Algorithm Analysis

An algorithm is a clearly specified set of simple instructions to be followed to solve a problem. Once an algorithm is given for a problem and decided (somehow) to be correct, an important step is to determine how much in the way of resources, such as time or space, the algorithm will require. An algorithm that solves a problem but requires a year is hardly of any use. Likewise, an algorithm that requires thousands of gigabytes of main memory is not (currently) useful on most machines.

In this chapter, we shall discuss...

- How to estimate the time required for a program.
- How to reduce the running time of a program from days or years to fractions of a second.
- The results of careless use of recursion.
- Very efficient algorithms to raise a number to a power and to compute the greatest common divisor of two numbers.

2.1 Mathematical Background

The analysis required to estimate the resource use of an algorithm is generally a theoretical issue, and therefore a formal framework is required. We begin with some mathematical definitions.

Throughout this book, we will use the following four definitions:

**Definition 2.1**

\[ T(N) = O(f(N)) \] if there are positive constants \( c \) and \( n_0 \) such that \( T(N) \leq cf(N) \) when \( N \geq n_0 \).

**Definition 2.2**

\[ T(N) = \Omega(g(N)) \] if there are positive constants \( c \) and \( n_0 \) such that \( T(N) \geq cg(N) \) when \( N \geq n_0 \).

**Definition 2.3**

\[ T(N) = \Theta(h(N)) \] if and only if \( T(N) = O(h(N)) \) and \( T(N) = \Omega(h(N)) \).

**Definition 2.4**

\[ T(N) = o(p(N)) \] if, for all positive constants \( c \), there exists an \( n_0 \) such that \( T(N) < cp(N) \) when \( N > n_0 \). Less formally, \( T(N) = o(p(N)) \) if \( T(N) = O(p(N)) \) and \( T(N) \neq \Theta(p(N)) \).
The idea of these definitions is to establish a relative order among functions. Given two functions, there are usually points where one function is smaller than the other. So it does not make sense to claim, for instance, \( f(N) < g(N) \). Thus, we compare their **relative rates of growth**. When we apply this to the analysis of algorithms, we shall see why this is the important measure.

Although \( 1,000N \) is larger than \( N^2 \) for small values of \( N \), \( N^2 \) grows at a faster rate, and thus \( N^2 \) will eventually be the larger function. The turning point is \( N = 1,000 \) in this case. The first definition says that eventually there is some point \( n_0 \) past which \( c \cdot f(N) \) is always at least as large as \( T(N) \), so that if constant factors are ignored, \( f(N) \) is at least as big as \( T(N) \). In our case, we have \( T(N) = 1,000N, f(N) = N^2, n_0 = 1,000 \), and \( c = 1 \). We could also use \( n_0 = 10 \) and \( c = 100 \). Thus, we can say that \( 1,000N = O(N^2) \) (order \( N \)-squared). This notation is known as **Big-Oh notation**. Frequently, instead of saying “order . . . ,” one says “Big-Oh . . . .”

If we use the traditional inequality operators to compare growth rates, then the first definition says that the growth rate of \( T(N) \) is less than or equal to (\( \leq \)) that of \( f(N) \). The second definition, \( T(N) = \Omega(g(N)) \) (pronounced “omega”), says that the growth rate of \( T(N) \) is greater than or equal to (\( \geq \)) that of \( g(N) \). The third definition, \( T(N) = \Theta(h(N)) \) (pronounced “theta”), says that the growth rate of \( T(N) \) equals (\( = \)) the growth rate of \( h(N) \). The last definition, \( T(N) = o(p(N)) \) (pronounced “little-oh”), says that the growth rate of \( T(N) \) is less than (\( < \)) the growth rate of \( p(N) \). This is different from Big-Oh, because Big-Oh allows the possibility that the growth rates are the same.

To prove that some function \( T(N) = O(f(N)) \), we usually do not apply these definitions formally but instead use a repertoire of known results. In general, this means that a proof (or determination that the assumption is incorrect) is a very simple calculation and should not involve calculus, except in extraordinary circumstances (not likely to occur in an algorithm analysis).

When we say that \( T(N) = O(f(N)) \), we are guaranteeing that the function \( T(N) \) grows at a rate no faster than \( f(N) \); thus \( f(N) \) is an **upper bound** on \( T(N) \). Since this implies that \( f(N) = \Omega(T(N)) \), we say that \( T(N) \) is a **lower bound** on \( f(N) \).

