2

Algorithm Analysis

Algorithms are the most important and durable part of computer science because they can be studied in a language- and machine-independent way. This means that we need techniques that enable us to compare the efficiency of algorithms without implementing them. Our two most important tools are (1) the RAM model of computation and (2) the asymptotic analysis of worst-case complexity.

Assessing algorithmic performance makes use of the "big Oh" notation that, proves essential to compare algorithms and design more efficient ones. While the hopelessly *practical* person may blanch at the notion of theoretical analysis, we present the material because it really is useful in thinking about algorithms.

This method of keeping score will be the most mathematically demanding part of this book. But once you understand the intuition behind these ideas, the formalism becomes a lot easier to deal with.

2.1 The RAM Model of Computation

Machine-independent algorithm design depends upon a hypothetical computer called the *Random Access Machine* or RAM. Under this model of computation, we are confronted with a computer where:

- Each *simple* operation (+, *, –, =, if, call) takes exactly one time step.
- Loops and subroutines are *not* considered simple operations. Instead, they are the composition of many single-step operations. It makes no sense for *sort* to be a single-step operation, since sorting 1,000,000 items will certainly take much longer than sorting 10 items. The time it takes to run through a loop or execute a subprogram depends upon the number of loop iterations or the specific nature of the subprogram.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_2,

- Each memory access takes exactly one time step. Further, we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk.

Under the RAM model, we measure run time by counting up the number of steps an algorithm takes on a given problem instance. If we assume that our RAM executes a given number of steps per second, this operation count converts naturally to the actual running time.

The RAM is a simple model of how computers perform. Perhaps it sounds too simple. After all, multiplying two numbers takes more time than adding two numbers on most processors, which violates the first assumption of the model. Fancy compiler loop unrolling and hyperthreading may well violate the second assumption. And certainly memory access times differ greatly depending on whether data sits in cache or on the disk. This makes us zero for three on the truth of our basic assumptions.

And yet, despite these complaints, the RAM proves an *excellent* model for understanding how an algorithm will perform on a real computer. It strikes a fine balance by capturing the essential behavior of computers while being simple to work with. We use the RAM model because it is useful in practice.

Every model has a size range over which it is useful. Take, for example, the model that the Earth is flat. You might argue that this is a bad model, since it has been fairly well established that the Earth is in fact round. But, when laying the foundation of a house, the flat Earth model is sufficiently accurate that it can be reliably used. It is so much easier to manipulate a flat-Earth model that it is inconceivable that you would try to think spherically when you don't have to.[1]

The same situation is true with the RAM model of computation. We make an abstraction that is generally very useful. It is quite difficult to design an algorithm such that the RAM model gives you substantially misleading results. The robustness of the RAM enables us to analyze algorithms in a machine-independent way.

Take-Home Lesson: Algorithms can be understood and studied in a language- and machine-independent manner.

2.1.1 Best, Worst, and Average-Case Complexity

Using the RAM model of computation, we can count how many steps our algorithm takes on any given input instance by executing it. However, to understand how good or bad an algorithm is in general, we must know how it works over *all* instances.

To understand the notions of the best, worst, and average-case complexity, think about running an algorithm over all possible instances of data that can be

[1]The Earth is not completely spherical either, but a spherical Earth provides a useful model for such things as longitude and latitude.

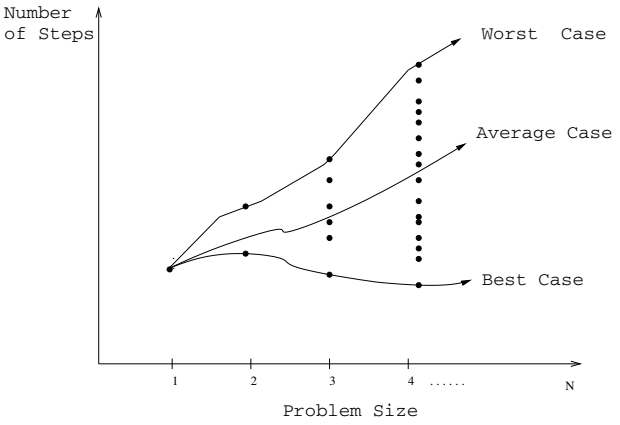

Figure 2.1: Best, worst, and average-case complexity

fed to it. For the problem of sorting, the set of possible input instances consists of all possible arrangements of n keys, over all possible values of n. We can represent each input instance as a point on a graph (shown in Figure 2.1) where the x-axis represents the size of the input problem (for sorting, the number of items to sort), and the y-axis denotes the number of steps taken by the algorithm in this instance.

