Chapter 3

Analyzing Algorithms

In Chapter 1, we saw that different algorithms for the same problem can have dramatically different performance. In this chapter, we will introduce techniques for mathematically analyzing the performance of algorithms. These analyses will enable us to predict, to a certain extent, the performance of programs using these algorithms.

3.1 Motivation

Perhaps the most common performance measure of a program is its running time. The running time of a program depends not only on the algorithms it uses, but also on such factors as the speed of the processor(s), the amount of main memory available, the speeds of devices accessed, and the impact of other software utilizing the same resources. Furthermore, the same algorithm can perform differently when coded in different languages, even when all other factors remain unchanged. When analyzing the performance of an algorithm, we would like to learn something about the running time of any of its implementations, regardless of the impact of these other factors.

Suppose we divide an execution of an algorithm into a sequence of steps, each of which does some fixed amount of work. For example, a step could be comparing two values or performing a single arithmetic operation. Assuming the values used are small enough to fit into a single machine word, we could reasonably expect that any processor could execute each step in a bounded amount of time. Some of these steps might be faster than others, but for any given processor, we should be able to identify both a lower bound $l > 0$ and an upper bound $u \geq l$ on the amount of time required for any single execution step, assuming no other programs are being executed by that
processor. Thus, if we simply count execution steps, we obtain an estimate on the running time, accurate to within a factor of $u/l$.

Obviously, these bounds will be different for different processors. Thus, if an analysis of an algorithm is to be independent of the platform on which the algorithm runs, the analysis must ignore constant factors. In other words, our analyses will be unable to conclude, for example, that algorithm A is twice as fast (or a million times as fast) as algorithm B. By ignoring constant factors, we therefore lose a great deal of precision in measuring performance. However, we will see that this loss of precision leads us to focus on the more dramatic differences in algorithm performance. These differences are important enough that they tend to transcend the differences in platforms on which an algorithm is executed.

Because we are ignoring constant factors, it only makes sense to consider the behavior of an algorithm on an infinite set of inputs. To see why, consider that the execution times of two algorithms on the same single input are always related by a constant factor — we simply divide the number of steps in one execution by the number of steps in the other. This argument can be extended to any finite set of inputs by dividing the number of steps in the longest execution of one algorithm by the number of steps in the shortest execution of the other.

Mathematically, we will describe the running time of an algorithm by a function $f : \mathbb{N} \rightarrow \mathbb{N}$. The input to $f$ is a natural number representing the size of an input. $f(n)$ then represents the number of steps taken by the algorithm on some particular input of size $n$. The context will determine which input of size $n$ we are considering, but usually we will be interested in the worst-case input — an input of size $n$ resulting in the maximum number of execution steps.

Our analysis will then focus on this function $f$, not its value at specific points. More precisely, we will focus our attention on the behavior of $f(n)$ as $n$ increases. This behavior is known as the asymptotic behavior of $f$. Most algorithms behave well enough if their inputs are small enough. By focusing on asymptotic behavior, we can see how quickly the algorithm’s performance will degrade as it processes larger inputs.

Throughout the remainder of this chapter, we will define various notations that allow us to relate the asymptotic behaviors of various functions to each other. In this context, all functions will be of the form $f : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$, where $\mathbb{R}^{\geq 0}$ denotes the set of nonnegative real numbers (likewise, we will use $\mathbb{R}$ to denote the set of all real numbers and $\mathbb{R}^{>0}$ to denote the set of positive real numbers). Each of the notations we introduce will relate a set of functions to one given function $f$ based on their respective asymptotic
growth rates. Typically, \( f \) will be fairly simple, e.g., \( f(n) = n^2 \). In this way, we will be able describe the growth rates of complicated — or even unknown — functions using well-understood functions like \( n^2 \).

### 3.2 Big-O Notation

**Definition 3.1** Let \( f : \mathbb{N} \to \mathbb{R}_{\geq 0} \). \( O(f(n)) \) is defined to be the set of all functions \( g : \mathbb{N} \to \mathbb{R}_{\geq 0} \) such that for some natural number \( n_0 \) and some strictly positive real number \( c \), \( g(n) \leq cf(n) \) whenever \( n \geq n_0 \).

The above definition formally defines big-O notation. Let us now dissect this definition to see what it means. We start with some specific function \( f \) which maps natural numbers to nonnegative real numbers. \( O(f(n)) \) is then defined to be a set whose elements are all functions. Each of the functions in \( O(f(n)) \) maps natural numbers to nonnegative real numbers. Furthermore, if we consider any function \( g(n) \) in \( O(f(n)) \), then for every sufficiently large \( n \) (i.e., \( n \geq n_0 \)), \( g(n) \) cannot exceed \( f(n) \) by more than some fixed constant factor (i.e., \( g(n) \leq cf(n) \)). Thus, all of the functions in \( O(f(n)) \) grow no faster than some constant multiple of \( f \) as \( n \) becomes sufficiently large. Note that the constants \( n_0 \) and \( c \) may differ for different \( f \) and \( g \), but are the same for all \( n \).

Notice that big-O notation is defined solely in terms of mathematical functions — not in terms of algorithms. Presently, we will show how it can be used to analyze algorithms. First, however, we will give a series of examples illustrating some of its mathematical properties.

**Example 3.2** Let \( f(n) = n^2 \), and let \( g(n) = 2n^2 \). Then \( g(n) \in O(f(n)) \) because \( g(n) \leq 2f(n) \) for every \( n \geq 0 \). Here, the constant \( n_0 \) is 0, and the constant \( c \) is 2.

**Example 3.3** Let \( f(n) = n^2 \), and let \( g(n) = 3n + 10 \). We wish to show that \( g(n) \in O(f(n)) \). Hence, we need to find a positive real number \( c \) and a natural number \( n_0 \) such that \( 3n + 10 \leq cn^2 \) whenever \( n \geq n_0 \). If \( n > 0 \), we can divide both sides of this inequality by \( n \), obtaining an equivalent inequality, \( 3 + 10/n \leq cn \). The left-hand side of this inequality is maximized when \( n \) is minimized. Because we have assumed \( n > 0 \), 1 is the minimum value of \( n \). Thus, if we can satisfy \( cn \geq 13 \), the original inequality will be satisfied. This inequality can be satisfied by choosing \( c = 13 \) and \( n \geq 1 \). Therefore, \( g(n) \in O(f(n)) \).
Example 3.4 \( n^3 \not\in O(n^2) \) because \( n^3 = n(n^2) \), so that whatever values we pick for \( n_0 \) and \( c \), we can find an \( n \geq n_0 \) such that \( n(n^2) > cn^2 \). Note that in this example, we are using \( n^3 \) and \( n^2 \) to denote functions.

Example 3.5 \( 1000 \in O(1) \). Here, \( 1000 \) and \( 1 \) denote constant functions — functions whose values are the same for all \( n \). Thus, for every \( n \geq 0 \), \( 1000 \leq 1000(1) \).

Example 3.6 \( O(n) \subseteq O(n^2) \); i.e., every function in \( O(n) \) is also in \( O(n^2) \). To see this, note that for any function \( f(n) \in O(n) \), there exist a positive real number \( c \) and a natural number \( n_0 \) such that \( f(n) \leq cn \) whenever \( n \geq n_0 \). Furthermore, \( n \leq n^2 \) for all \( n \in \mathbb{N} \). Therefore, \( f(n) \leq cn^2 \) whenever \( n \geq n_0 \).

Example 3.7 \( O(n^2) = O(4n^2 + 7n) \); i.e., the sets \( O(n^2) \) and \( O(4n^2 + 7n) \) contain exactly the same functions. It is easily seen that \( O(n^2) \subseteq O(4n^2 + 7n) \) using an argument similar to that of Example 3.6. Consider any function \( f(n) \in O(4n^2 + 7n) \). There exist a positive real number \( c \) and a natural number \( n_0 \) such that \( f(n) \leq c(4n^2 + 7n) \) whenever \( n \geq n_0 \). Furthermore, \( 4n^2 + 7n \leq 11n^2 \) for all \( n \in \mathbb{N} \). Letting \( c' = 11c \), we therefore have \( f(n) \leq c'n^2 \) whenever \( n \geq n_0 \). Therefore, \( f(n) \in O(n^2) \). Note that although \( O(n^2) \) and \( O(4n^2 + 7n) \) denote the same set of functions, the preferred notation is \( O(n^2) \) because it is simpler.

Let us now illustrate the use of big-\( O \) notation by analyzing the running time of \textsc{MaxSumBU} from Figure 1.14 on page 18. The initialization statements prior to the loop, including the initialization of the loop index \( i \), require a fixed number of steps. Their running time is therefore bounded by some constant \( a \). Likewise, the number of steps required by any single iteration of the loop (including the loop test and the increment of \( i \)) is bounded by some constant \( b \). Because the loop iterates \( n \) times, the total number of steps required by the loop is at most \( bn \). Finally, the last loop condition test and the \texttt{return} statement require a number of steps bounded by some constant \( c \). The running time of the entire algorithm is therefore bounded by \( a + bn + c \), where \( a, b, \) and \( c \) are fixed positive constants. The running time of \textsc{MaxSumBU} is in \( O(n) \), because \( a + bn + c \leq (a+b+c)n \) for all \( n \geq 1 \).

We can simplify the above analysis somewhat using the following theorem.

Theorem 3.8 Suppose \( f_1(n) \in O(g_1(n)) \) and \( f_2(n) \in O(g_2(n)) \). Then
1. \( f_1(n)f_2(n) \in O(g_1(n)g_2(n)) \); and
2. \( f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n))) \).

(By \( f_1(n)f_2(n) \), we mean the function that maps \( n \) to the product of \( f_1(n) \) and \( f_2(n) \). Likewise, \( \max(g_1(n), g_2(n)) \) denotes the function that maps \( n \) to the maximum of \( g_1(n) \) and \( g_2(n) \).)

**Proof:** Because \( f_1(n) \in O(g_1(n)) \) and \( f_2(n) \in O(g_2(n)) \), there exist positive real numbers \( c_1 \) and \( c_2 \) and natural numbers \( n_1 \) and \( n_2 \) such that

\[
f_1(n) \leq c_1g_1(n) \quad \text{whenever } n \geq n_1 \tag{3.1}
\]

and

\[
f_2(n) \leq c_2g_2(n) \quad \text{whenever } n \geq n_2. \tag{3.2}
\]

Because both of the above inequalities involve only nonnegative numbers, we may multiply the inequalities, obtaining

\[
f_1(n)f_2(n) \leq c_1c_2g_1(n)g_2(n)
\]

whenever \( n \geq \max(n_1, n_2) \). Let \( c = c_1c_2 \) and \( n_0 = \max(n_1, n_2) \). Then

\[
f_1(n)f_2(n) \leq cg_1(n)g_2(n)
\]

whenever \( n \geq n_0 \). Therefore, \( f_1(n)f_2(n) \in O(g_1(n)g_2(n)) \).

If we add inequalities (3.1) and (3.2), we obtain

\[
f_1(n) + f_2(n) \leq c_1g_1(n) + c_2g_2(n) \\
\leq c_1 \max(g_1(n), g_2(n)) + c_2 \max(g_1(n), g_2(n)) \\
= (c_1 + c_2) \max(g_1(n), g_2(n))
\]

whenever \( n \geq \max(n_1, n_2) \). Therefore, \( f_1(n)+f_2(n) \in O(\max(g_1(n), g_2(n))) \).

\(\square\)

Let us now apply these two theorems to obtain a simpler analysis of the running time of \( \text{MaxSumBU} \). Recall that in our original analysis, we concluded that the running time of a single iteration of the loop is bounded by a fixed constant. We can therefore conclude that the running time of a single iteration is in \( O(1) \). Because there are \( n \) iterations, the running time for the entire loop is bounded by the product of \( n \) and the running time of a single iteration. By Theorem 3.8 part 1, the running time of the loop is in \( O(n) \). Clearly, the running times of the code segments before and after
the loop are each in \(O(1)\). The total running time is then the sum of the running times of these segments and that of the loop. By applying Theorem 3.8 part 2 twice, we see that the running time of the algorithm is in \(O(n)\) (because \(\max(1, n) \leq n\) whenever \(n \geq 1\)).

Recall that the actual running time of the program implementing Max-SumOpt (Figure 1.11 on page 15) was much slower than that of Max-SumBU. Let us now analyze MaxSumOpt to see why this is the case.

We will begin with the inner loop. It is easily seen that each iteration runs in \(O(1)\) time. The number of iterations of this loop varies from 1 to \(n\). Because the number of iterations is in \(O(n)\), we can conclude that this loop runs in \(O(n)\) time. It is then easily seen that a single iteration of the outer loop runs in \(O(n)\) time. Because the outer loop iterates \(n\) times, this loop, and hence the entire algorithm, runs in \(O(n^2)\) time.

