all the nodes in the last level are grouped on the left side. It's a *sorted tree* because every node's value is at least as large as its left child and no larger than its right child.

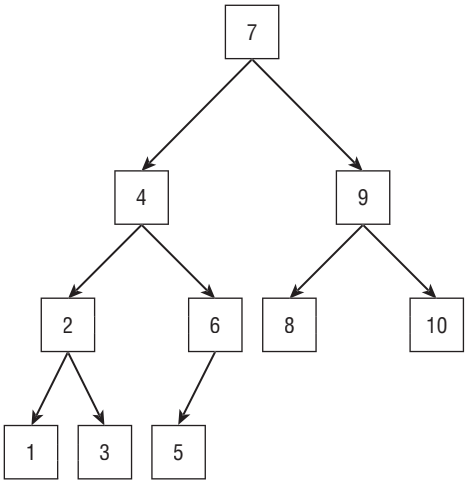

Figure 1-1: Searching a full binary tree takes O(log N) steps.

The following pseudocode shows one way you might search the tree shown in Figure 1-1 to find a particular item.

```
Node: FindItem(Integer: target_value)
    Node: test_node = <root of tree>

    Do Forever
```

```
        // If we fell off the tree. The value isn't present.
        If (test_node == null) Return null

        If (target_value == test_node.Value) Then
            // test_node holds the target value. This is the node we want.
            Return test_node
        Else If (target_value < test_node.Value) Then
            // Move to the left child.
            test_node = test_node.LeftChild
        Else
            // Move to the right child.
            test_node = test_node.RightChild
        End If
    End Do
End FindItem
```

Chapter 10 covers tree algorithms in detail, but you should be able to get the gist of the algorithm from the following discussion.

The algorithm declares and initializes the variable `test_node` so that it points to the root at the top of the tree. (Traditionally, trees in computer programs are drawn with the root at the top, unlike real trees.) It then enters an infinite loop.

If `test_node` is `null`, the target value isn't in the tree, so the algorithm returns `null`.

NOTE `null` is a special value that you can assign to a variable that should normally point to an object such as a node in a tree. The value `null` means "This variable doesn't point to anything."

If `test_node` holds the target value, `test_node` is the node we're seeking, so the algorithm returns it.

If `target_value`, the value we're searching for, is less than the value in `test_node`, the algorithm sets `test_node` equal to its left child. (If `test_node` is at the bottom of the tree, its `LeftChild` value is `null`, and the algorithm handles the situation the next time it goes through the loop.)

If `test_node`'s value does not equal `target_value` and is not less than `target_value`, it must be greater than `target_value`. In that case, the algorithm sets `test_node` equal to its right child. (Again, if `test_node` is at the bottom of the tree, its `RightChild` is `null`, and the algorithm handles the situation the next time it goes through the loop.)

The variable `test_node` moves down through the tree and eventually either finds the target value or falls off the tree when `test_node` is `null`.

Understanding this algorithm's performance becomes a question of how far down the tree `test_node` must move before it finds `target_value` or falls off the tree.

Sometimes the algorithm gets lucky and finds the target value right away. If the target value is 7 in Figure 1-1, the algorithm finds it in one step and stops.

Even if the target value isn't at the root node—for example, if it's 4—the program might have to check only a bit of the tree before stopping.

In the worst case, however, the algorithm needs to search the tree from top to bottom.

In fact, roughly half the tree's nodes are the nodes at the bottom that have missing children. If the tree were a *full complete* tree, with every node having exactly zero or two children, the bottom level would hold exactly half the tree's nodes. That means if you search for randomly chosen values in the tree, the algorithm will have to travel through most of the tree's height most of the time.

Now the question is, "How tall is the tree?" A full complete binary tree of height H has $2^{H+1} - 1$ nodes. To look at it from the other direction, a full complete binary tree that contains N nodes has height $\log_2(N + 1) - 1$. Because the algorithm searches the tree from top to bottom in the worst (and average) case, and because the tree has a height of roughly $\log_2(N)$, the algorithm runs in $O(\log_2(N))$ time.

At this point a curious feature of logarithms comes into play. You can convert a logarithm from base A to base B using this formula:

$$\log_B(x) = \log_A(x) \,/\, \log_A(B)$$

Setting B = 2, you can use this formula to convert the value $O(\log_2(N)$ into any other log base A:

$$O(\log_2(N)) = O(\log_A(N) \,/\, \log_A(2))$$

The value $1 / \log_A(2)$ is a constant for any given A, and Big O notation ignores constant multiples, so that means $O(\log_2(N))$ is the same as $O(\log_A(N))$ for any log base A. For that reason, this runtime is often written O(log N) with no indication of the base (and no parentheses to make it look less cluttered).