These points naturally align themselves into columns, because only integers represent possible input size (e.g., it makes no sense to sort 10.57 items). We can define three interesting functions over the plot of these points:

- The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken in any instance of size n. This represents the curve passing through the highest point in each column.
- The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken in any instance of size n. This represents the curve passing through the lowest point of each column.
- The *average-case complexity* of the algorithm, which is the function defined by the average number of steps over all instances of size n.

The worst-case complexity proves to be most useful of these three measures in practice. Many people find this counterintuitive. To illustrate why, try to project what will happen if you bring n dollars into a casino to gamble. The best case, that you walk out owning the place, is possible but so unlikely that you should not even think about it. The worst case, that you lose all n dollars, is easy to calculate and distressingly likely to happen. The average case, that the typical bettor loses 87.32% of the money that he brings to the casino, is difficult to establish and its meaning subject to debate. What exactly does *average* mean? Stupid people lose

more than smart people, so are you smarter or stupider than the average person, and by how much? Card counters at blackjack do better on average than customers who accept three or more free drinks. We avoid all these complexities and obtain a very useful result by just considering the worst case.

The important thing to realize is that each of these time complexities define a numerical function, representing time versus problem size. These functions are as well defined as any other numerical function, be it $y = x^2 - 2x + 1$ or the price of Google stock as a function of time. But time complexities are such complicated functions that we must simplify them to work with them. For this, we need the "Big Oh" notation.

2.2 The Big Oh Notation

The best, worst, and average-case time complexities for any given algorithm are numerical functions over the size of possible problem instances. However, it is very difficult to work precisely with these functions, because they tend to:

- *Have too many bumps* – An algorithm such as binary search typically runs a bit faster for arrays of size exactly $n = 2^k - 1$ (where k is an integer), because the array partitions work out nicely. This detail is not particularly significant, but it warns us that the *exact* time complexity function for any algorithm is liable to be very complicated, with little up and down bumps as shown in Figure 2.2.

- *Require too much detail to specify precisely* – Counting the exact number of RAM instructions executed in the worst case requires the algorithm be specified to the detail of a complete computer program. Further, the precise answer depends upon uninteresting coding details (e.g., did he use a case statement or nested ifs?). Performing a precise worst-case analysis like

  $$T(n) = 12754n^2 + 4353n + 834 \lg_2 n + 13546$$

  would clearly be very difficult work, but provides us little extra information than the observation that "the time grows quadratically with n."

It proves to be much easier to talk in terms of simple upper and lower bounds of time-complexity functions using the Big Oh notation. The Big Oh simplifies our analysis by ignoring levels of detail that do not impact our comparison of algorithms.

The Big Oh notation ignores the difference between multiplicative constants. The functions $f(n) = 2n$ and $g(n) = n$ are identical in Big Oh analysis. This makes sense given our application. Suppose a given algorithm in (say) C language ran twice as fast as one with the same algorithm written in Java. This multiplicative

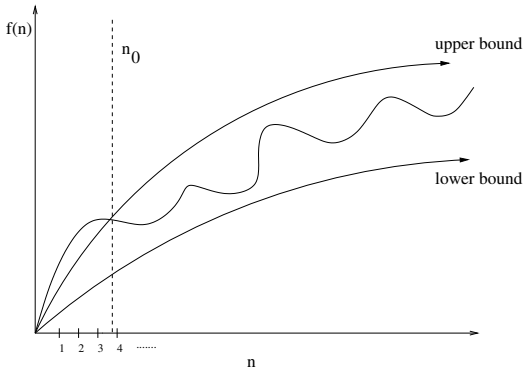

Figure 2.2: Upper and lower bounds valid for $n > n_0$ smooth out the behavior of complex functions

factor of two tells us nothing about the algorithm itself, since both programs implement exactly the same algorithm. We ignore such constant factors when comparing two algorithms.

The formal definitions associated with the Big Oh notation are as follows:

- $f(n) = O(g(n))$ means $c \cdot g(n)$ is an *upper bound* on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\leq c \cdot g(n)$, for large enough n (i.e. , $n \geq n_0$ for some constant n_0).
- $f(n) = \Omega(g(n))$ means $c \cdot g(n)$ is a *lower bound* on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\geq c \cdot g(n)$, for all $n \geq n_0$.
- $f(n) = \Theta(g(n))$ means $c_1 \cdot g(n)$ is an upper bound on $f(n)$ and $c_2 \cdot g(n)$ is a lower bound on $f(n)$, for all $n \geq n_0$. Thus there exist constants c_1 and c_2 such that $f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.