It is tempting to conclude that this analysis explains the difference in running times of the implementations of the algorithms; i.e., because \(n^2\) grows much more rapidly than does \(n\), MaxSumOpt is therefore much slower than MaxSumBU. However, this conclusion is not yet warranted, because we have only shown upper bounds on the running times of the two algorithms. In particular, it is perfectly valid to conclude that the running time of MaxSumBU is in \(O(n^2)\), because \(O(n) \subseteq O(n^2)\). Conversely, we have not shown that the running time of MaxSumOpt is not in \(O(n)\).

In general, big-O notation is useful for expressing upper bounds on the growth rates of functions. In order to get a complete analysis, however, we need additional notation for expressing lower bounds.

### 3.3 Big-\(\Omega\) and Big-\(\Theta\)

**Definition 3.9** Let \(f : \mathbb{N} \to \mathbb{R}^{\geq 0}\). \(\Omega(f(n))\) is defined to be the set of all functions \(g : \mathbb{N} \to \mathbb{R}^{\geq 0}\) such that for some natural number \(n_0\) and some strictly positive real number \(c\), \(g(n) \geq cf(n)\) whenever \(n \geq n_0\).

Note that the definition of \(\Omega\) is identical to the definition of \(O\), except that the inequality, \(g(n) \leq cf(n)\), is replaced by the inequality, \(g(n) \geq cf(n)\). Thus, \(\Omega\) notation is used to express a lower bound in the same way that \(O\) notation is used to express an upper bound. Specifically, if \(g(n) \in \Omega(f(n))\), then for sufficiently large \(n\), \(g(n)\) is at least some constant multiple of \(f(n)\). This constant multiple is only required to be a positive real number, so it may be very close to 0.

**Example 3.10** Let \(f(n) = 3n + 10\) and \(g(n) = n^2\). We wish to show that...
CHAPTER 3. ANALYZING ALGORITHMS

$g(n) \in \Omega(f(n))$. We therefore need to find a positive real number $c$ and a natural number $n_0$ such that $n^2 \geq c(3n + 10)$ for every $n \geq n_0$. We have already found such values in Example 3.3: $c = 1/13$ and $n_0 = 1$.

The above example illustrates a duality between $O$ and $\Omega$, namely, that for any positive real number $c$, $g(n) \leq cf(n)$ iff $f(n) \geq g(n)/c$. The following theorem summarizes this duality.

**Theorem 3.11** Let $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Then $g(n) \in O(f(n))$ iff $f(n) \in \Omega(g(n))$.

By applying Theorem 3.11 to Examples 3.2, 3.4, 3.6, and 3.7, we can see that $n^2 \in \Omega(2n^2)$, $n^2 \notin \Omega(n^3)$, $\Omega(n^2) \subseteq \Omega(n)$, and $\Omega(n^2) = \Omega(4n^2 + 7n)$.

When we analyze the growth rate of a function $g$, we would ideally like to find a simple function $f$ such that $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$. Doing so would tell us that the growth rate of $g(n)$ is the same as that of $f(n)$, within a constant factor in either direction. We therefore have another notation for expressing such results.

**Definition 3.12** Let $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$. $\Theta(f(n))$ is defined to be $O(f(n)) \cap \Omega(f(n))$.

In other words, $\Theta(f(n))$ is the set of all functions belonging to both $O(f(n))$ and $\Omega(f(n))$ (see Figure 3.1). We can restate this definition by the following theorem, which characterizes $\Theta(f(n))$ in terms similar to the definitions of $O$ and $\Omega$.

**Theorem 3.13** $g(n) \in \Theta(f(n))$ iff there exist positive constants $c_1$ and $c_2$ and a natural number $n_0$ such that

$$c_1 f(n) \leq g(n) \leq c_2 f(n) \quad (3.3)$$

whenever $n \geq n_0$.

**Proof:** We must prove the implication in two directions.

$\Rightarrow$: Suppose $g(n) \in \Theta(f(n))$. Then $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$. By the definition of $\Omega$, there exist a positive real number $c_1$ and a natural number $n_1$ such that $c_1 f(n) \leq g(n)$ whenever $n \geq n_1$. By the definition of $O$, there exist a positive real number $c_2$ and a natural number $n_2$ such that...
Figure 3.1 Venn diagram depicting the relationships between the sets $O(f(n))$, $\Omega(f(n))$, and $\Theta(f(n))$

\[ g(n) \leq c_2 f(n) \] whenever \( n \geq n_2 \). Let \( n_0 = \max(n_1, n_2) \). Then (3.3) holds whenever \( n \geq n_0 \).

\[ \Leftarrow: \] Suppose (3.3) holds whenever \( n \geq n_0 \). From the first inequality, \( g(n) \in \Omega(f(n)) \). From the second inequality, \( g(n) \in O(f(n)) \). Therefore, \( g(n) \in \Theta(f(n)) \).

\[ \Rightarrow: \] Let us now use these definitions to continue the analysis of MaxSumBU. The analysis follows the same outline as the upper bound analysis; hence, we need the following theorem, whose proof is left as an exercise.

**Theorem 3.15** Suppose \( f_1(n) \in \Omega(g_1(n)) \) and \( f_2(n) \in \Omega(g_2(n)) \). Then
1. \( f_1(n) f_2(n) \in \Omega(g_1(n) g_2(n)) \); and
2. \( f_1(n) + f_2(n) \in \Omega(\max(g_1(n), g_2(n))) \).

By combining Theorems 3.8 and 3.15, we obtain the following corollary.

**Corollary 3.16** Suppose \( f_1(n) \in \Theta(g_1(n)) \) and \( f_2(n) \in \Theta(g_2(n)) \). Then
1. \( f_1(n) f_2(n) \in \Theta(g_1(n) g_2(n)) \); and
2. \( f_1(n) + f_2(n) \in \Theta(\max(g_1(n), g_2(n))) \).

We are now ready to proceed with our analysis of MaxSumBU. Clearly, the body of the loop must take some positive number of steps, so its running time is in \( \Omega(1) \). Furthermore, the loop iterates \( n \) times. We may therefore use Theorem 3.15 to conclude that the running time of the algorithm is in \( \Omega(n) \). Because we have already shown the running time to be in \( O(n) \), it therefore is in \( \Theta(n) \).

Let us now analyze the lower bound for MaxSumOpt. Again, the inner loop has a running time in \( \Omega(1) \). Its number of iterations ranges from 1 to \( n \), so the best lower bound we can give on the number of iterations is in \( \Omega(1) \). Using this lower bound, we conclude that the running time of the inner loop is in \( \Omega(1) \). Because the outer loop iterates \( n \) times, the running time of the algorithm is in \( \Omega(n) \).

Unfortunately, this lower bound does not match our upper bound of \( O(n^2) \). In some cases, we may not be able to make the upper and lower bounds match. In most cases, however, if we work hard enough, we can bring them together.

Clearly, the running time of a single iteration of the inner loop will require a constant number of steps in the worst case. Let \( a > 0 \) denote that constant. The loop iterates \( n - i \) times, so that the total number of steps required by the inner loop is \( (n - i) a \). An iteration of the outer loop requires a constant number of steps apart from the inner loop. Let \( b > 0 \) denote that constant. The loop iterates \( n \) times. However, because the number of steps required for the inner loop depends on the value of \( i \), which is different for each iteration of the outer loop, we must be more careful in computing the total number of steps required by the outer loop. That number is given by

\[
b + \sum_{i=0}^{n-1} (n - i)a = b + a \sum_{i=0}^{n-1} (n - i).
\]
The above summation can be simplified if we observe that the quantity \((n - i)\) takes on the values \(n, n - 1, \ldots, 1\). We can therefore rewrite the sum by taking the terms in the opposite order:

\[ 1 + 2 + \cdots + n = \sum_{i=1}^{n} i. \]

Thus, the number of steps required by the inner loop is

\[ b + a \sum_{i=1}^{n} i. \]

We can now use (2.1) from page 49 to conclude that the number of steps taken by the outer loop is

\[ b + \frac{an(n+1)}{2} \in \Theta(n^2). \]

Therefore, the running time of the algorithm is in \(\Theta(n^2)\).

This is a rather tedious analysis for such a simple algorithm. Fortunately, there are techniques for simplifying analyses. In the next two sections, we will present some of these techniques.

### 3.4 Operations on Sets

Asymptotic analysis can be simplified if we extend operations on functions to operations on sets of functions. Such an extension will allow us to streamline our notation without the need to introduce new constants or functions representing the running times of various code segments.

**Definition 3.17** Let \(\circ\) be a binary operation on functions of the form \(f : \mathbb{N} \rightarrow \mathbb{R}^\geq 0\) (for example, \(\circ\) might represent addition or multiplication). Let \(f\) be such a function, and let \(A\) and \(B\) be sets of such functions. We then define:

- \(f \circ A = \{ f \circ g \mid g \in A \} \);
- \(A \circ f = \{ g \circ f \mid g \in A \}\); and
- \(A \circ B = \{ g \circ h \mid g \in A, h \in B \}\).
Example 3.18 $n^2 + \Theta(n^3)$ is the set of all functions that can be written $n^2 + g(n)$ for some $g(n) \in \Theta(n^3)$. This set includes such functions as:

- $n^2 + 3n^3$;
- $(n^3 + 1)/2$, which can be written $n^2 + (n^3 + 1)/2 - n^2$ (note that $n^3 + 1/2 - n^2 \geq 0$ for all natural numbers $n$); and
- $n^3 + 2n$, which can be written $n^2 + (n^3 + 2n - n^2)$.

Because all functions in this set belong to $\Theta(n^3)$, $n^2 + \Theta(n^3) \subseteq \Theta(n^3)$.

Example 3.19 $O(n^2) + O(n^3)$ is the set of functions that can be written $f(n) + g(n)$, where $f(n) \in O(n^2)$ and $g(n) \in O(n^3)$. Functions in this set include:

- $2n^2 + 3n^3$;
- $2n$, which can be written as $n + n$; and
- $2n^3$, which can be written as $0 + 2n^3$.

Because all functions in this set belong to $O(n^3)$, $O(n^2) + O(n^3) \subseteq O(n^3)$.

Definition 3.20 Let $A$ be a set of functions of the form $f : \mathbb{N} \to \mathbb{R}^\geq 0$. We define

$$\sum_{i=k}^{n} A(i)$$

to be the set of all functions $g : \mathbb{N} \to \mathbb{R}^\geq 0$ such that

$$g(n) = \sum_{i=k}^{n} f(i)$$

for some $f \in A$. We define products analogously.

Example 3.21

$$\sum_{i=1}^{n} \Theta(i^2)$$

is the set of all functions of the form

$$\sum_{i=1}^{n} f(i)$$

such that $f(i) \in \Theta(i^2)$. 
Example 3.22 $f(n) \in f(n - 1) + \Theta(n)$ for $n \geq 1$. Here, we interpret “$f(n - 1) \ldots$ for $n \geq 1$” as shorthand for the following function:

$$g(n) = \begin{cases} f(n - 1) & \text{for } n \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

This is an example of an asymptotic recurrence. The meaning is that $f$ is a function satisfying a recurrence of the form

$$f(n) = \begin{cases} f(n - 1) + h(n) & \text{for } n \geq 1 \\ h(n) & \text{otherwise} \end{cases}$$

for some $h(n) \in \Theta(n)$. Note that because $h(0)$ may have any nonnegative value, so may $f(0)$.

We can use the above definitions to simplify our analysis of the lower bound for MAXSUMOPT. Instead of introducing the constant $a$ to represent the running time of a single iteration of the inner loop, we can simply use $\Omega(1)$ to represent the lower bound for this running time. We can therefore conclude that the total running time of the inner loop is in $\Omega(n - i)$. Using Definition 3.20, we can then express the running time of the outer loop, and hence, of the entire algorithm, as being in

$$\sum_{i=0}^{n-1} \Omega(n - i).$$

While this notation allows us to simplify the expression of bounds on running times, we still need a way of manipulating such expressions as the one above. In the next section, we present powerful tools for performing such manipulation.

### 3.5 Smooth Functions and Summations

Asymptotic analysis involving summations can be simplified by applying a rather general property of summations. This property relies on the fact that our summations typically involve well-behaved functions — functions that obey three important properties. The following definitions characterize these properties.

**Definition 3.23** Let $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$. $f$ is said to be **eventually nondecreasing** if there is a natural number $n_0$ such that $f(n) \leq f(n + 1)$ whenever $n \geq n_0$. 
Definition 3.24 Let $f : \mathbb{N} \rightarrow \mathbb{R}^\geq 0$. $f$ is said to be *eventually positive* if there is a natural number $n_0$ such that $f(n) > 0$ whenever $n \geq n_0$.

Definition 3.25 Let $f : \mathbb{N} \rightarrow \mathbb{R}^\geq 0$ be an eventually nondecreasing and eventually positive function. $f$ is said to be *smooth* if there exist a real number $c$ and a natural number $n_0$ such that $f(2n) \leq cf(n)$ whenever $n \geq n_0$.