This algorithm is typical of many algorithms that have O(log N) performance. At each step, it divides roughly in half the number of items it must consider.

Because the log base doesn't matter in Big O notation, it doesn't matter which fraction the algorithm uses to divide the items it is considering. This example divides the number of items in half at each step, which is common for many logarithmic algorithms. But it would still have O(log N) performance if it divided the remaining items by a factor of 1/10th and made lots of progress at each step, or if it divided the items by a factor of 9/10ths and made relatively little progress.

The logarithmic function log(N) grows relatively slowly as N increases, so algorithms with O(log N) performance generally are fast enough to be useful.

Sqrt N

Some algorithms have O(sqrt(N)) performance (where sqrt is the square root function), but they're not common, and none are covered in this book. This function grows very slowly but a bit faster than log(N).

N

The `FindLargest` algorithm described in the earlier section "Rule 1" has O(N) performance. See that section for an explanation of why it has O(N) performance.

The function N grows more quickly than log(N) and sqrt(N) but still not too quickly, so most algorithms that have O(N) performance work quite well in practice.

N log N

Suppose an algorithm loops over all the items in its problem set and then, for each loop, performs some sort of O(log N) calculation on that item. In that case, the algorithm has O(N × log N) or O(N log N) performance.

Alternatively, an algorithm might perform some sort of O(log N) operation and, for each step in it, do something to each of the items in the problem.

For example, suppose you have built a sorted tree containing N items as described earlier. You also have an array of N values and you want to know which values in the array are also in the tree.

One approach would be to loop through the values in the array. For each value, you could use the method described earlier to search the tree for that value. The algorithm examines N items and for each it performs log(N) steps so the total runtime is O(N log N).

Many sorting algorithms that work by comparing items have an O(N log N) runtime. In fact, it can be proven that *any* algorithm that sorts by comparing items must use at least O(N log N) steps, so this is the best you can do, at least in Big O notation. Some algorithms are still faster than others because of the constants that Big O notation ignores.

N^2

An algorithm that loops over all its inputs and then for each input loops over the inputs again has $O(N^2)$ performance. For example, the `ContainsDuplicates` algorithm described earlier, in the section "Rule 4," runs in $O(N^2)$ time. See that section for a description and analysis of the algorithm.

Other powers of N, such as $O(N^3)$ and $O(N^4)$, are possible and are obviously slower than $O(N^2)$.

An algorithm is said to have *polynomial runtime* if its runtime involves any polynomial involving N. O(N), $O(N^2)$, $O(N^6)$, and even $O(N^{4000})$ are all polynomial runtimes.

Polynomial runtimes are important because in some sense these problems can still be solved. The exponential and factorial runtimes described next grow extremely quickly, so algorithms that have those runtimes are practical for only very small numbers of inputs.

2^N

Exponential functions such as 2^N grow extremely quickly, so they are practical for only small problems. Typically algorithms with these runtimes look for optimal selection of the inputs.

For example, consider the knapsack problem. You are given a set of objects that each has a weight and a value. You also have a knapsack that can hold a certain amount of weight. You can put a few heavy items in the knapsack, or you can put lots of lighter items in it. The challenge is to select the items with the greatest total value that fit in the knapsack.

This may seem like an easy problem, but the only known algorithms for finding the best possible solution essentially require you to examine every possible combination of items.

To see how many combinations are possible, note that each item is either in the knapsack or out of it, so each item has two possibilities. If you multiply the number of possibilities for the items, you get $2 \times 2 \times ... \times 2 = 2^N$ total possible selections.

Sometimes you don't have to try every possible combination. For example, if adding the first item fills the knapsack completely, you don't need to add any selections that include the first item plus another item. In general, however, you cannot exclude enough possibilities to narrow the search significantly.

For problems with exponential runtimes, you often need to use *heuristics*—algorithms that usually produce good results but that you cannot guarantee will produce the best possible results.

N!

The factorial function, written N! and pronounced "N factorial," is defined for integers greater than 0 by $N! = 1 \times 2 \times 3 \times ... \times N$. This function grows much more quickly than even the exponential function 2^N. Typically algorithms with factorial runtimes look for an optimal arrangement of the inputs.

For example, in the traveling salesman problem (TSP), you are given a list of cities. The goal is to find a route that visits every city exactly once and returns to the starting point while minimizing the total distance traveled.