Got it? These definitions are illustrated in Figure 2.3. Each of these definitions assumes a constant n_0 beyond which they are always satisfied. We are not concerned about small values of n (i.e. , anything to the left of n_0). After all, we don't really care whether one sorting algorithm sorts six items faster than another, but seek which algorithm proves faster when sorting 10,000 or 1,000,000 items. The Big Oh notation enables us to ignore details and focus on the big picture.

Take-Home Lesson: The Big Oh notation and worst-case analysis are tools that greatly simplify our ability to compare the efficiency of algorithms.

Make sure you understand this notation by working through the following examples. We choose certain constants (c and n_0) in the explanations below because

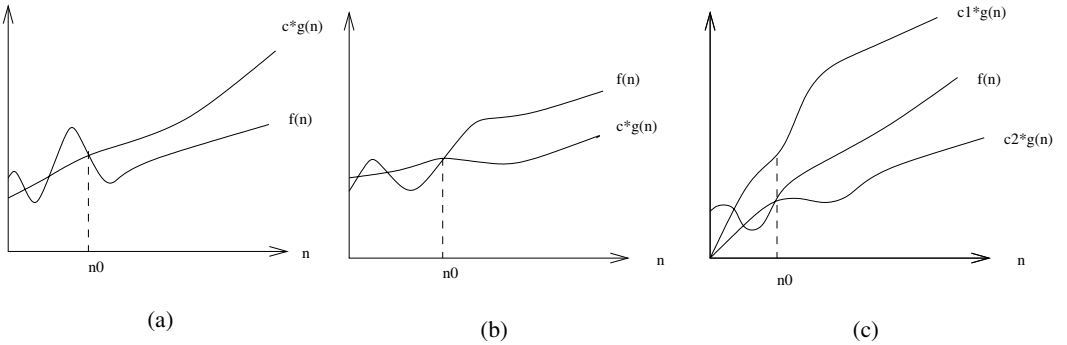

Figure 2.3: Illustrating the big (a) O, (b) Ω, and (c) Θ notations

they work and make a point, but other pairs of constants will do exactly the same job. You are free to choose any constants that maintain the same inequality—ideally constants that make it obvious that the inequality holds:

$3n^2 - 100n + 6 = O(n^2)$, because I choose $c = 3$ and $3n^2 > 3n^2 - 100n + 6$;

$3n^2 - 100n + 6 = O(n^3)$, because I choose $c = 1$ and $n^3 > 3n^2 - 100n + 6$ when $n > 3$;

$3n^2 - 100n + 6 \neq O(n)$, because for any c I choose $c \times n < 3n^2$ when $n > c$;

$3n^2 - 100n + 6 = \Omega(n^2)$, because I choose $c = 2$ and $2n^2 < 3n^2 - 100n + 6$ when $n > 100$;

$3n^2 - 100n + 6 \neq \Omega(n^3)$, because I choose $c = 3$ and $3n^2 - 100n + 6 < n^3$ when $n > 3$;

$3n^2 - 100n + 6 = \Omega(n)$, because for any c I choose $cn < 3n^2 - 100n + 6$ when $n > 100c$;

$3n^2 - 100n + 6 = \Theta(n^2)$, because both O and Ω apply;

$3n^2 - 100n + 6 \neq \Theta(n^3)$, because only O applies;

$3n^2 - 100n + 6 \neq \Theta(n)$, because only Ω applies.

The Big Oh notation provides for a rough notion of equality when comparing functions. It is somewhat jarring to see an expression like $n^2 = O(n^3)$, but its meaning can always be resolved by going back to the definitions in terms of upper and lower bounds. It is perhaps most instructive to read the "=" here as meaning *one of the functions that are*. Clearly, n^2 is one of functions that are $O(n^3)$.

Stop and Think: Back to the Definition

Problem: Is $2^{n+1} = \Theta(2^n)$?

Solution: Designing novel algorithms requires cleverness and inspiration. However, applying the Big Oh notation is best done by swallowing any creative instincts you may have. All Big Oh problems can be correctly solved by going back to the definition and working with that.

- *Is* $2^{n+1} = O(2^n)$*?* Well, $f(n) = O(g(n))$ iff (*if and only if*) there exists a constant c such that for all sufficiently large n $f(n) \leq c \cdot g(n)$. Is there? The key observation is that $2^{n+1} = 2 \cdot 2^n$, so $2 \cdot 2^n \leq c \cdot 2^n$ for any $c \geq 2$.
- *Is* $2^{n+1} = \Omega(2^n)$*?* Go back to the definition. $f(n) = \Omega(g(n))$ iff there exists a constant $c > 0$ such that for all sufficiently large n $f(n) \geq c \cdot g(n)$. This would be satisfied for any $0 < c \leq 2$. Together the Big Oh and Ω bounds imply $2^{n+1} = \Theta(2^n)$

∎

Stop and Think: Hip to the Squares?