Example 3.26 $f(n) = n$ is a smooth function. Clearly, $f$ is eventually nondecreasing and eventually positive, and $f(2n) = 2f(n)$ for all $n \in \mathbb{N}$.

Example 3.27 $f(n) = 2^n$ is not smooth. $f$ is eventually nondecreasing, and eventually positive, but $f(2n) = 2^{2n} = f^2(n)$ for all $n \in \mathbb{N}$. Because $f$ is unbounded, for any real $c$, $f(2n) > cf(n)$ for all sufficiently large $n$.

We will soon discuss in more detail which functions are smooth. First, however, let’s see why this notion is important. Suppose we want to give asymptotic bounds for a summation of the form

$$\sum_{i=1}^{g(n)} \Omega(f(i))$$

for some smooth function $f$. The following theorem, whose proof is outlined in Exercise 3.10, can then be applied.

Theorem 3.28 Let $f : \mathbb{N} \rightarrow \mathbb{R}^\geq 0$ be a smooth function, $g : \mathbb{N} \rightarrow \mathbb{N}$ be an eventually nondecreasing and unbounded function, and let $X$ denote either $O$, $\Omega$, or $\Theta$. Then

$$\sum_{i=1}^{g(n)} X(f(i)) \subseteq X(g(n)f(g(n))).$$

Thus, if we know that $f$ is smooth, we have an asymptotic solution to the summation. We therefore need to examine the property of smoothness more closely. The following theorem can be used to show a wide variety of functions to be smooth.

Theorem 3.29 Let $f : \mathbb{N} \rightarrow \mathbb{R}^\geq 0$ and $g : \mathbb{N} \rightarrow \mathbb{R}^\geq 0$ be smooth functions, and let $c \in \mathbb{R}^\geq 0$. Then the following functions are smooth:
• \( f(n) + g(n); \)
• \( f(n)g(n); \)
• \( f^c(n); \) and
• \( f(g(n)), \) provided \( g \) is unbounded.

The proof is left as an exercise. Knowing that \( f(n) = n \) is smooth, we can apply Theorem 3.29 to conclude that any polynomial is smooth. In fact, such functions as \( \sqrt{n} \) and \( n^{\sqrt{2}} \) are also smooth. We can extend this idea to logarithms as well. In particular, let \( \lg x \) denote the base-2 logarithm; i.e.,

\[
2^{\lg x} = x
\]

for all positive \( x \). Strictly speaking, \( \lg \) is not a function of the form \( f : \mathbb{N} \to \mathbb{R}_{\geq 0} \), because \( \lg 0 \) is undefined. However, whenever we have a function that maps all but finitely many natural numbers to nonnegative real numbers, we simply “patch” the function by defining it to be 0 at all other points. This is safe to do when we are performing asymptotic analysis because for sufficiently large \( n \), the patched function matches the original function.

**Example 3.30** \( \lg n \) is smooth. Clearly \( \lg n \) is eventually nondecreasing and eventually positive. Furthermore, \( \lg(2n) = 1 + \lg n \leq 2 \lg n \) whenever \( n \geq 2 \).

Thus far, the only example we have seen of a non-smooth function is \( 2^n \). Indeed, almost any polynomial-bounded, eventually nondecreasing, eventually positive function we encounter will turn out to be smooth. However, we can contrive exceptions. For example, we leave it as an exercise to show that \( 2^{\lfloor \lg \lg n \rfloor} \in O(n) \), but is not smooth.

We can now continue the analysis of the lower bound for \( \text{MAXSUMOPT} \). As we showed in the previous section, this lower bound is in

\[
\sum_{i=0}^{n-1} \Omega(n - i).
\]

Unfortunately, Theorem 3.28 does not immediately apply to this summation. First, the lower limit of the index \( i \) is 0, not 1 as required by Theorem 3.28. Furthermore, the theorem requires the expression inside the asymptotic notation to be a function of the summation index \( i \), not of \( n - i \).

On the other hand, we can take care of both of the above problems using the same technique that we used in our original analysis in Section
3.3. Specifically, we reverse the order of the summation to obtain
\[ \sum_{i=0}^{n-1} \Omega(n - i) = \sum_{i=1}^{n} \Omega(i). \]

Now the initial index of \( i \) is 1, and \( i \) is a smooth function. In order to apply Theorem 3.28, we observe that \( g(n) \) in the theorem corresponds to \( n \) in the above summation, and that \( f(i) \) in the theorem corresponds to \( i \) in the above summation. \( g(n)f(g(n)) \) is therefore just \( n^2 \). From Theorem 3.28 the running time of \( \text{MaxSumOpt} \) is in \( \Omega(n^2) \). Note that this is the same bound that we obtained in Section 3.3, but instead of using Equation 2.1, we used the more general (and hence, more widely applicable) Theorem 3.28.

To further illustrate the power of Theorem 3.28, let’s now analyze the running time of \( \text{MaxSumIter} \), given in Figure 1.10 on page 14. A single iteration of the inner loop has a running time in \( \Theta(1) \). This loop iterates \( j - i \) times, so its running time is in \( \Theta(j - i) \). The total running time of the middle loop is then in
\[ \sum_{j=i}^{n} \Theta(j - i). \]

Again, this summation does not immediately fit the form of Theorem 3.28, as the starting value of the summation index \( j \) is \( i \), not 1. Furthermore, \( j - i \) is not a function of \( j \). Notice that the expression \( j - i \) takes on the values \( 0, 1, \ldots, n - i \). We can therefore rewrite this sum as
\[ \sum_{j=i}^{n} \Theta(j - i) = \sum_{j=1}^{n-i+1} \Theta(j - 1). \]

What we have done here is simply to shift the range of \( j \) downward by \( i - 1 \) (i.e., from \( i, \ldots, n \) to \( 1, \ldots, n - i + 1 \)), and to compensate for this shift by adding \( i - 1 \) to each occurrence of \( j \) in the expression being summed.

Applying Theorem 3.28 to the above sum, we find that the running time of the middle loop is in \( \Theta((n - i + 1)(n - i)) = \Theta((n - i)^2) \). The running time of the outer loop is then
\[ \sum_{i=0}^{n} \Theta((n - i)^2). \]
CHAPTER 3. ANALYZING ALGORITHMS

The values in this summation are \( n^2, (n-1)^2, \ldots, 1 \); hence, we can rewrite this sum as

\[
\sum_{i=1}^{n+1} (i - 1)^2.
\]

Applying Theorem 3.28 to this sum, we find that the running time of this loop is in

\[
\Theta((n + 1)((n + 1) - 1)^2) = \Theta((n + 1)n^2)
\]

\[
= \Theta(n^3).
\]

The running time of the algorithm is therefore in \( \Theta(n^3) \).

3.6 Analyzing while Loops

To analyze algorithms with \texttt{while} loops, we can use the same techniques as we have used to analyze \texttt{for} loops. For example, consider \texttt{INSERTIONSORT}, shown in Figure 1.7 on page 11. Let us consider the \texttt{while} loop. The value of \( j \) begins at \( i \) and decreases by 1 on each loop iteration. Furthermore, if its value reaches 1, the loop terminates. The loop therefore iterates at most \( i-1 \) times. Because each iteration runs in \( \Theta(1) \) time, the \texttt{while} loop runs in \( O(i) \) time in the worst case.

In order to be able to conclude that the loop runs in \( \Theta(i) \) time in the worst case, we must determine that for arbitrarily large \( i \), the loop may iterate until \( j = 1 \). This is certainly the case if, prior to the beginning of the loop, \( A[i] \) is strictly less than every element in \( A[1..i-1] \). Thus, the \texttt{while} loop runs in \( \Theta(i) \) time in the worst case.

It is now tempting to use Theorem 3.28 to conclude that the entire algorithm’s running time is in

\[
\Theta(1) + \sum_{i=1}^{n} \Theta(i) \subseteq \Theta(1) + \Theta(n^2)
\]

\[
= \Theta(n^2).
\]

However, we must be careful, because we have not shown that the \texttt{while} loop runs in \( \Omega(i) \) time for \textit{every} iteration of the \texttt{for} loop; hence the running time of the \texttt{for} loop might not be in

\[
\sum_{i=1}^{n} \Theta(i).
\]
CHAPTER 3. ANALYZING ALGORITHMS

We must show that there are inputs of size $n$, for every sufficiently large $n$, such that the while loop iterates $i - 1$ times for each iteration of the for loop. It is not hard to show that an array of distinct elements in decreasing order will produce the desired behavior. Therefore, the algorithm indeed operates in $\Theta(n^2)$ time.

3.7 Analyzing Recursion

Before we consider how to analyze recursion, let us first consider how to analyze non-recursive function calls. For example, consider SimpleSelect from Figure 1.2 on page 6. This algorithm is easy to analyze if we know the running time of Sort. Suppose we use InsertionSort (Figure 1.7, page 11). We saw in the last section that InsertionSort runs in $\Theta(n^2)$ time. The running time of SimpleSelect is therefore in $\Theta(1) + \Theta(n^2) \subseteq \Theta(n^2)$.

Suppose now that we wish to analyze an algorithm that makes one or more recursive calls. For example, consider MaxSuffixTD from Figure 1.13 on page 17. We analyze such an algorithm in exactly the same way. Specifically, this algorithm has a running time in $\Theta(1)$ plus whatever is required by the recursive call. The difficulty here is in how to determine the running time of the recursive call without knowing the running time of the algorithm.

The solution to this difficulty is to express the running time as a recurrence. Specifically, let $f(n)$ denote the worst-case running time of MaxSuffixTD on an array of size $n$. Then for $n > 0$, we have the equation,

$$f(n) = g(n) + f(n - 1) \quad (3.5)$$

where $g(n) \in \Theta(1)$ is the worst-case running time of the body of the function, excluding the recursive call. Note that $f(n - 1)$ has already been defined to be the worst-case running time of MaxSuffixTD on an array of size $n - 1$; hence, $f(n - 1)$ gives the worst-case running time of the recursive call.

The solution of arbitrary recurrences is beyond the scope of this book. However, asymptotic solutions are often much simpler to obtain than are exact solutions. First, we observe that (3.5) can be simplified using set operations:

$$f(n) \in f(n - 1) + \Theta(1) \quad (3.6)$$
for $n > 0$.

It turns out that most of the recurrences that we derive when analyzing algorithms fit into a few general forms. With asymptotic solutions to these general forms, we can analyze recursive algorithms without using a great deal of detailed mathematics. (3.6) fits one of the most basic of these forms. The following theorem, whose proof is outlined in Exercise 3.20, gives the asymptotic solution to this form.

**Theorem 3.31** Let

$$f(n) \in af(n-1) + X(b^n g(n))$$

for $n > n_0$, where $n_0 \in \mathbb{N}$, $a \geq 1$ and $b \geq 1$ are real numbers, $g(n)$ is a smooth function, and $X$ is either $O$, $\Omega$, or $\Theta$. Then

$$f(n) \in \begin{cases} 
X(b^n g(n)) & \text{if } a < b \\
X(na^n g(n)) & \text{if } a = b \\
X(a^n) & \text{if } a > b.
\end{cases}$$

When we apply this theorem to the analysis of algorithms, $a$ in the recurrence denotes the number of recursive calls. The set $X(b^n g(n))$ contains the function giving the running time of the algorithm, excluding recursive calls. Note that the expression $b^n g(n)$ is general enough to describe a wide variety of functions. However, the main restriction on the applicability of this theorem is that $f(n)$ is in terms of $f(n-1)$, so that it applies only to those algorithms whose recursive calls reduce the size of the problem by 1.

Let us now see how Theorem 3.31 can be applied to the analysis of MAXSUFFIXTD. (3.6) fits the form given in Theorem 3.31, where $a = 1$, $b = 1$, $g(n) = 1$, and $X = \Theta$. Therefore, the second case of Theorem 3.31 applies. Substituting the values for $X, a$, and $g(n)$ in that solution, we obtain $f(n) \in \Theta(n)$.

Knowing that MAXSUFFIXTD operates in $\Theta(n)$ time, we can now analyze MAXSUMTD in the same way. In this case, the time required excluding the recursive call is in $\Theta(n)$, because a call to MAXSUFFIXTD is made. Letting $f(n)$ denote the running time for MAXSUMTD on an array of size $n$, we see that

$$f(n) \in f(n-1) + \Theta(n)$$

for $n > 0$. Again, this recurrence fits the form of Theorem 3.31 with $a = 1$, $b = 1$, $g(n) = n$, and $X = \Theta$. The second case again holds, so that the running time is in $\Theta(n^2)$. 
CHAPTER 3. ANALYZING ALGORITHMS

Figure 3.2 When applying divide-and-conquer, the maximum subsequence sum may not lie entirely in either half.

It is no coincidence that both of these analyses fit the second case of Theorem 3.31. Note that unless \(a\) and \(b\) are both 1, Theorem 3.31 yields an exponential result. Thus, efficient algorithms will always fit the second case if this theorem applies. As a result, we can observe that an algorithm that makes more than one recursive call of size \(n - 1\) will yield an exponential-time algorithm.