As an example, \( N^3 \) grows faster than \( N^2 \), so we can say that \( N^2 = O(N^3) \) or \( N^3 = \Omega(N^2) \). \( f(N) = N^2 \) and \( g(N) = 2N^2 \) grow at the same rate, so both \( f(N) = O(g(N)) \) and \( f(N) = \Omega(g(N)) \) are true. When two functions grow at the same rate, then the decision of whether or not to signify this with \( \Theta() \) can depend on the particular context. Intuitively, if \( g(N) = 2N^2 \), then \( g(N) = O(N^3) \), \( g(N) = O(N^3) \), and \( g(N) = O(N^2) \) are all technically correct, but the last option is the best answer. Writing \( g(N) = \Theta(N^2) \) says not only that \( g(N) = O(N^2) \) but also that the result is as good (tight) as possible.

Here are the important things to know:

**Rule 1**

If \( T_1(N) = O(f(N)) \) and \( T_2(N) = O(g(N)) \), then

(a) \( T_1(N) + T_2(N) = O(f(N) + g(N)) \) (intuitively and less formally it is \( O(\max(f(N), g(N))) \)),

(b) \( T_1(N) \cdot T_2(N) = O(f(N) \cdot g(N)) \).

**Rule 2**

If \( T(N) \) is a polynomial of degree \( k \), then \( T(N) = \Theta(N^k) \).
2.1 Mathematical Background

<table>
<thead>
<tr>
<th>Function</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>Constant</td>
</tr>
<tr>
<td>$\log N$</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>$\log^2 N$</td>
<td>Log-squared</td>
</tr>
<tr>
<td>$N$</td>
<td>Linear</td>
</tr>
<tr>
<td>$N \log N$</td>
<td></td>
</tr>
<tr>
<td>$N^2$</td>
<td>Quadratic</td>
</tr>
<tr>
<td>$N^3$</td>
<td>Cubic</td>
</tr>
<tr>
<td>$2^N$</td>
<td>Exponential</td>
</tr>
</tbody>
</table>

**Figure 2.1** Typical growth rates

**Rule 3**

$\log^k N = O(N)$ for any constant $k$. This tells us that logarithms grow very slowly.

This information is sufficient to arrange most of the common functions by growth rate (see Fig. 2.1).

Several points are in order. First, it is very bad style to include constants or low-order terms inside a Big-Oh. Do not say $T(N) = O(2N^2)$ or $T(N) = O(N^2 + N)$. In both cases, the correct form is $T(N) = O(N^2)$. This means that in any analysis that will require a Big-Oh answer, all sorts of shortcuts are possible. Lower-order terms can generally be ignored, and constants can be thrown away. Considerably less precision is required in these cases.

Second, we can always determine the relative growth rates of two functions $f(N)$ and $g(N)$ by computing $\lim_{N \to \infty} f(N)/g(N)$, using L'Hôpital's rule if necessary. The limit can have four possible values:

- The limit is 0: This means that $f(N) = o(g(N))$.
- The limit is $c \neq 0$: This means that $f(N) = \Theta(g(N))$.
- The limit is $\infty$: This means that $g(N) = o(f(N))$.
- The limit does not exist: There is no relation (this will not happen in our context).

Using this method almost always amounts to overkill. Usually the relation between $f(N)$ and $g(N)$ can be derived by simple algebra. For instance, if $f(N) = N \log N$ and $g(N) = N^{1.5}$, then to decide which of $f(N)$ and $g(N)$ grows faster, one really needs to determine which of $\log N$ and $N^{0.5}$ grows faster. This is like determining which of $\log^2 N$ or $N$ grows faster. This is a simple problem, because it is already known that $N$ grows faster than any power of a log. Thus, $g(N)$ grows faster than $f(N)$.

One stylistic note: It is bad to say $f(N) \leq O(g(N))$, because the inequality is implied by the definition. It is wrong to write $f(N) \geq O(g(N))$, because it does not make sense.

---

1 L'Hôpital's rule states that if $\lim_{N \to \infty} f(N) = \infty$ and $\lim_{N \to \infty} g(N) = \infty$, then $\lim_{N \to \infty} f(N)/g(N) = \lim_{N \to \infty} f'(N)/g'(N)$, where $f'(N)$ and $g'(N)$ are the derivatives of $f(N)$ and $g(N)$, respectively.
As an example of the typical kinds of analyses that are performed, consider the problem of downloading a file over the Internet. Suppose there is an initial 3-sec delay (to set up a connection), after which the download proceeds at 1.5M(bytes)/sec. Then it follows that if the file is \( N \) megabytes, the time to download is described by the formula \( T(N) = N/1.5 + 3 \). This is a linear function. Notice that the time to download a 1,500M file (1,003 sec) is approximately (but not exactly) twice the time to download a 750M file (503 sec). This is typical of a linear function. Notice, also, that if the speed of the connection doubles, both times decrease, but the 1,500M file still takes approximately twice the time to download as a 750M file. This is the typical characteristic of linear-time algorithms, and it is the reason we write \( T(N) = O(N) \), ignoring constant factors. (Although using big-theta would be more precise, Big-Oh answers are typically given.)