Problem: Is $(x + y)^2 = O(x^2 + y^2)$.

Solution: Working with the Big Oh means going back to the definition at the slightest sign of confusion. By definition, this expression is valid iff we can find some c such that $(x + y)^2 \leq c(x^2 + y^2)$.

My first move would be to expand the left side of the equation, i.e. $(x + y)^2 = x^2 + 2xy + y^2$. If the middle $2xy$ term wasn't there, the inequality would clearly hold for any $c > 1$. But it is there, so we need to relate the $2xy$ to $x^2 + y^2$. What if $x \leq y$? Then $2xy \leq 2y^2 \leq 2(x^2 + y^2)$. What if $x \geq y$? Then $2xy \leq 2x^2 \leq 2(x^2 + y^2)$. Either way, we now can bound this middle term by two times the right-side function. This means that $(x + y)^2 \leq 3(x^2 + y^2)$, and so the result holds. ∎

2.3 Growth Rates and Dominance Relations

With the Big Oh notation, we cavalierly discard the multiplicative constants. Thus, the functions $f(n) = 0.001n^2$ and $g(n) = 1000n^2$ are treated identically, even though $g(n)$ is a million times larger than $f(n)$ for all values of n.

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 2.4: Growth rates of common functions measured in nanoseconds

The reason why we are content with coarse Big Oh analysis is provided by Figure 2.4, which shows the growth rate of several common time analysis functions. In particular, it shows how long algorithms that use $f(n)$ operations take to run on a fast computer, where each operation takes one nanosecond (10^{-9} seconds). The following conclusions can be drawn from this table:

- All such algorithms take roughly the same time for $n = 10$.
- Any algorithm with $n!$ running time becomes useless for $n \geq 20$.
- Algorithms whose running time is 2^n have a greater operating range, but become impractical for $n > 40$.
- Quadratic-time algorithms whose running time is n^2 remain usable up to about $n = 10,000$, but quickly deteriorate with larger inputs. They are likely to be hopeless for $n >$ 1,000,000.
- Linear-time and $n \lg n$ algorithms remain practical on inputs of one billion items.
- An $O(\lg n)$ algorithm hardly breaks a sweat for any imaginable value of n.

The bottom line is that even ignoring constant factors, we get an excellent idea of whether a given algorithm is appropriate for a problem of a given size. An algorithm whose running time is $f(n) = n^3$ seconds will beat one whose running time is $g(n) =$ 1,000,000 $\cdot n^2$ seconds only when $n <$ 1,000,000. Such enormous differences in constant factors between algorithms occur far less frequently in practice than large problems do.

2.3.1 Dominance Relations

The Big Oh notation groups functions into a set of classes, such that all the functions in a particular class are equivalent with respect to the Big Oh. Functions $f(n) = 0.34n$ and $g(n) = 234{,}234n$ belong in the same class, namely those that are order $\Theta(n)$. Further, when two functions f and g belong to different classes, they are *different* with respect to our notation. Either $f(n) = O(g(n))$ or $g(n) = O(f(n))$, but not both.

We say that a faster-growing function *dominates* a slower-growing one, just as a faster-growing country eventually comes to dominate the laggard. When f and g belong to different classes (i.e., $f(n) \neq \Theta(g(n))$), we say g *dominates* f when $f(n) = O(g(n))$, sometimes written $g \gg f$.

The good news is that only a few function classes tend to occur in the course of basic algorithm analysis. These suffice to cover almost all the algorithms we will discuss in this text, and are listed in order of increasing dominance:

- *Constant functions,* $f(n) = 1$ – Such functions might measure the cost of adding two numbers, printing out "The Star Spangled Banner," or the growth realized by functions such as $f(n) = \min(n, 100)$. In the big picture, there is no dependence on the parameter n.

- *Logarithmic functions,* $f(n) = \log n$ – Logarithmic time-complexity shows up in algorithms such as binary search. Such functions grow quite slowly as n gets big, but faster than the constant function (which is standing still, after all). Logarithms will be discussed in more detail in Section 2.6 (page 46)

- *Linear functions,* $f(n) = n$ – Such functions measure the cost of looking at each item once (or twice, or ten times) in an n-element array, say to identify the biggest item, the smallest item, or compute the average value.

- *Superlinear functions,* $f(n) = n \lg n$ – This important class of functions arises in such algorithms as Quicksort and Mergesort. They grow just a little faster than linear (see Figure 2.4), just enough to be a different dominance class.