We have included the first and third cases in Theorem 3.31 because they are useful in deriving a solution for certain other types of recurrences. To illustrate how these recurrences arise, we consider another solution to the maximum subsequence sum problem (see Section 1.6).

The technique we will use is called divide-and-conquer. This technique, which we will examine in detail in Chapter 10, involves reducing the size of recursive calls to a fixed fraction of the size of the original call. For example, we may attempt to make recursive calls on arrays of half the original size.

We therefore begin this solution by dividing a large array in half, as nearly as possible. The subsequence giving us the maximum sum can then lie in one of three places: entirely in the first half, entirely in the second half, or partially in both halves, as shown in Figure 3.2. We can find the maximum subsequence sum of each half by solving the two smaller problem instances recursively. If we can then find the maximum sum of any sequence that begins in the first half and ends in the second half, then the maximum of these three values is the overall maximum subsequence sum.

For example, consider again the array \(A[0..5] = (-1, 3, -2, 7, -9, 7)\) from Example 1.1 (page 13). The maximum subsequence sum of the first half, namely, of \(A[0..2] = (-1, 3, -2)\), has a value of 3. Likewise, the maximum subsequence sum of the second half, \((7, -9, 7)\), is 7. In examining the two halves, we have missed the actual maximum, \(A[1..3] = (3, -2, 7)\), which
resides in neither half. However, notice that such a sequence that resides in
neither half can be expressed as a suffix of the first half followed by a prefix
of the last half; e.g., $\langle 3, -2, 7 \rangle$ can be expressed as $\langle 3, -2 \rangle$ followed by $\langle 7 \rangle$.

Let us define the maximum prefix sum analogously to the maximum
suffix sum as follows:

$$\max \left\{ \sum_{k=0}^{i-1} A[k] \mid 0 \leq i \leq n \right\}.$$ 

It is not hard to see that the maximum sum of any sequence crossing the
boundary is simply the maximum suffix sum of the first half plus the max-
imum prefix sum of the second half. For example, returning to Example
1.1, the maximum suffix sum of the first half is 1, obtained from the suffix
$\langle 3, -2 \rangle$. Likewise, the maximum prefix sum of the second half is 7, obtained
from the prefix $\langle 7 \rangle$. The sum of these two values gives us 8, the maximum
subsequence sum.

Note that when we create smaller instances by splitting the array in half,
one of the two smaller instances — the upper half — does not begin with
index 0. For this reason, let us describe the input array more generally, as
$A[lo..hi]$. We can then modify the definitions of maximum subsequence sum,
maximum suffix sum, and maximum prefix sum by replacing 0 with $lo$ and
$n - 1$ with $hi$. We will discuss the ranges of $lo$ and $hi$ shortly.

We must be careful that each recursive call is of a strictly smaller size.
We wish to divide the array in half, as nearly as possible. We begin by
finding the midpoint between $lo$ and $hi$; i.e,

$$mid = \left\lfloor \frac{lo + hi}{2} \right\rfloor.$$ 

Note that if $hi > lo$, then $lo \leq mid < hi$. In this case, we can split
$A[lo..hi]$ into $A[lo..mid]$ and $A[mid+1..hi]$, and both sub-arrays are smaller
than the original. However, a problem occurs when $lo = hi$ — i.e., when the
array contains only one element — because in this case $mid = hi$. In fact,
it is impossible to divide an array of size 1 into two subarrays, each smaller
than the original. Fortunately, it is easy to solve a one-element instance
directly. Furthermore, it now makes sense to consider an empty array as a
special case, because it can only occur when we begin with an empty array,
and not as a result of dividing a nonempty array in half. We will therefore
require in our precondition that $lo \leq hi$, and that both are natural numbers.

We can compute the maximum suffix sum as in MAXSMBU (see Figure
1.14 on page 18), and the maximum prefix sum in a similar way. The entire
algorithm is shown in Figure 3.3. Note that the specification has been changed from the one given in Figure 1.9. However, it is a trivial matter to give an algorithm that takes as input $A[0..n-1]$ and calls $\text{MaxSumDC}$ if $n > 0$, or returns 0 if $n = 0$. Such an algorithm would satisfy the specification given in Figure 1.9.

This algorithm contains two recursive calls on arrays of size $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$, respectively. In addition, it calls $\text{MaxSuffixBU}$ on an array of size $\lfloor \frac{n}{2} \rfloor$ and $\text{MaxPrefixBU}$ on an array of size $\lceil \frac{n}{2} \rceil$. These two algorithms are easily seen to have running times in $\Theta(n)$; hence, if $f(n)$ denotes the worst-case running time of $\text{MaxSumDC}$ on an array of size $n$, we have

$$f(n) \in f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + \Theta(n) \quad (3.7)$$

for $n > 1$.

This equation does not fit the form of Theorem 3.31. However, suppose we focus only on those values of $n$ that are powers of 2; i.e., let $n = 2^k$ for some $k > 0$, and let $g(k) = f(2^k) = f(n)$. Then

$$g(k) = f(2^k)$$

$$\in 2f(2^{k-1}) + \Theta(2^k)$$

$$= 2g(k-1) + \Theta(2^k) \quad (3.8)$$

for $k > 0$. Theorem 3.31 applies to (3.8), yielding $g(k) \in \Theta(k2^k)$. Because $n = 2^k$, we have $k = \log n$, so that

$$f(n) = g(k) = g(\log n). \quad (3.9)$$

It is now tempting to conclude that because $g(\log n) \in \Theta(n \log n)$, $f(n) \in \Theta(n \log n)$; however, (3.9) is valid only when $n$ is a power of 2. In order to conclude that $f(n) \in \Theta(n \log n)$, we must know something about $f(n)$ for every sufficiently large $n$. However, we can show by induction on $n$ that $f(n)$ is eventually nondecreasing (the proof is left as an exercise). This tells us that for sufficiently large $n$, when $2^k \leq n \leq 2^{k+1}$, $f(2^k) \leq f(n) \leq f(2^{k+1})$.

From the fact that $f(2^k) = g(k) \in \Theta(k2^k)$, there exist positive real numbers $c_1$ and $c_2$ such that $c_1 k 2^k \leq f(n) \leq c_2 (k + 1) 2^{k+1}$. Furthermore, because $n \log n$ is smooth, there is a positive real number $d$ such that for sufficiently large $m$, $2m \log(2m) \leq dm \log m$. Hence, substituting $2^k$ for $m$, we have
Figure 3.3 Divide-and-conquer algorithm for maximum subsequence sum, specified in Figure 1.9

Precondition: \(A[lo..hi]\) is an array of Numbers, \(lo \leq hi\), and both \(lo\) and \(hi\) are Nats.

Postcondition: Returns the maximum subsequence sum of \(A[lo..hi]\).

\[
\text{MaxSumDC}(A[lo..hi])
\]
if \(lo = hi\)
   \[\text{return} \; \text{Max}(0, A[lo])\]
else
   \[\text{mid} \leftarrow \lfloor (lo + hi)/2 \rfloor; \; \text{mid1} \leftarrow \text{mid} + 1;\]
   \[\text{sum1} \leftarrow \text{MaxSumDC}(A[lo..\text{mid}])\]
   \[\text{sum2} \leftarrow \text{MaxSumDC}(A[\text{mid1}..hi])\]
   \[\text{sum3} \leftarrow \text{MaxSuffixBU}(A[lo..\text{mid}]) + \text{MaxPrefixBU}(A[\text{mid1}..hi])\]
   \[\text{return} \; \text{Max}(\text{sum1}, \text{sum2}, \text{sum3})\]

Precondition: \(A[lo..hi]\) is an array of Numbers, \(lo \leq hi\), and both \(lo\) and \(hi\) are Nats.

Postcondition: Returns the maximum suffix sum of \(A[lo..hi]\).

\[
\text{MaxSuffixBU}(A[lo..hi])
\]
\[m \leftarrow 0\]
   // Invariant: \(m\) is the maximum suffix sum of \(A[lo..i-1]\)
   \[\text{for } i \leftarrow lo \text{ to } hi\]
      \[m \leftarrow \text{Max}(0, m + A[i])\]
   \[\text{return } m\]

Precondition: \(A[lo..hi]\) is an array of Numbers, \(lo \leq hi\), and both \(lo\) and \(hi\) are Nats.

Postcondition: Returns the maximum prefix sum of \(A[lo..hi]\).

\[
\text{MaxPrefixBU}(A[lo..hi])
\]
\[m \leftarrow 0\]
   // Invariant: \(m\) is the maximum prefix sum of \(A[i+1..hi]\)
   \[\text{for } i \leftarrow hi \text{ to } lo \text{ by } -1\]
      \[m \leftarrow \text{Max}(0, m + A[i])\]
   \[\text{return } m\]
$2^{k+1}(k+1) \leq d2^k$. Putting it all together, we have

$$f(n) \leq c_2(k+1)2^{k+1}$$
$$\leq c_2dk2^k$$
$$\leq c_2dn \lg n$$
$$\in O(n \lg n).$$

Likewise,

$$f(n) \geq c_1k2^k$$
$$\geq \frac{c_1(k+1)2^{k+1}}{d}$$
$$\geq \frac{c_1n \lg n}{d}$$
$$\in \Omega(n \lg n).$$

Thus, $f(n) \in \Theta(n \lg n)$. The running time of MaxSumDC is therefore slightly worse than that of MaxSumBU.

The above technique is often useful when we have a recurrence which is not of a form for which we have a solution. More importantly, however, we can generalize this technique to prove the following theorem; the details are left as an exercise.

**Theorem 3.32** Let $a \geq 1$ and $q \geq 0$ be real numbers, and let $n_0 \geq 1$ and $b \geq 2$ be integers. Let $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be such that $g'(n) = g(n_0 b^n)$ is smooth. Finally, let $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be an eventually nondecreasing function satisfying

$$f(n) \in af(n/b) + X(n^q g(n))$$

whenever $n = n_0 b^k$ for a positive integer $k$, where $X$ is either $O$, $\Omega$, or $\Theta$. Then

$$f(n) \in \begin{cases} 
X(n^q g(n)) & \text{if } a < b^q \\
X(n^q g(n) \lg n) & \text{if } a = b^q \\
X(n^{\log_b a}) & \text{if } a > b^q.
\end{cases}$$

Let us first see that (3.7) fits the form of Theorem 3.32. As we have already observed, $f$ is eventually nondecreasing (this requirement is typically met by recurrences obtained in the analysis of algorithms). When $n = 2^k$, (3.7) simplifies to

$$f(n) \in 2f(n/2) + \Theta(n).$$
Therefore, we can let \( n_0 = 1, a = b = 2, q = 1, \) and \( g(n) = 1. \) This yields \( g'(n) = g(2^n) = 1, \) which is smooth. Therefore, the second case applies, yielding \( f(n) \in \Theta(n \lg n). \)

An important prerequisite for applying Theorem 3.32 is that \( g(n_0 b^n) \) is smooth. Due to the exponential term, any function that satisfies this property must be in \( O(\lg^k n) \) for some fixed \( k. \) This is not really a restriction, however, because the term expressing the non-recursive part of the analysis may be in \( X(n^q \lg n) \) for arbitrary real \( q \geq 0; \) hence, we can express most polynomially-bounded functions. What is important is that we separate this function into a polynomial part and a “polylogarithmic” part, because the degree of the polynomial affects the result.

**Example 3.33** Let \( f : \mathbb{N} \to \mathbb{R}^\geq 0 \) be an eventually nondecreasing function such that

\[
f(n) = 3f(n/2) + \Theta(n^2 \lg n)
\]

whenever \( n = 2^k \) for a positive integer \( k. \) We then let \( n_0 = 1, a = 3, b = 2, q = 2, \) and \( g(n) = \lg n. \) Then

\[
g'(n) = g(2^n) = \lg 2^n = n
\]

is smooth. We can therefore apply Theorem 3.32. Because \( b^q = 2^2 = 4 \) and \( a = 3, \) the first case applies. Therefore, \( f(n) \in \Theta(n \lg n). \)

**Example 3.34** Let \( f : \mathbb{N} \to \mathbb{R}^\geq 0 \) be an eventually nondecreasing function such that

\[
f(n) = 4f(n/3) + O(n \lg^2 n)
\]

whenever \( n = 5 \cdot 3^k \) for a positive integer \( k. \) We then let \( n_0 = 5, a = 4, b = 3, q = 1, \) and \( g(n) = \lg^2 n. \) Then

\[
g'(n) = g(5 \cdot 3^n) = \lg^2 (5 \cdot 3^n) = \lg^2 5 + n^2 \lg^2 3
\]

is smooth. We can therefore apply Theorem 3.32. Because \( b^q = 3 \) and \( a = 4, \) the third case applies. Therefore, \( f(n) \in O(n^{\log_3 4}) \) (\( \log_3 4 \) is approximately 1.26).
3.8 Analyzing Space Usage

As we mentioned earlier, running time is not the only performance measure we may be interested in obtaining. For example, recall that the implementation of MaxSumTD from Figure 1.13 on page 17 terminated with a StackOverflowError on an input of 4096 elements. As we explained in Section 1.6, this error was caused by high stack usage due to the recursion. In contrast, the implementation of MaxSumDC can handle an input of several million elements, even though it, too, is recursive. In order to see why, we can analyze the space usage of these algorithms using the techniques we have already developed.