This isn't too hard with just a few cities, but with many cities the problem becomes challenging. The most obvious approach is to try every possible arrangement of cities. Following that algorithm, you can pick N possible cities for the first city. After making that selection, you have $N - 1$ possible cities to visit next. Then there are $N - 2$ possible third cities, and so forth, so the total number of arrangements is $N \times (N - 1) \times (N - 2) \times ... \times 1 = N!$.

Visualizing Functions

Table 1-1 shows a few values for the runtime functions described in the preceding sections so that you can see how quickly these functions grow.

Table 1-1: Function Values for Various Inputs

N	$LOG_2(n)$	SQRT(n)	n	n^2	2^n	n!
1	0	1	1	1	2	1
5	2.32	2.23	5	25	32	625
10	3.32	3.16	10	100	1,024	1.0×10^{9}
15	3.90	3.87	15	225	3.3×10^{4}	2.9×10^{16}
20	4.32	4.47	20	400	1.0×10^{6}	5.24×10^{24}
50	5.64	7.07	50	2,500	1.1×10^{15}	1.8×10^{83}
100	6.64	10	100	1×10^{4}	1.3×10^{30}	1.0×10^{198}
1000	9.96	31.62	1,000	1×10^{6}	1.1×10^{301}	—
10000	13.28	100	1×10^{4}	1×10^{8}	—	—
100000	16.60	316.22	1×10^{5}	1×10^{10}	—	—

Figure 1-2 shows a graph of these functions. Some of the functions have been scaled so that they fit better on the graph, but you can easily see which grows fastest when x grows large. Even dividing by 100 doesn't keep the factorial function on the graph for very long.

Practical Considerations

Although theoretical behavior is important in understanding an algorithm's runtime behavior, practical considerations also play an important role in real-world performance for several reasons.

The analysis of an algorithm typically considers all steps as taking the same amount of time even though that may not be the case. Creating and destroying new objects, for example, may take much longer than moving integer values from one part of an array to another. In that case an algorithm that uses arrays may outperform one that uses lots of objects even though the second algorithm does better in Big O notation.

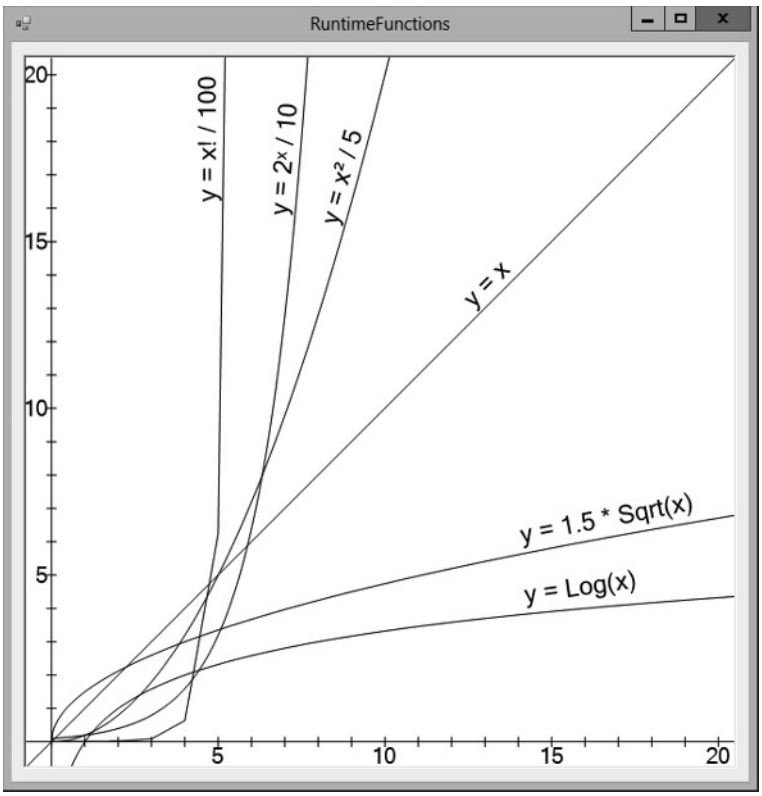

Figure 1-2: The log, sqrt, linear, and even polynomial functions grow at a reasonable pace, but exponential and factorial functions grow incredibly quickly.

Many programming environments also provide access to operating system functions that are more efficient than basic algorithmic techniques. For example, part of the insertionsort algorithm requires you to move some of the items in an array down one position so that you can insert a new item before them. This is a fairly slow process and contributes greatly to the algorithm's $O(N^2)$ performance. However, many programs can use a function (such as `RtlMoveMemory` in .NET programs and `MoveMemory` in Windows C++ programs) that moves blocks of memory all at once. Instead of walking through the array, moving items one at a time, a program can call these functions to move the whole set of array values at once, making the program much faster.