Observe, too, that this behavior is not true of all algorithms. For the first selection algorithm described in Section 1.1, the running time is controlled by the time it takes to perform a sort. For a simple sorting algorithm, such as the suggested bubble sort, when the amount of input doubles, the running time increases by a factor of four for large amounts of input. This is because those algorithms are not linear. Instead, as we will see when we discuss sorting, trivial sorting algorithms are \( O(N^2) \), or quadratic.

### 2.2 Model

In order to analyze algorithms in a formal framework, we need a model of computation. Our model is basically a normal computer in which instructions are executed sequentially. Our model has the standard repertoire of simple instructions, such as addition, multiplication, comparison, and assignment, but, unlike the case with real computers, it takes exactly one time unit to do anything (simple). To be reasonable, we will assume that, like a modern computer, our model has fixed-size (say, 32-bit) integers and no fancy operations, such as matrix inversion or sorting, which clearly cannot be done in one time unit. We also assume infinite memory.

This model clearly has some weaknesses. Obviously, in real life, not all operations take exactly the same time. In particular, in our model, one disk reads counts the same as an addition, even though the addition is typically several orders of magnitude faster. Also, by assuming infinite memory, we ignore the fact that the cost of a memory access can increase when slower memory is used due to larger memory requirements.

### 2.3 What to Analyze

The most important resource to analyze is generally the running time. Several factors affect the running time of a program. Some, such as the compiler and computer used, are obviously beyond the scope of any theoretical model, so, although they are important, we cannot deal with them here. The other main factors are the algorithm used and the input to the algorithm.

Typically, the size of the input is the main consideration. We define two functions, \( T_{\text{avg}}(N) \) and \( T_{\text{worst}}(N) \), as the average and worst-case running time, respectively, used by an algorithm on input of size \( N \). Clearly, \( T_{\text{avg}}(N) \leq T_{\text{worst}}(N) \). If there is more than one input, these functions may have more than one argument.
Occasionally, the best-case performance of an algorithm is analyzed. However, this is often of little interest, because it does not represent typical behavior. Average-case performance often reflects typical behavior, while worst-case performance represents a guarantee for performance on any possible input. Notice also that, although in this chapter we analyze C++ code, these bounds are really bounds for the algorithms rather than programs. Programs are an implementation of the algorithm in a particular programming language, and almost always the details of the programming language do not affect a Big-Oh answer. If a program is running much more slowly than the algorithm analysis suggests, there may be an implementation inefficiency. This can occur in C++ when arrays are inadvertently copied in their entirety, instead of passed with references. Another extremely subtle example of this is in the last two paragraphs of Section 12.6. Thus in future chapters, we will analyze the algorithms rather than the programs.

Generally, the quantity required is the worst-case time, unless otherwise specified. One reason for this is that it provides a bound for all input, including particularly bad input, which an average-case analysis does not provide. The other reason is that average-case bounds are usually much more difficult to compute. In some instances, the definition of “average” can affect the result. (For instance, what is average input for the following problem?)

As an example, in the next section, we shall consider the following problem:

**Maximum Subsequence Sum Problem**

Given (possibly negative) integers $A_1, A_2, \ldots, A_N$, find the maximum value of $\sum_{k=i}^{j} A_k$. (For convenience, the maximum subsequence sum is 0 if all the integers are negative.)

Example:

For input $-2, 11, -4, 13, -5, -2$, the answer is 20 ($A_2$ through $A_4$).

This problem is interesting mainly because there are so many algorithms to solve it, and the performance of these algorithms varies drastically. We will discuss four algorithms to solve this problem. The running time on some computers (the exact computer is unimportant) for these algorithms is given in Figure 2.2.