Let us first consider MaxSuffixTD from Figure 1.13. Because there is no need to copy the array in order to perform the recursive call, this algorithm requires only a constant amount of space, ignoring that needed by the recursive call. (We typically do not count the space occupied by the input or the output in measuring the space usage of an algorithm.) Thus, the total space usage is given by

\[ f(n) \in f(n-1) + \Theta(1) \] (3.10)

for \( n > 0 \). From Theorem 3.31, \( f(n) \in \Theta(n) \).

Already this is enough to tell us why MaxSumTD has poor space performance. If MaxSuffixTD requires \( \Theta(n) \) space, then MaxSumTD surely must require \( \Omega(n) \) space. Furthermore, it is easily seen from the above analysis that (3.10) the space used is almost entirely from the runtime stack; hence, the stack usage is in \( \Theta(n) \). We typically would not have a runtime stack capable of occupying space proportional to an input of 100,000 elements.

Let us now complete the analysis of MaxSumTD. Ignoring the space usage of the recursive call, we see that MaxSumTD uses \( \Theta(n) \) space, due to the space usage of MaxSuffixTD. However, this does not mean that the following recurrence describes the total space usage:

\[ f(n) \in f(n-1) + \Theta(n) \]

for \( n > 0 \). The reason is that the call made to MaxSuffixTD can reuse the space used by the recursive call. Furthermore, any calls made to MaxSuffixTD as a result of the recursive call will be on arrays of fewer than \( n \) elements, so they may reuse the space used by MaxSuffixTD\( (A[0..n-1]) \). Therefore, the total space used by all calls to MaxSuffixTD is in \( \Theta(n) \). Ignoring this space, the space used by MaxSumTD is given by

\[ f(n) \in f(n-1) + \Theta(1) \]
Figure 3.4 An algorithm to add two matrices.

**Precondition:** $A[1..m, 1..n]$ and $B[1..m, 1..n]$ are arrays of Numbers, and $m$ and $n$ are positive NatS.

**Postcondition:** Returns the sum of $A[1..m, 1..n]$ and $B[1..m, 1..n]$; i.e., returns the array $C[1..m, 1..n]$ such that for $1 \leq i \leq m$ and $1 \leq j \leq n$, $C[i, j] = A[i, j] + B[i, j]$.

\[
\text{AddMatrices}(A[1..m, 1..n], B[1..m, 1..n])
\]

\[
C \leftarrow \text{new ARRAY}[1..m, 1..n]
\]

\[
\text{for } i \leftarrow 1 \text{ to } m
\]

\[
\text{for } j \leftarrow 1 \text{ to } n
\]

\[
C[i, j] \leftarrow A[i, j] + B[i, j]
\]

\[
\text{return } C[1..m, 1..n]
\]

for $n > 0$, so that $f(n) \in \Theta(n)$. The total space used is therefore in $\Theta(n) + \Theta(n) = \Theta(n)$.

Now let’s consider MaxSumDC. MaxSuffixBU and MaxPrefixBU each use $\Theta(1)$ space. Because the two recursive calls can reuse the same space, the total space usage is given by

\[
f(n) \in f(\lceil n/2 \rceil) + \Theta(1)
\]

for $n > 1$. Applying Theorem 3.32, we see that $f(n) \in \Theta(\lg n)$. Because $\lg n$ is a slow-growing function (e.g., $\lg 10^6 < 20$), we can see that MaxSumDC is a much more space-efficient algorithm than MaxSumTD. Because the space used by both algorithms is almost entirely from the runtime stack, MaxSumDC will not have the stack problems that MaxSumTD has.

### 3.9 Multiple Variables

Consider the algorithm AddMatrices shown in Figure 3.4. Applying the techniques we have developed so far, we can easily see that the inner loop runs in $\Theta(n)$ time. Furthermore, the outer loop iterates exactly $m$ times, so that the nested loops run in $m\Theta(n)$ time. It is tempting at this point to simplify this bound to $\Theta(mn)$ using Theorem 3.8; however, we must be careful here because we have defined asymptotic notation for single-variable...
Figure 3.5 An algorithm illustrating difficulties with asymptotic notation
with multiple variables

Precondition: $m$ and $n$ are Nats.
Postcondition: true.

\[
F(m, n)
\]
for $i \leftarrow 0$ to $m - 1$
if $i = 0$
for $j \leftarrow 1$ to $2^n$
   // Do nothing
else
for $j \leftarrow 1$ to $in$
   // Do nothing

functions only. In this section, we discuss how to apply asymptotic analysis
to functions on more than one variable.

We would like to extend the definitions to multiple variables in as straight-
forward a manner as possible. For example, we would like for $O(f(m, n))$
to include all functions $g : \mathbb{N} \times \mathbb{N} \to \mathbb{R}_{\geq 0}$ such that for some $c \in \mathbb{R}^>0$ and
$n_0 \in \mathbb{N}$, $g(m, n) \leq cf(m, n)$ whenever certain conditions hold. The question
is what exactly these “certain conditions” should be. Should the inequality
be required to hold whenever at least one of $m$ or $n$ is at least $n_0$? Or should
it be required to hold only when both $m$ and $n$ are at least $n_0$?

Suppose first that we were to require that the inequality hold whenever
at least one of $m$ or $n$ is at least $n_0$. Unfortunately, a consequence of such a
definition would be that $mn + 1 \not\in O(mn)$. To see why, observe that whatever
values we choose for $c$ and $n_0$, when $m = 0$ and $n \geq n_0$, $mn + 1 > cmn$.
As a result, working with asymptotic notation would become much messier
with multiple variables than with a single variable.

On the other hand, only requiring the inequality to hold when both $m$
and $n$ are at least $n_0$ also presents problems. Consider, for example, how we
would analyze the rather silly algorithm shown in Figure 3.5. We can observe
that the first of the inner loops iterates $2^n$ times, and that the second iterates
$in$ times. However, the first loop is only executed when $i = 0$; hence, when
both $i$ and $n$ are sufficiently large, only the second loop is executed. We
could therefore legitimately conclude that the body of the outer loop runs
in $O(in)$ time. Unfortunately, this would lead to an incorrect analysis of the
algorithm because the first inner loop will always execute once, assuming \( m \) is a \( \text{Nat} \).

Thus, we can see that if we want to retain the properties of asymptotic notation on a single variable, we must extend it to multiple variables in a way that is not straightforward. Unfortunately, the situation is worse than this — it can be shown that it is impossible to extend the notation to multiple variables in a way that retains the properties of asymptotic notation on a single variable. What we can do, however, is to extend it so that these properties are retained whenever the function inside the asymptotic notation is strictly nondecreasing. Note that restricting the functions in this way does not avoid the problems discussed above, as the functions inside the asymptotic notation in this discussion are all strictly nondecreasing. We therefore must use some less straightforward extension.

The definition we propose for \( O(f(m, n)) \) considers all values of a function \( g(m, n) \), rather than ignoring values when \( m \) and/or \( n \) are small. However, it allows even infinitely many values of \( g(m, n) \) to be large in comparison to \( f(m, n) \), provided that they are not too large in comparison to the overall growth rate of \( f \). In order to accomplish these goals, we first give the following definition.

**Definition 3.35** For a function \( f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \), we define \( \hat{f} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \) so that

\[
\hat{f}(m, n) = \max \{ f(i, j) \mid 0 \leq i \leq m, 0 \leq j \leq n \}.
\]

Using the above definition, we can now define big-\( O \) for 2-variable functions.

**Definition 3.36** For a function \( f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \), we define \( O(f(m, n)) \) to be the set of all functions \( g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \) such that there exist \( c \in \mathbb{R}^{>0} \) and \( n_0 \in \mathbb{N} \) so that

\[
g(m, n) \leq cf(m, n)
\]

and

\[
\hat{g}(m, n) \leq c\hat{f}(m, n)
\]

whenever \( m \geq n_0 \) and \( n \geq n_0 \).

Likewise, we can define big-\( \Omega \) and big-\( \Theta \) for 2-variable functions.

**Definition 3.37** For a function \( f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \), we define \( \Omega(f(m, n)) \) to be the set of all functions \( g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \) such that there exist \( c \in \mathbb{R}^{>0} \) and \( n_0 \in \mathbb{N} \) so that

\[
g(m, n) \geq cf(m, n)
\]
CHAPTER 3. ANALYZING ALGORITHMS

and
\[ \hat{g}(m, n) \geq c \hat{f}(m, n) \]
whenever \( m \geq n_0 \) and \( n \geq n_0 \).

Definition 3.38 For a function \( f : \mathbb{N} \times \mathbb{N} \to \mathbb{R}_{\geq 0} \),
\[ \Theta(f(m, n)) = O(f(m, n)) \cap \Omega(f(m, n)). \]

We extend these definitions to more than two variables in the obvious way. Using the above definitions, it is an easy matter to show that Theorem 3.11 extends to more than one variable. The proof is left as an exercise.

Theorem 3.39 Let \( f : \mathbb{N} \times \mathbb{N} \to \mathbb{R}_{\geq 0} \) and \( g : \mathbb{N} \times \mathbb{N} \to \mathbb{R}_{\geq 0} \). Then \( g(m, n) \in O(f(m, n)) \) iff \( f(m, n) \in \Omega(g(m, n)) \).

We would now like to show that the theorems we have presented for single variables extend to multiple variables, provided the functions within the asymptotic notation are strictly nondecreasing. Before we do this, however, we will first prove a theorem that will allow us to simplify the proofs of the individual properties.

Theorem 3.40 Let \( f : \mathbb{N} \times \mathbb{N} \to \mathbb{R}_{\geq 0} \) be a strictly nondecreasing function. Then

1. \( O(f(m, n)) \) is the set of all functions \( g : \mathbb{N} \times \mathbb{N} \to \mathbb{R}_{\geq 0} \) such that there exist \( c \in \mathbb{R}_{> 0} \) and \( n_0 \in \mathbb{N} \) such that
\[ \hat{g}(m, n) \leq c \hat{f}(m, n) \]
whenever \( m \geq n_0 \) and \( n \geq n_0 \).

2. \( \Omega(f(m, n)) \) is the set of all functions \( g : \mathbb{N} \times \mathbb{N} \to \mathbb{R}_{\geq 0} \) such that there exist \( c \in \mathbb{R}_{> 0} \) and \( n_0 \in \mathbb{N} \) such that
\[ g(m, n) \geq cf(m, n) \]
whenever \( m \geq n_0 \) and \( n \geq n_0 \).

Proof: From the definitions, for any function \( g(m, n) \) in \( O(f(m, n)) \) or in \( \Omega(f(m, n)) \), respectively, there are a \( c \in \mathbb{R}_{> 0} \) and an \( n_0 \in \mathbb{N} \) such that whenever \( m \geq n_0 \) and \( n \geq n_0 \), the corresponding inequality above is satisfied. We therefore only need to show that if there are \( c \in \mathbb{R}_{> 0} \) and \( n_0 \in \mathbb{N} \) such
CHAPTER 3. ANALYZING ALGORITHMS

that whenever \( m \geq n_0 \) and \( n \geq n_0 \), the given inequality is satisfied, then \( g(m, n) \) belongs to \( O(f(m, n)) \) or \( \Omega(f(m, n)) \), respectively.

We first observe that if \( f \) is strictly nondecreasing, then

\[
\hat{f}(m, n) = f(m, n)
\]

for all natural numbers \( m \) and \( n \). Furthermore, for any function \( g : \mathbb{N} \times \mathbb{N} \to \mathbb{R}_{\geq 0} \),

\[
\hat{g}(m, n) \geq g(m, n).
\]

Now suppose \( c \in \mathbb{R}^>0 \) and \( n_0 \in \mathbb{N} \) such that whenever \( m \geq n_0 \) and \( n \geq n_0 \), \( \hat{g}(m, n) \leq cf(m, n) \). Then for \( m \geq n_0 \) and \( n \geq n_0 \),

\[
g(m, n) \leq \hat{g}(m, n) \leq cf(m, n) = cf(m, n).
\]

Hence, \( g(m, n) \in O(f(m, n)) \).

Likewise, suppose now that \( c \in \mathbb{R}^>0 \) and \( n_0 \in \mathbb{N} \) such that whenever \( m \geq n_0 \) and \( n \geq n_0 \), \( g(m, n) \leq cf(m, n) \). Then for \( m \geq n_0 \) and \( n \geq n_0 \),

\[
\hat{g}(m, n) \geq g(m, n) \geq cf(m, n) = cf(m, n).
\]

Therefore, \( g(m, n) \in \Omega(f(m, n)) \). \( \square \)

As a result of the above theorem, in order to prove properties about either \( O(f(m, n)) \) or \( \Omega(f(m, n)) \), where \( f \) is strictly nondecreasing, we only need to prove one of the two inequalities in the definition. Consider, for example, the following extension to Theorem 3.8.