Just because an algorithm has a certain theoretical asymptotic performance doesn't mean you can't take advantage of whatever tools your programming environment offers to improve performance. Some programming environments also provide tools that can perform the same tasks as some of the algorithms described in this book. For example, many libraries include sorting routines that do a very good job of sorting arrays. Microsoft's .NET Framework, used by C# and Visual Basic, includes an `Array.Sort` method that uses an implementation

that you are unlikely to beat using your own code—at least in general. For specific problems you can still beat `Array.Sort`'s performance if you have extra information about the data. (For more information, read about countingsort in Chapter 6.)

Special-purpose libraries may also be available that can help you with certain tasks. For example, you may be able to use a network analysis library instead of writing your own network tools. Similarly, database tools may save you a lot of work building trees and sorting things. You may get better performance building your own balanced trees, but using a database is a lot less work.

If your programming tools include functions that perform the tasks of one of these algorithms, by all means use them. You may get better performance than you could achieve on your own, and you'll certainly have less debugging to do.

Finally, the best algorithm isn't always the one that is fastest for very large problems. If you're sorting a huge list of numbers, quicksort usually provides good performance. If you're sorting only three numbers, a simple series of `If` statements will probably give better performance and will be a lot simpler. Even if quicksort does give better performance, does it matter whether the program finishes sorting in 1 millisecond or 2? Unless you plan to perform the sort many times, you may be better off going with the simpler algorithm that's easier to debug and maintain rather than the complicated one to save 1 millisecond.

If you use libraries such as those described in the preceding paragraphs, you may not need to code all these algorithms yourself, but it's still useful to understand how the algorithms work. If you understand the algorithms, you can take better advantage of the tools that implement them even if you don't write them. For example, if you know that relational databases typically use B-trees (and similar trees) to store their indices, you'll have a better understanding of how important pre-allocation and fill factors are. If you understand quicksort, you'll know why some people think the .NET Framework's `Array.Sort` method is not cryptographically secure. (This is discussed in the section "Using Quicksort" in Chapter 6.)

Understanding the algorithms also lets you apply them to other situations. You may not need to use mergesort, but you may be able to use its divide-and-conquer approach to solve some other problem on multiple processors.

Summary

To get the most out of an algorithm, you not only need to understand how it works, but you also need to understand its performance characteristics. This chapter explained Big O notation, which you can use to study an algorithm's performance. If you know an algorithm's Big O runtime behavior, you can estimate how much the runtime will change if you change the problem size.

This chapter also described some algorithmic situations that lead to common runtime functions. Figure 1-2 showed graphs of these equations so that you can get a feel for just how quickly each grows as the problem size increases. As a rule of thumb, algorithms that run in polynomial time are often fast enough that you can run them for moderately large problems. Algorithms with exponential or factorial runtimes, however, grow extremely quickly as the problem size increases, so you can run them only with relatively small problem sizes.

Now that you have some understanding of how to analyze algorithm speeds, you're ready to study some specific algorithms. The next chapter discusses numerical algorithms. They tend not to require elaborate data structures, so they usually are quite fast.

Exercises

Asterisks indicate particularly difficult problems.

1. The section "Rule 4" described a `ContainsDuplicates` algorithm that has runtime $O(N^2)$. Consider the following improved version of that algorithm:

```
Boolean: ContainsDuplicates(Integer: array[])
    // Loop over all of the array's items except the last one.
    For i = 0 To <largest index> - 1
        // Loop over the items after item i.
        For j = i + 1 To <largest index>
            // See if these two items are duplicates.
            If (array[i] == array[j]) Then Return True
        Next j
    Next i

    // If we get to this point, there are no duplicates.
    Return False
End ContainsDuplicates
```

   What is the runtime of this new version?

2. Table 1-1 shows the relationship between problem size N and various runtime functions. Another way to study that relationship is to look at the largest problem size that a computer with a certain speed could execute within a given amount of time.

   For example, suppose a computer can execute 1 million algorithm steps per second. Consider an algorithm that runs in $O(N^2)$ time. In 1 hour the computer could solve a problem where N = 60,000 (because $60{,}000^2 = 3{,}600{,}000{,}000$, which is the number of steps the computer can execute in 1 hour).

Make a table showing the largest problem size N that this computer could execute for each of the functions listed in Table 1-1 in one second, minute, hour, day, week, and year.