There are several important things worth noting in this table. For a small amount of input, the algorithms all run in the blink of an eye. So if only a small amount of input is

\begin{table}[h]
\centering
\begin{tabular}{lcccc}
\hline
Input Size & 1 & 2 & 3 & 4 \\
\hline
$N = 100$ & $O(N^3)$ & $O(N^2)$ & $O(N \log N)$ & $O(N)$ \\
0.000159 & 0.000006 & 0.000005 & 0.000002 \\
$N = 1,000$ & 0.095857 & 0.000371 & 0.000060 & 0.000022 \\
86.67 & 0.033322 & 0.000619 & 0.000222 \\
$N = 100,000$ & NA & 3.33 & 0.006700 & 0.002205 \\
$N = 1,000,000$ & NA & NA & 0.074870 & 0.022711 \\
\hline
\end{tabular}
\caption{Running times of several algorithms for maximum subsequence sum (in seconds)}
\end{table}
expected, it might be silly to expend a great deal of effort to design a clever algorithm. On the other hand, there is a large market these days for rewriting programs that were written five years ago based on a no-longer-valid assumption of small input size. These programs are now too slow because they used poor algorithms. For large amounts of input, algorithm 4 is clearly the best choice (although algorithm 3 is still usable).

Second, the times given do not include the time required to read the input. For algorithm 4, the time merely to read the input from a disk is likely to be an order of magnitude larger than the time required to solve the problem. This is typical of many efficient algorithms. Reading the data is generally the bottleneck; once the data are read, the problem can be solved quickly. For inefficient algorithms this is not true, and significant computer resources must be used. Thus, it is important that, whenever possible, algorithms be efficient enough not to be the bottleneck of a problem.

Notice that for algorithm 4, which is linear, as the problem size increases by a factor of 10, so does the running time. Algorithm 2, which is quadratic, does not display this behavior; a tenfold increase in input size yields roughly a hundredfold \(10^2\) increase in running time. And algorithm 1, which is cubic, yields a thousandfold \(10^3\) increase in running time. We would expect algorithm 1 to take nearly 9,000 seconds (or two and a half hours) to complete for \(N = 100,000\). Similarly, we would expect algorithm 2 to take roughly 333 seconds to complete for \(N = 1,000,000\). However, it is possible that algorithm 2 could take somewhat longer to complete due to the fact that \(N = 1,000,000\) could also yield slower memory accesses than \(N = 100,000\) on modern computers, depending on the size of the memory cache.

Figure 2.3 shows the growth rates of the running times of the four algorithms. Even though this graph encompasses only values of \(N\) ranging from 10 to 100, the relative

![Figure 2.3](image_url)  
**Figure 2.3** Plot \((N \text{ vs. time})\) of various algorithms
2.4 Running-Time Calculations

There are several ways to estimate the running time of a program. The previous table was obtained empirically. If two programs are expected to take similar times, probably the best way to decide which is faster is to code them both and run them!

Generally, there are several algorithmic ideas, and we would like to eliminate the bad ones early, so an analysis is usually required. Furthermore, the ability to do an analysis usually provides insight into designing efficient algorithms. The analysis also generally pinpoints the bottlenecks, which are worth coding carefully.

To simplify the analysis, we will adopt the convention that there are no particular units of time. Thus, we throw away leading constants. We will also throw away low-order terms, so what we are essentially doing is computing a Big-Oh running time. Since Big-Oh is an upper bound, we must be careful never to underestimate the running time of the program. In effect, the answer provided is a guarantee that the program will terminate within a certain time period. The program may stop earlier than this, but never later.
2.4.1 A Simple Example

Here is a simple program fragment to calculate $\sum_{i=1}^{N} i^3$:

```c
int sum( int n )
{
    int partialSum;
    partialSum = 0;
    for( int i = 1; i <= n; ++i )
        partialSum += i * i * i;
    return partialSum;
}
```

The analysis of this fragment is simple. The declarations count for no time. Lines 1 and 4 count for one unit each. Line 3 counts for four units per time executed (two multiplications, one addition, and one assignment) and is executed $N$ times, for a total of $4N$ units. Line 2 has the hidden costs of initializing $i$, testing $i \leq N$, and incrementing $i$. The total cost of all these is 1 to initialize, $N + 1$ for all the tests, and $N$ for all the increments, which is $2N + 2$. We ignore the costs of calling the function and returning, for a total of $6N + 4$. Thus, we say that this function is $O(N)$.

If we had to perform all this work every time we needed to analyze a program, the task would quickly become infeasible. Fortunately, since we are giving the answer in terms of Big-Oh, there are lots of shortcuts that can be taken without affecting the final answer. For instance, line 3 is obviously an $O(1)$ statement (per execution), so it is silly to count precisely whether it is two, three, or four units; it does not matter. Line 1 is obviously insignificant compared with the for loop, so it is silly to waste time here. This leads to several general rules.

2.4.2 General Rules

**Rule 1—FOR loops**

The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.