**Theorem 3.41** Suppose \( f_1(m, n) \in O(g_1(m, n)) \) and \( f_2(m, n) \in O(g_2(m, n)) \), where \( g_1 \) and \( g_2 \) are strictly nondecreasing. Then

1. \( f_1(m, n)f_2(m, n) \in O(g_1(m, n)g_2(m, n)) \); and
2. \( f_1(m, n) + f_2(m, n) \in O(\max(g_1(m, n), g_2(m, n))) \).

**Proof:** We will only show part 1; part 2 will be left as an exercise. Because \( f_1(m, n) \in O(g_1(m, n)) \) and \( f_2(m, n) \in O(g_2(m, n)) \), there exist positive
real numbers $c_1$ and $c_2$ and natural numbers $n_1$ and $n_2$ such that whenever $m \geq n_1$ and $n \geq n_1$, 
\[ \hat{f}_1(m,n) \leq c_1 \hat{g}_1(m,n), \]
and whenever $m \geq n_2$ and $n \geq n_2$, 
\[ \hat{f}_2(m,n) \leq c_2 \hat{g}_2(m,n). \]

In what follows we will let 
\[ \hat{f}_1 f_2(m,n) = \max \{ f_1(i,j)f_2(i,j) \mid 0 \leq i \leq m, 0 \leq j \leq n \}. \]

We first observe that for any natural numbers $m$ and $n$, 
\[ \hat{f}_1 f_2(m,n) \leq \hat{f}_1(m,n) \hat{f}_2(m,n). \]

Furthermore, because both $g_1$ and $g_2$ are strictly nondecreasing, so is $g_1g_2$.

Let $n_0 = \max(n_1,n_2)$. Then whenever $m \geq n_0$ and $n \geq n_0$, 
\[ \hat{f}_1 f_2(m,n) \leq \hat{f}_1(m,n) \hat{f}_2(m,n) \leq c_1 \hat{g}_1(m,n) \hat{g}_2(m,n) \]
\[ = c_1c_2 \hat{g}_1(m,n) \hat{g}_2(m,n), \]
where $c = c_1c_2$. From Theorem 3.40,
\[ f_1(m,n)f_2(m,n) \in O(g_1(m,n)g_2(m,n)). \]

\[ \square \]

In a similar way, the following extension to Theorem 3.15 can be shown. The proof is left as an exercise.

**Theorem 3.42** Suppose $f_1(m,n) \in \Omega(g_1(m,n))$ and $f_2(m,n) \in \Omega(g_2(m,n))$, where $g_1$ and $g_2$ are strictly nondecreasing. Then

1. $f_1(m,n)f_2(m,n) \in \Omega(g_1(m,n)g_2(m,n))$; and
2. $f_1(m,n) + f_2(m,n) \in \Omega(\max(g_1(m,n),g_2(m,n)))$.

We therefore have the following corollary.

**Corollary 3.43** Suppose $f_1(m,n) \in \Theta(g_1(m,n))$ and $f_2(m,n) \in \Theta(g_2(m,n))$, where $g_1$ and $g_2$ are strictly nondecreasing. Then
1. \( f_1(m, n)f_2(m, n) \in \Theta(g_1(m, n)g_2(m, n)) \); and
2. \( f_1(m, n) + f_2(m, n) \in \Theta(\max(g_1(m, n), g_2(m, n))) \).

Before we can extend Theorem 3.28 to more than one variable, we must first extend the definition of smoothness. In order to do this, we must first extend the definitions of eventually nondecreasing and eventually positive.

**Definition 3.44** Let \( f : \mathbb{N} \times \mathbb{N} \to \mathbb{R}^+ \). \( f \) is said to be *eventually nondecreasing* if there is a natural number \( n_0 \) such that \( f(m, n) \leq f(m + 1, n) \) and \( f(m, n) \leq f(m, n + 1) \) whenever both \( m \geq n_0 \) and \( n \geq n_0 \).

**Definition 3.45** Let \( f : \mathbb{N} \times \mathbb{N} \to \mathbb{R}^+ \). \( f \) is said to be *eventually positive* if there is a natural number \( n_0 \) such that \( f(m, n) > 0 \) whenever both \( m \geq n_0 \) and \( n \geq n_0 \).

**Definition 3.46** Let \( f : \mathbb{N} \times \mathbb{N} \to \mathbb{R}^+ \) be an eventually nondecreasing and eventually positive function. \( f \) is said to be *smooth* if there exist a real number \( c \) and a natural number \( n_0 \) such that \( f(2m, n) \leq cf(m, n) \) and \( f(m, 2n) \leq cf(m, n) \) whenever both \( m \geq n_0 \) and \( n \geq n_0 \).

The following extension to Theorem 3.28 can now be shown — the proof is left as an exercise.

**Theorem 3.47** Let \( f : \mathbb{N} \times \mathbb{N} \to \mathbb{R}^+ \) be a strictly nondecreasing smooth function. Let \( g : \mathbb{N} \to \mathbb{N} \) be an eventually nondecreasing and unbounded function, and let \( X \) denote either \( O \), \( \Omega \), or \( \Theta \). Then

\[
\sum_{i=1}^{g(m)} X(f(i, n)) \subseteq X(g(m)f(g(m), n)).
\]

Having the above theorems, we can now complete the analysis of Add-Matrices. Because we are analyzing the algorithm with respect to two parameters, we view \( n \) as the 2-variable function \( f(m, n) = n \), and we view \( m \) as the 2-variable function \( g(m, n) = m \). We can then apply Corollary 3.43 to \( \Theta(m)\Theta(n) \) to obtain a running time in \( \Theta(mn) \). Alternatively, because \( n \) is smooth, we could apply Theorem 3.47 to obtain

\[
\sum_{i=1}^{m} \Theta(n) \subseteq \Theta(mn).
\]
The results from this section give us the tools we need to analyze iterative algorithms with two natural parameters. Furthermore, all of these results can be easily extended to more than two parameters. Recursive algorithms, however, present a greater challenge. In order to analyze recursive algorithms using more than one natural parameter, we need to be able to handle asymptotic recurrences in more than one variable. This topic is beyond the scope of this book.

### 3.10 Little-\(o\) and Little-\(\omega\)

Occasionally, we would like to use asymptotic notation without ignoring constant factors. Consider, for example, \(f(n) = 3n^2 + 7n + 2\). As \(n\) increases, the \(7n + 2\) term becomes less relevant. In fact, as \(n\) increases, the ratio \(3n^2/f(n)\) approaches 1. We might therefore wish to say that \(f(n)\) is \(3n^2\), plus some low-order terms. We would like to be able to express the fact that these low-order terms are insignificant as \(n\) increases. To this end, we give the following definitions.

**Definition 3.48** Let \(f : \mathbb{N} \to \mathbb{R}_{\geq 0}\). \(o(f(n))\) is the set of all functions \(g : \mathbb{N} \to \mathbb{R}_{\geq 0}\) such that for every positive real number \(c\), there is a natural number \(n_0\) such that \(g(n) < cf(n)\) whenever \(n \geq n_0\).

\(o(f(n))\) is pronounced “little-oh of \(f\) of \(n\).”

**Definition 3.49** Let \(f : \mathbb{N} \to \mathbb{R}_{\geq 0}\). \(\omega(f(n))\) is the set of all functions \(g : \mathbb{N} \to \mathbb{R}_{\geq 0}\) such that for every positive real number \(c\), there is a natural number \(n_0\) such that \(g(n) > cf(n)\) whenever \(n \geq n_0\).

\(\omega(f(n))\) is pronounced “little-omega of \(f\) of \(n\).”

**Example 3.50** \(7n + 2 \in o(n^2)\). In proof, suppose \(c > 0\). We need to find a natural number \(n_0\) such that \(7n + 2 < cn^2\) whenever \(n \geq n_0\). We first observe that this inequality holds if \(n > 0\) and \((7 + 2/n)/c < n\). The left-hand side of this inequality is maximized when \(n = 1\); therefore, if \(n \geq [9/c] + 1\), \(7n + 2 < cn^2\).

Thus, if \(f(n) = 3n^2 + 7n + 2\), then \(f(n) \in 3n^2 + o(n^2)\).

These definitions are similar to the definitions of \(O\) and \(\Omega\), respectively, except that the inequalities hold for every positive real number \(c\), rather than for some positive real number \(c\). Thus, \(g(n) \in o(f(n))\) is a strictly stronger statement than \(g(n) \in O(f(n))\), and \(g(n) \in \omega(f(n))\) is a strictly stronger statement than \(g(n) \in \Omega(f(n))\) (see Figure 3.6). This idea is formalized by the following theorem.

**Theorem 3.51** Let \(f : \mathbb{N} \to \mathbb{R}_{\geq 0}\) be an eventually positive function. Then
Figure 3.6 Venn diagram depicting the relationships between the sets $O(f(n))$, $\Omega(f(n))$, $\Theta(f(n))$, $o(f(n))$, and $\omega(f(n))$

1. $o(f(n)) \subseteq O(f(n)) \setminus \Theta(f(n))$; and
2. $\omega(f(n)) \subseteq \Omega(f(n)) \setminus \Theta(f(n))$,

where $A \setminus B$ denotes the set of elements in $A$ but not in $B$.

Proof: We will only prove part 1; the proof of part 2 is symmetric. Let $g(n) \in o(f(n))$, and let $c$ be any positive real number. Then there is a natural number $n_0$ such that $g(n) < cf(n)$ whenever $n \geq n_0$. Hence, $g(n) \in O(f(n))$.

Furthermore, because the choice of $c$ is arbitrary, we can conclude that $g(n) \notin \Omega(f(n))$; hence, $g(n) \notin \Theta(f(n))$. \hfill $\square$

It may seem at this point that the above theorem could be strengthened to say that $o(f(n)) = O(f(n)) \setminus \Theta(f(n))$ and $\omega(f(n)) = \Omega(f(n)) \setminus \Theta(f(n))$. Indeed, for functions $f$ and $g$ that we typically encounter in the analysis of algorithms, it will be the case that if $g(n) \in O(f(n)) \setminus \Theta(f(n))$ then $g(n) \in o(f(n))$. However, there are exceptions. For example, let $f(n) = n$, and let $g(n) = 2^{\lfloor \lg n \rfloor}$. Then $g(n) \in O(f(n))$ because $g(n) \leq f(n)$ for all
CHAPTER 3. ANALYZING ALGORITHMS

$n \in \mathbb{N}$. Furthermore, when $n = 2^{2^k} - 1$ for $k > 0$, $g(n) = 2^{2^{k-1}} = \sqrt{n + 1}$; hence, $g(n) \not\in \Theta(f(n))$. Finally, when $n = 2^{2^k}$, $g(n) = n$, so $g(n) \not\in o(f(n))$.

Note that we have the same duality between $o$ and $\omega$ as between $O$ and $\Omega$. We therefore have the following theorem.

**Theorem 3.52** Let $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Then $g(n) \in o(f(n))$ iff $f(n) \in \omega(g(n))$.

The following theorems express relationships between common functions using $o$-notation.

**Theorem 3.53** Let $p, q \in \mathbb{R}_{\geq 0}$ such that $p < q$, and suppose $f(n) \in O(n^p)$ and $g(n) \in \Omega(n^q)$. Then $f(n) \in o(g(n))$.

**Proof:** Because $f(n) \in O(n^p)$, there exist a positive real number $c_1$ and a natural number $n_1$ such that

$$f(n) \leq c_1 n^p$$

whenever $n \geq n_1$. Because $g(n) \in \Omega(n^q)$, there exist a positive real number $c_2$ and a natural number $n_2$ such that

$$g(n) \geq c_2 n^q$$

whenever $n \geq n_2$. Combining (3.11) and (3.12), we have

$$f(n) \leq \frac{c_1 g(n)}{c_2 n^{q-p}}$$

whenever $n \geq \max(n_1, n_2)$. Let $c$ be an arbitrary positive real number. Let $n_0 = \max(n_1, n_2, \lceil (c_1/(c_2 c))^{1/(q-p)} \rceil) + 1$. Then when $n \geq n_0$, $n^{q-p} > c_1/(c_2 c)$ because $q > p$. We therefore have,

$$f(n) \leq \frac{c_1 g(n)}{c_2 n^{q-p}} < cg(n).$$

Therefore, $f(n) \in o(g(n))$. □

**Theorem 3.54** Let $p$ and $q$ be any positive real numbers. Then

1. $O(lg^n n) \subseteq o(n^q)$; and
2. $O(n^p) \subseteq o(2^{qn})$.