3. Sometimes the constants that you ignore in Big O notation are important. For example, suppose you have two algorithms that can do the same job. The first requires 1,500 × N steps, and the other requires $30 \times N^2$ steps. For what values of N would you choose each algorithm?
4. *Suppose you have two algorithms—one that uses $N^3 / 75 - N^2 / 4 + N + 10$ steps, and one that uses $N / 2 + 8$ steps. For what values of N would you choose each algorithm?
5. Suppose a program takes as inputs N letters and generates all possible unordered pairs of the letters. For example, with inputs ABCD, the program generates the combinations AB, AC, AD, BC, BD, and CD. (Here unordered means that AB and BA count as the same pair.) What is the algorithm's runtime?
6. Suppose an algorithm with N inputs generates values for each unit square on the surface of an N × N × N cube. What is the algorithm's runtime?
7. Suppose an algorithm with N inputs generates values for each unit cube on the edges of an N × N × N cube, as shown in Figure 1-3. What is the algorithm's runtime?

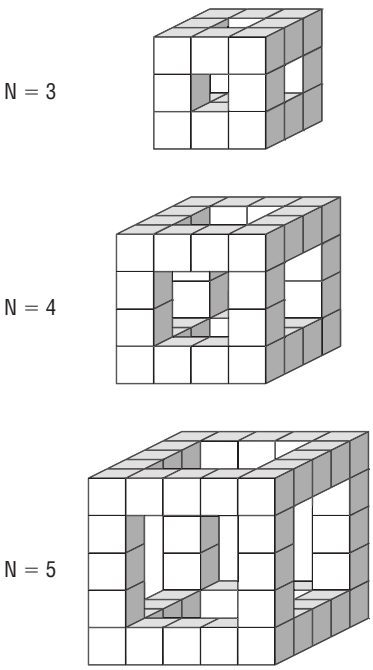

Figure 1-3: This algorithm generates values for cubes on a cube's "skeleton."

8. *Suppose you have an algorithm that, for N inputs, generates a value for each small cube in the shapes shown in Figure 1-4. Assuming that the obvious hidden cubes are present so that the shapes in the figure are not hollow, what is the algorithm's runtime?

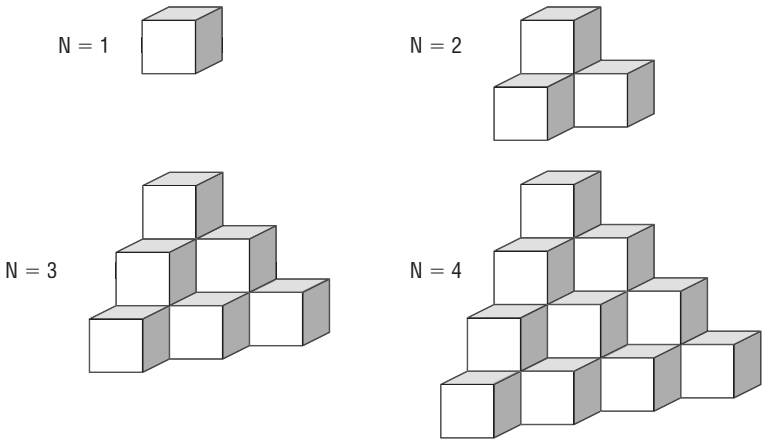

Figure 1-4: This algorithm adds one more level to the shape as N increases.

9. Can you have an algorithm without a data structure? Can you have a data structure without an algorithm?
10. Consider the following two algorithms for painting a fence:

```
Algorithm1()
    For i = 0 To <number of boards in fence> - 1
        <paint board number i>
    Next i
End Algorithm1

Algorithm2(Integer: first_board, Integer: last_board)
    If (first_board == last_board) Then
        // There's only one board. Just paint it.
        <paint board number first_board>
    Else
        // There's more than one board. Divide the boards
        // into two groups and recursively paint them.
        Integer: middle_board = (first_board + last_board) / 2
        Algorithm2(first_board, middle_board)
        Algorithm2(middle_board, last_board)
    End If
End Algorithm2
```

What are the runtimes for these two algorithms, where N is the number of boards in the fence? Which algorithm is better?

11. *Fibonacci numbers can be defined recursively by the following rules:

```
Fibonacci(0) = 1
Fibonacci(1) = 1
Fibonacci(n) = Fibonacci(n - 1) + Fibonacci(n - 2)
```

The Fibonacci sequence starts with the values 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89.

How does the Fibonacci function compare to the runtime functions shown in Figure 1-2?