**Rule 2—Nested loops**

Analyze these inside out. The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

As an example, the following program fragment is $O(N^2)$:

```c
for( i = 0; i < n; ++i )
    for( j = 0; j < n; ++j )
        ++k;
```

**Rule 3—Consecutive Statements**

These just add (which means that the maximum is the one that counts; see rule 1 on page 52).
As an example, the following program fragment, which has $O(N)$ work followed by $O(N^2)$ work, is also $O(N^2)$:

```c
for( i = 0; i < n; ++i )
    a[ i ] = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < n; ++j )
        a[ i ] += a[ j ] + i + j;
```

**Rule 4—If/Else**

For the fragment

```c
if( condition )
    S1
else
    S2
```

the running time of an *if*/*else* statement is never more than the running time of the test plus the larger of the running times of S1 and S2.

Clearly, this can be an overestimate in some cases, but it is never an underestimate.

Other rules are obvious, but a basic strategy of analyzing from the inside (or deepest part) out works. If there are function calls, these must be analyzed first. If there are recursive functions, there are several options. If the recursion is really just a thinly veiled *for* loop, the analysis is usually trivial. For instance, the following function is really just a simple loop and is $O(N)$:

```c
long factorial( int n )
{
    if( n <= 1 )
        return 1;
    else
        return n * factorial( n - 1 );
}
```

This example is really a poor use of recursion. When recursion is properly used, it is difficult to convert the recursion into a simple loop structure. In this case, the analysis will involve a recurrence relation that needs to be solved. To see what might happen, consider the following program, which turns out to be a terrible use of recursion:

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

At first glance, this seems like a very clever use of recursion. However, if the program is coded up and run for values of $N$ around 40, it becomes apparent that this program
is terribly inefficient. The analysis is fairly simple. Let $T(N)$ be the running time for the function call $\text{fib}(n)$. If $N = 0$ or $N = 1$, then the running time is some constant value, which is the time to do the test at line 1 and return. We can say that $T(0) = T(1) = 1$ because constants do not matter. The running time for other values of $N$ is then measured relative to the running time of the base case. For $N > 2$, the time to execute the function is the constant work at line 1 plus the work at line 3. Line 3 consists of an addition and two function calls. Since the function calls are not simple operations, they must be analyzed by themselves. The first function call is $\text{fib}(n-1)$ and hence, by the definition of $T$, requires $T(N - 1)$ units of time. A similar argument shows that the second function call requires $T(N - 2)$ units of time. The total time required is then $T(N - 1) + T(N - 2) + 2$, where the 2 accounts for the work at line 1 plus the addition at line 3. Thus, for $N \geq 2$, we have the following formula for the running time of $\text{fib}(n)$:

$$T(N) = T(N - 1) + T(N - 2) + 2$$

Since $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$, it is easy to show by induction that $T(N) \geq \text{fib}(n)$. In Section 1.2.5, we showed that $\text{fib}(N) < (5/3)^N$. A similar calculation shows that (for $N > 4$) $\text{fib}(N) \geq (3/2)^N$, and so the running time of this program grows exponentially. This is about as bad as possible. By keeping a simple array and using a for loop, the running time can be reduced substantially.

This program is slow because there is a huge amount of redundant work being performed, violating the fourth major rule of recursion (the compound interest rule), which was presented in Section 1.3. Notice that the first call on line 3, $\text{fib}(n-1)$, actually computes $\text{fib}(n-2)$ at some point. This information is thrown away and recomputed by the second call on line 3. The amount of information thrown away compounds recursively and results in the huge running time. This is perhaps the finest example of the maxim “Don’t compute anything more than once” and should not scare you away from using recursion. Throughout this book, we shall see outstanding uses of recursion.

### 2.4.3 Solutions for the Maximum Subsequence Sum Problem

We will now present four algorithms to solve the maximum subsequence sum problem posed earlier. The first algorithm, which merely exhaustively tries all possibilities, is depicted in Figure 2.5. The indices in the for loop reflect the fact that in C++, arrays begin at 0 instead of 1. Also, the algorithm does not compute the actual subsequences; additional code is required to do this.

Convince yourself that this algorithm works (this should not take much convincing). The running time is $O(N^3)$ and is entirely due to lines 13 and 14, which consist of an $O(1)$ statement buried inside three nested for loops. The loop at line 8 is of size $N$.

The second loop has size $N - i$, which could be small but could also be of size $N$. We must assume the worst, with the knowledge that this could make the final bound a bit high. The third loop has size $j - i + 1$, which again we must assume is of size $N$. The total is $O(1 \cdot N \cdot N \cdot N) = O(N^3)$. Line 6 takes only $O(1)$ total, and lines 16 and 17 take only $O(N^2)$ total, since they are easy expressions inside only two loops.