The proof of Theorem 3.54 requires some additional techniques, which we present in the next section.
3.11 * Use of Limits in Asymptotic Analysis

The astute reader may have noticed a relationship between asymptotic analysis and the concept of a limit. Both of these concepts involve the behavior of a function \( f(n) \) as \( n \) increases. In order to examine this relationship precisely, we now give the formal definition of a limit.

**Definition 3.55** Let \( f : \mathbb{N} \rightarrow \mathbb{R} \), and let \( u \in \mathbb{R} \). We say that
\[
\lim_{n \rightarrow \infty} f(n) = u
\]
if for every positive real number \( c \), there is a natural number \( n_0 \) such that \( |f(n) - u| < c \) whenever \( n \geq n_0 \). Likewise, for a function \( g : \mathbb{R}^0 \rightarrow \mathbb{R} \), we say that
\[
\lim_{x \rightarrow \infty} g(x) = u
\]
if for every positive real number \( c \), there is a real number \( x_0 \) such that \( |g(x) - u| < c \) whenever \( x \geq x_0 \).

Note that for \( f : \mathbb{N} \rightarrow \mathbb{R} \) and \( g : \mathbb{R}^0 \rightarrow \mathbb{R} \), if \( f(n) = g(n) \) for every \( n \in \mathbb{N} \), it follows immediately from the above definition that
\[
\lim_{n \rightarrow \infty} f(n) = \lim_{x \rightarrow \infty} g(x)
\]
whenever the latter limit exists. It is also possible to define infinite limits, but for our purposes we only need finite limits as defined above. Given this definition, we can now formally relate limits to asymptotic notation.

**Theorem 3.56** Let \( f : \mathbb{N} \rightarrow \mathbb{R}^0 \) and \( g : \mathbb{N} \rightarrow \mathbb{R}^0 \). Then
1. \( g(n) \in o(f(n)) \) iff \( \lim_{n \rightarrow \infty} g(n)/f(n) = 0 \); and
2. \( g(n) \in \Theta(f(n)) \) if \( \lim_{n \rightarrow \infty} g(n)/f(n) = x > 0 \).

Note that part 1 is an “if and only if”, whereas part 2 is an “if”. The reason for this is that there are four possibilities, given arbitrary \( f \) and \( g \):

1. \( \lim_{n \rightarrow \infty} g(n)/f(n) = 0 \). In this case \( g(n) \in o(f(n)) \) and \( f(n) \in \omega(g(n)) \).
2. \( \lim_{n \rightarrow \infty} f(n)/g(n) = 0 \). In this case \( f(n) \in o(g(n)) \) and \( g(n) \in \omega(f(n)) \).
3. \( \lim_{n \to \infty} g(n)/f(n) = x > 0 \). In this case, \( g(n) \in \Theta(f(n)) \) and \( f(n) \in \Theta(g(n)) \). (Note that \( \lim_{n \to \infty} f(n)/g(n) = 1/x > 0 \).)

4. Neither \( \lim_{n \to \infty} g(n)/f(n) \) nor \( \lim_{n \to \infty} f(n)/g(n) \) exists. In this case, we can only conclude that \( g(n) \notin o(f(n)) \) and \( f(n) \notin o(g(n)) \)—we do not have enough information to determine whether \( g(n) \in \Theta(f(n)) \).

### Proof of Theorem 3.56:

1. This follows immediately from the definitions of limit and \( o \).

2. Suppose \( \lim_{n \to \infty} g(n)/f(n) = x > 0 \). Then for every positive real number \( c \), there is a natural number \( n_0 \) such that

\[
    x - c < g(n)/f(n) < x + c
\]

whenever \( n \geq n_0 \). Multiplying the above inequalities by \( f(n) \), we have

\[
    (x - c)f(n) < g(n) < (x + c)f(n).
\]

Because these inequalities hold for every positive real number \( c \), and because \( x > 0 \), we may choose \( c = x/2 \), so that both \( x - c \) and \( x + c \) are positive. Therefore, \( g(n) \in \Theta(f(n)) \).

\( \square \)

A powerful tool for evaluating limits of the form given in Theorem 3.56 is L'Hôpital’s rule, which we present without proof in the following theorem.

**Theorem 3.57 (L'Hôpital’s rule)** Let \( f : \mathbb{R}^{\geq 0} \to \mathbb{R} \) and \( g : \mathbb{R}^{\geq 0} \to \mathbb{R} \) be functions such that \( \lim_{x \to \infty} 1/f(x) = 0 \) and \( \lim_{x \to \infty} 1/g(x) = 0 \). Let \( f' \) and \( g' \) denote the derivatives of \( f \) and \( g \), respectively. If \( \lim_{x \to \infty} g'(x)/f'(x) = u \in \mathbb{R} \), then \( \lim_{x \to \infty} g(x)/f(x) = u \).

With this theorem, we can now prove Theorem 3.54.

### Proof of Theorem 3.54:

1. We will use L'Hôpital’s rule to show that \( \lim_{x \to \infty} \lg x/x^{q/p} = 0 \). It will therefore follow that \( \lim_{x \to \infty} \lg^p x/x^q = 0 \). From Theorem 3.56, it will then follow that \( \lg^p n \in o(n^q) \). We leave it as an exercise to show that if \( g(n) \in o(f(n)) \), then \( O(g(n)) \subseteq o(f(n)) \).
CHAPTER 3. ANALYZING ALGORITHMS

We first note that because both \( \log x \) and \( x^{q/p} \) are nondecreasing and unbounded (because \( q \) and \( p \) are both positive), \( \lim_{x \to \infty} 1/ \log x = 0 \) and \( \lim_{x \to \infty} 1/x^{q/p} = 0 \). In order to compute the derivative of \( \log x \), we first observe that \( \log x \ln 2 = \ln x \), where \( \ln \) denotes the natural logarithm or base-\( e \) logarithm, where \( e \approx 2.718 \). Thus, the derivative of \( \log x \) is \( 1/(x \ln 2) \). The derivative of \( x^{q/p} \) is

\[
\frac{q}{p} - 1/p.
\]

Using L’Hôpital’s rule,

\[
\lim_{x \to \infty} \frac{\log x}{x^{q/p}} = \lim_{x \to \infty} \frac{1}{q x^{q/p-1} \ln 2/p} = \lim_{x \to \infty} \frac{p}{q x^{q/p-1} \ln 2} = 0.
\]

Hence, \( \lim_{x \to \infty} \log^p x/x^q = 0 \). Therefore, \( \log^p n \in o(n^q) \) and \( O(\log^p n) \subseteq o(n^q) \).

2. Because \( \lim_{x \to \infty} \log^p x/x^q = 0 \) and \( 2^x \) is nondecreasing and unbounded, it follows that

\[
\lim_{x \to \infty} x^p/2^{qx} = \lim_{x \to \infty} \log^p(2^x)/(2^x)^q = 0.
\]

Therefore, \( n^p \in o(2^{qn}) \) and \( O(n^p) \subseteq o(2^{qm}) \).

\[\square\]

3.12 Summary

Asymptotic notation can be used to express the growth rates of functions in a way that ignores constant factors and focuses on the behavior as the function argument increases. We can therefore use asymptotic notation to analyze performance of algorithms in terms of such measures as worst-case running time or space usage. \( O \) and \( \Omega \) are used to express upper and lower bounds, respectively, while \( \Theta \) is used to express the fact that the upper and lower bounds are tight. \( o \) gives us the ability to abstract away low-order
terms when we don’t want to ignore constant factors. \( \omega \) provides a dual for \( o \).

Analysis of iterative algorithms typically involves summations. Theorem 3.28 gives us a powerful tool for obtaining asymptotic solutions for summations. Analysis of recursive algorithms, on the other hand, typically involves recurrence relations. Theorems 3.31 and 3.32 provide asymptotic solutions for the most common forms of recurrences.

The analyses of the various algorithms for the maximum subsequence sum problem illustrate the utility of asymptotic analysis. We saw that the five algorithms have worst-case running times shown in Figure 3.7. These results correlate well with the actual running times shown in Figure 1.15.

The results of asymptotic analyses can also be used to predict performance degradation. If an algorithm’s running time is in \( \Theta(f(n)) \), then as \( n \) increases, the running time of an implementation must lie between \( cf(n) \) and \( df(n) \) for some positive real numbers \( c \) and \( d \). In fact, for most algorithms, this running time will approach \( cf(n) \) for a single positive real number \( c \). Assuming that this convergence occurs, if we run the algorithm on sufficiently large input, we can approximate \( c \) by dividing the actual running time by \( f(n) \), where \( n \) is the size of the input.

For example, our implementation of MaxSumIter took 1283 seconds to process an input of size \( 2^{14} = 16,384 \). Dividing 1283 by \((16,384)^3\), we obtain a value of \( c = 2.92 \times 10^{-10} \). Evaluating \( cn^3 \) for \( n = 2^{13} \), we obtain a value of 161 seconds. This is very close to the actual running time of 160 seconds on an input of size \( 2^{13} \). Thus, the running time does appear to be converging to \( cn^3 \) for sufficiently large \( n \).

Figure 3.8 shows a plot of the functions estimating the running times of the various maximum subsequence sum implementations, along with the

---

**Figure 3.7** Asymptotic worst-case running times of maximum subsequence sum algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxSumIter</td>
<td>( \Theta(n^3) )</td>
</tr>
<tr>
<td>MaxSumOpt</td>
<td>( \Theta(n^2) )</td>
</tr>
<tr>
<td>MaxSumTD</td>
<td>( \Theta(n^2) )</td>
</tr>
<tr>
<td>MaxSumDC</td>
<td>( \Theta(n \log n) )</td>
</tr>
<tr>
<td>MaxSumBU</td>
<td>( \Theta(n) )</td>
</tr>
</tbody>
</table>
Figure 3.8 Estimated performance of implementations of maximum subsequence sum algorithms

measured running times from Figure 1.15. The functions were derived via the technique outlined above using the timing information from Figure 1.15, taking the largest data set tested for each algorithm. We have extended both axes to show how these functions compare as \( n \) grows as large as \( 2^{30} = 1,073,741,824 \).

For example, consider the functions estimating the running times of MaxSumIter and MaxSumBU. As we have already shown, the function estimating the running time of MaxSumIter is \( f(n) = (2.92 \times 10^{-10})n^3 \). The function we obtained for MaxSumBU is \( g(n) = (1.11 \times 10^{-8})n \). Let us now use these functions to estimate the time these implementations would require to process an array of \( 2^{30} \) elements. \( g(2^{30}) = 11.9 \) seconds, whereas \( f(2^{30}) = 3.61 \times 10^{17} \) seconds, or over 11 billion years! Even if we could speed up the processor by a factor of one million, this implementation would still require over 11,000 years.

Though this example clearly illustrates the utility of asymptotic analysis, a word of caution is in order. Asymptotic notation allows us to focus on growth rates while ignoring constant factors. However, constant factors
Figure 3.9 Functions illustrating the practical limitations of asymptotic notation

\[ \frac{\sqrt{n}}{\lg^{16} n} \]

\[ \frac{\sqrt{n}}{\ln n} \]

can be relevant. For example, two linear-time algorithms will not yield comparable performance if the hidden constants are very different.

For a more subtle example, consider the functions \( \lg^{16} n \) and \( \sqrt{n} \), shown in Figure 3.9. From Theorem 3.54, \( O(\lg^{16} n) \subseteq o(\sqrt{n}) \), so that as \( n \) increases, \( \lg^{16} n \) grows much more slowly than does \( \sqrt{n} \). However, consider \( n = 2^{32} = 4,294,967,296 \). For this value, \( \sqrt{n} = 2^{16} = 65,536 \), whereas

\[
\lg^{16} n = 32^{16} = 1,208,925,819,614,629,174,706,176.
\]

\( \lg^{16} n \) remains larger than \( \sqrt{n} \) until \( n = 2^{256} \) — a 78-digit number. After that, \( \sqrt{n} \) does grow much more rapidly than does \( \lg^{16} n \), but it is hard to see any practical value in studying the behaviors of these functions at such large values.

Finally, the running time analyses we have seen in this chapter have all been worst-case analyses. For some algorithms, the worst case is much worse than typical cases, so that in practice, the algorithm performs much better than a worst-case analysis would suggest. Later, we will see other kinds of analyses that may be more appropriate in such cases. However, we must realize that there is a limit to what can be determined analytically.
3.13 Exercises

Exercise 3.1 Prove that if $g(n) \in O(f(n))$, then $O(g(n)) \subseteq O(f(n))$.

Exercise 3.2 Prove that for any $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$, $f(n) \in \Theta(f(n))$.  

Exercise 3.3 Prove that if $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$.

Exercise 3.4 Prove Theorem 3.15.

Exercise 3.5 For each of the following, give functions $f(n) \in \Theta(n)$ and $g(n) \in \Theta(n)$ that satisfy the given property.

a. $f(n) - g(n) \in \Theta(n)$.

b. $f(n) - g(n) \notin \Theta(n)$.

Exercise 3.6 Suppose that $g_1(n) \in \Theta(f_1(n))$ and $g_2(n) \in \Theta(f_2(n))$, where $g_2$ and $f_2$ are eventually positive. Prove that $g_1(n)/g_2(n) \in \Theta(f_1(n)/f_2(n))$.

Exercise 3.7 Show that the result in Exercise 3.6 does not necessarily hold if we replace $\Theta$ by $O$.

Exercise 3.8 Let $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$, where $g$ is eventually positive. Prove that $f(n) \in O(g(n))$ iff there is a positive real number $c$ such that $f(n) \leq cg(n)$ whenever $g(n) > 0$.

* Exercise 3.9 Let $f(n) = 2^{\lfloor \log \log n \rfloor}$, where we assume that $f(n) = 0$ for $n \leq 1$.

a. Show that $f(n) \in O(n)$.

b. Show that $f(n)$ is not smooth; i.e., show that for every $c \in \mathbb{R}_{>0}$ and every $n_0 \in \mathbb{N}$, there is some $n \geq n_0$ such that $f(2n) > cf(n)$. [Hint: Consider a sufficiently large value of $n$ having the form $2^{2^k - 1}$.

* Exercise 3.10 The goal of this exercise is to prove Theorem 3.28. Let $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a smooth function, $g : \mathbb{N} \to \mathbb{N}$ be an eventually nondecreasing and unbounded function, and $h : \mathbb{N} \to \mathbb{R}_{\geq 0}$. 
a. Show that if \( h(n) \in O(f(n)) \), then there exist natural numbers \( n_0 \) and \( n_1 \), a positive real number \( c \), and a nonnegative real number \( d \) such that for every \( n \geq n_1 \),
\[
\sum_{i=1}^{g(n)} h(i) \leq d + \sum_{i=n_0}^{g(n)} cf(g(n)).
\]

b. Use part a to prove that
\[
\sum_{i=1}^{g(n)} O(f(i)) \subseteq O(g(n)f(g(n))).
\]

c. Show that if \( h(n) \in \Omega(f(n)) \), then there exist natural numbers \( n_0 \) and \( n_1 \) and positive real numbers \( c \) and \( d \) such that for every \( n \geq n_0 \),
\[
f(n) \geq f(2n)/d,
\]
and for every \( n \geq n_1 \), both
\[
\sum_{i=1}^{g(n)} h(i) \geq \sum_{i=[g(n)/2]}^{g(n)} cf([g(n)/2])
\]
and
\[
g(n) \geq 2n_0
\]
hold.

d. Use part c to prove that
\[
\sum_{i=1}^{g(n)} \Omega(f(i)) \subseteq \Omega(g(n)f(g(n))).
\]

e. Use parts b and d to prove that
\[
\sum_{i=1}^{g(n)} \Theta(f(i)) \subseteq \Theta(g(n)f(g(n))).
\]
* Exercise 3.11 Prove that for every smooth function \( f : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \) and every eventually nondecreasing and unbounded function \( g : \mathbb{N} \rightarrow \mathbb{N} \), and every \( X \in \{ \Omega, \Theta \} \),

\[
\sum_{i=1}^{g(n)} X(f(i)) \neq X(g(n)f(g(n))).
\]

**Hint:** First identify a property that every function in the set on the left-hand side must satisfy, but which functions in the set on the right-hand side need not satisfy.

Exercise 3.12 Prove Theorem 3.29.

Exercise 3.13 Analyze the worst-case running time of the following code fragments, assuming that \( n \) represents the problem size. Express your result as simply as possible using \( \Theta \)-notation.

a. for \( i \leftarrow 0 \) to \( 2n \)
   for \( j \leftarrow 0 \) to \( 3n \)
   \( k \leftarrow k + i + j \)

b. for \( i \leftarrow 1 \) to \( n^2 \)
   for \( j \leftarrow i \) to \( i^3 \)
   \( k \leftarrow k + 1 \)

* c. \( i \leftarrow n \)
   while \( i > 0 \)
   for \( j \leftarrow 1 \) to \( i^2 \)
   \( x \leftarrow (x + j)/2 \)
   \( i \leftarrow \lfloor i/2 \rfloor \)

Exercise 3.14 Give asymptotic solutions to the following asymptotic recurrences. In each case, you may assume that \( f : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \) is an eventually nondecreasing function.

a. \( f(n) \in 2f(n-1) + \Theta(1) \)

   for \( n > 0 \).
CHAPTER 3. ANALYZING ALGORITHMS

b. \[ f(n) \in f(n - 1) + \Omega(n \log n) \]

for \( n > 0 \).

c. \[ f(n) \in 4f(n/2) + O(\log^2 n) \]

whenever \( n = 3 \cdot 2^k \) for a positive integer \( k \).

d. \[ f(n) \in 5f(n/3) + \Theta(n^2) \]

whenever \( n = 3^k \) for a positive integer \( k \).

e. \[ f(n) \in 3f(n/2) + O(n) \]

whenever \( n = 8 \cdot 2^k \) for a positive integer \( k \).

**Exercise 3.15** Analyze the worst-case running time of **SelectByMedian**, shown in Figure 2.7, assuming that **Median** is implemented to run in \( \Theta(n) \) time. Express your result as simply as possible using \( \Theta \)-notation.

**Exercise 3.16** Analyze the worst-case running time of the following functions. Express your result as simply as possible using \( \Theta \)-notation.

a. **SlowSort**(A[1..n])
   - else if \( n > 2 \)
     - **SlowSort**(A[1..n - 1])
     - **SlowSort**(A[2..n])
     - **SlowSort**(A[1..n - 1])

b. **FindMax**(A[1..n])
   - if \( n = 0 \)
     - error
   - else if \( n = 1 \)
     - return \( A[1] \)
   - else
     - return \( \text{Max}(\text{FindMax}(A[1..\lfloor n/2 \rfloor]), \text{FindMax}(A[\lfloor n/2 \rfloor + 1..n])) \)
c. \texttt{FindMin}(A[1..n])
   \begin{verbatim}
   if \ n = 0
       error
   else if \ n = 1
       return \ A[1]
   else
       B ← new Array[1..\lceil n/2 \rceil]
       for \ i ← 1 to \ \lfloor n/2 \rfloor
       if \ n \ mod 2 = 1
           B[\lceil n/2 \rceil] ← A[n]
       return \ FindMin(B[1..\lceil n/2 \rceil])
   \end{verbatim}

\textbf{Exercise 3.17} Analyze the worst-case space usage of each of the functions given in Exercise 3.16. Express your result as simply as possible using $\Theta$-notation.

\textbf{* Exercise 3.18} Prove that if $f : \mathbb{N} \to \mathbb{R}^\geq 0$ is smooth and $g(n) \in \Theta(n)$, then $f(g(n)) \in \Theta(f(n))$.

\textbf{* Exercise 3.19} Prove that for any smooth function $g : \mathbb{N} \to \mathbb{R}^\geq 0$, there is a natural number $k$ such that $g(n) \in O(n^k)$.

\textbf{* Exercise 3.20} The goal of this exercise is to prove Theorem 3.31. Let

\[ f(n) = af(n−1) + X(b^ng(n)) \]

for $n > n_0$, where $n_0 \in \mathbb{N}$, $a \geq 1$ and $b \geq 1$ are real numbers, $g(n)$ is a smooth function, and $X$ is either $O$, $\Omega$, or $\Theta$. In what follows, let $n_1$ be any natural number such that $n_1 \geq n_0$ and whenever $n \geq n_1$, $0 < g(n) \leq g(n+1)$.

a. Prove by induction on $n$ that if $X$ is $O$, then there is a positive real number $c$ such that for $n \geq n_1$,

\[ f(n) \leq a^{n−n_1}f(n_1) + ca^n \sum_{i=n_1+1}^{n} (b/a)^i g(i). \]

b. Prove by induction on $n$ that if $X$ is $\Omega$, then there is a positive real number $d$ such that

\[ f(n) \geq a^{n−n_1}f(n_1) + da^n \sum_{i=n_1+1}^{n} (b/a)^i g(i). \]
c. Use parts a and b, together with Equation (2.2), to show that if \( a < b \), then \( f(n) \in X(b^n g(n)) \).

d. Use parts a and b, together with Theorem 3.28, to show that if \( a = b \), then \( f(n) \in X(na^n g(n)) \).

e. Suppose \( a > b \), and let \( r = \sqrt{a/b} \). Show that there is a natural number \( n_2 \geq n_0 \) such that for every \( n \geq n_2 \), \( 0 < g(n) \leq g(n + 1) \) and

\[
\sum_{i=n_2+1}^{n} (b/a)^i g(i) \leq \frac{r}{r-1}.
\]

[**Hint:** Use the result of Exercise 3.19 and Theorem 3.54 to show that for sufficiently large \( i \), \( g(i) \leq r^i \); then apply Equation (2.2).]

f. Use parts a, b, and e to show that if \( a > b \), then \( f(n) \in X(a^n) \).

**Exercise 3.21** Let \( f : \mathbb{N} \to \mathbb{R}^\geq 0 \) be a function satisfying (3.7). Prove by induction on \( n \) that \( f(n) \leq f(n + 1) \) for \( n \geq 1 \).

**Exercise 3.22** Prove Theorem 3.32.

**Exercise 3.23** Show that COPY, specified in Figure 1.18 on page 22, can be implemented to run in \( \Theta(n) \) time, \( \Theta(n) \) space, and \( \Theta(1) \) stack space, where \( n \) is the size of both of the arrays. Note that function calls use space from the stack, but constructed arrays do not. Also recall that the parameters \( A[1..n] \) and \( B[1..n] \) should not be included in the analysis of space usage. Your algorithm should work correctly even for calls like COPY(\( A[1..n-1] \), \( A[2..n] \)) (see Exercise 1.4).

**Exercise 3.24** Prove Theorem 3.39.

**Exercise 3.25** Complete the proof of Theorem 3.41.

**Exercise 3.26** Prove Theorem 3.42.

* **Exercise 3.27** Prove Theorem 3.47. [**Hint:** First work Exercise 3.10, but note that not all parts of that exercise extend directly to multiple variables.]

2/25/11.
CHAPTER 3. ANALYZING ALGORITHMS

Exercise 3.28 Let \( A[1..n] \) be an array of numbers. An inversion is a pair of indices \( 1 \leq i < j \leq n \) such that \( A[i] > A[j] \). The number of inversions in \( A \) is a way to quantify how nearly sorted \( A \) is — the fewer inversions \( A \) has, the more nearly sorted it is. Let \( I \) denote the number of inversions in \( A \). Show that INSERTIONSORT (Figure 1.7, page 11) runs in \( \Theta(n + I) \) time in the worst case. (Thus, INSERTIONSORT is very efficient when the array is nearly sorted.) Note that because the analysis is in terms of two variables, “worst case” refers to the worst-case input for each pair of values \( n \) and \( I \).

Exercise 3.29 Prove that if \( g(n) \in o(f(n)) \), then \( O(g(n)) \subseteq o(f(n)) \).

** Exercise 3.30 Find two smooth functions \( f : \mathbb{N} \to \mathbb{R}_{\geq 0} \) and \( g : \mathbb{N} \to \mathbb{R}_{\geq 0} \) such that \( g(n) \in O(f(n)) \), but \( g(n) \) is in neither \( \Theta(f(n)) \) nor \( o(f(n)) \).

Exercise 3.31 Prove that for any real numbers \( a > 1 \) and \( b > 1 \),
\[
O(\log_a n) = O(\log_b n).
\]

* Exercise 3.32 Prove that

\[
\lg(n!) \in \Theta(n \lg n).
\]

3.14 Chapter Notes

Asymptotic notation predates electronic computing by several decades. Big-\(O\) notation was introduced by Bachman [7] in 1894, but with a meaning slightly different from our definition. In the original definition, \( O(f(n)) \) was used to denote a specific, but unknown, function belonging to the set we have defined to be \( O(f(n)) \). According to the original definition, it was proper to write,
\[
2n^2 + 7n - 4 = O(n^2).
\]
However, one would never have written,
\[
O(n^2) = 2n^2 + 7n - 4.
\]
Thus, the “=” symbol was used to denote not equality, but a relation that is not even symmetric.

Over the years, many have observed that a set-based definition, as we have given here, is more sound mathematically. In fact, Brassard [17] claims that as long ago as 1962, a set-based treatment was taught consistently
in Amsterdam. It was Brassard’s paper [17], however, that in 1985 first made a strong case for using set-based notation consistently. Though we are in full agreement with his position, use of the original definition is still widespread. Alternatively, some authors give set-based definitions, then abuse the notation by using “=” instead of “∈” or “⊆”. For a justification of this practice, see the second edition of Knuth [76] or Cormen, et al. [25]. For more information on the development of asymptotic notation, including variations not discussed here, see Brassard [17]. The definitions of asymptotic notation on multiple variables are from [64].

Knuth [76] introduced the study of the analysis of running times of algorithms. The notion of a smooth function is due to Brassard [17]. Many techniques exist for solving summations and recurrences; a good resource is Graham, Knuth, and Patashnik [56].