- *Quadratic functions,* $f(n) = n^2$ – Such functions measure the cost of looking at most or all *pairs* of items in an n-element universe. This arises in algorithms such as insertion sort and selection sort.

- *Cubic functions,* $f(n) = n^3$ – Such functions enumerate through all *triples* of items in an n-element universe. These also arise in certain dynamic programming algorithms developed in Chapter 8.

- *Exponential functions,* $f(n) = c^n$ *for a given constant* $c > 1$ – Functions like 2^n arise when enumerating all subsets of n items. As we have seen, exponential algorithms become useless fast, but not as fast as...

- *Factorial functions,* $f(n) = n!$ – Functions like $n!$ arise when generating all permutations or orderings of n items.

The intricacies of dominance relations will be futher discussed in Section 2.9.2 (page 56). However, all you really need to understand is that:

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$$

Take-Home Lesson: Although esoteric functions arise in advanced algorithm analysis, a small variety of time complexities suffice and account for most algorithms that are widely used in practice.

2.4 Working with the Big Oh

You learned how to do simplifications of algebraic expressions back in high school. Working with the Big Oh requires dusting off these tools. *Most* of what you learned there still holds in working with the Big Oh, but not everything.

2.4.1 Adding Functions

The sum of two functions is governed by the dominant one, namely:

$$O(f(n)) + O(g(n)) \rightarrow O(\max(f(n), g(n)))$$

$$\Omega(f(n)) + \Omega(g(n)) \rightarrow \Omega(\max(f(n), g(n)))$$

$$\Theta(f(n)) + \Theta(g(n)) \rightarrow \Theta(\max(f(n), g(n)))$$

This is very useful in simplifying expressions, since it implies that $n^3 + n^2 + n + 1 = O(n^3)$. Everything is small potatoes besides the dominant term.

The intuition is as follows. At least half the bulk of $f(n) + g(n)$ must come from the larger value. The dominant function will, by definition, provide the larger value as $n \rightarrow \infty$. Thus, dropping the smaller function from consideration reduces the value by at most a factor of $1/2$, which is just a multiplicative constant. Suppose $f(n) = O(n^2)$ and $g(n) = O(n^2)$. This implies that $f(n) + g(n) = O(n^2)$ as well.

2.4.2 Multiplying Functions

Multiplication is like repeated addition. Consider multiplication by any constant $c > 0$, be it 1.02 or 1,000,000. Multiplying a function by a constant can not affect its asymptotic behavior, because we can multiply the bounding constants in the Big Oh analysis of $c \cdot f(n)$ by $1/c$ to give appropriate constants for the Big Oh analysis of $f(n)$. Thus:

$$O(c \cdot f(n)) \rightarrow O(f(n))$$

$$\Omega(c \cdot f(n)) \rightarrow \Omega(f(n))$$

$$\Theta(c \cdot f(n)) \rightarrow \Theta(f(n))$$

Of course, c must be strictly positive (i.e., $c > 0$) to avoid any funny business, since we can wipe out even the fastest growing function by multiplying it by zero.

On the other hand, when two functions in a product are increasing, both are important. The function $O(n! \log n)$ dominates $n!$ just as much as $\log n$ dominates 1. In general,

$$O(f(n)) * O(g(n)) \rightarrow O(f(n) * g(n))$$

$$\Omega(f(n)) * \Omega(g(n)) \rightarrow \Omega(f(n) * g(n))$$

$$\Theta(f(n)) * \Theta(g(n)) \rightarrow \Theta(f(n) * g(n))$$

Stop and Think: Transitive Experience

Problem: Show that Big Oh relationships are transitive. That is, if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

Solution: We always go back to the definition when working with the Big Oh. What we need to show here is that $f(n) \leq c_3 h(n)$ for $n > n_3$ given that $f(n) \leq c_1 g(n)$ and $g(n) \leq c_2 h(n)$, for $n > n_1$ and $n > n_2$, respectively. Cascading these inequalities, we get that

$$f(n) \leq c_1 g(n) \leq c_1 c_2 h(n)$$

for $n > n_3 = \max(n_1, n_2)$. ∎

2.5 Reasoning About Efficiency

Gross reasoning about an algorithm's running time of is usually easy given a precise written description of the algorithm. In this section, I will work through several examples, perhaps in greater detail than necessary.

2.5.1 Selection Sort

Here we analyze the selection sort algorithm, which repeatedly identifies the smallest remaining unsorted element and puts it at the end of the sorted portion of the array. An animation of selection sort in action appears in Figure 2.5, and the code is shown below:

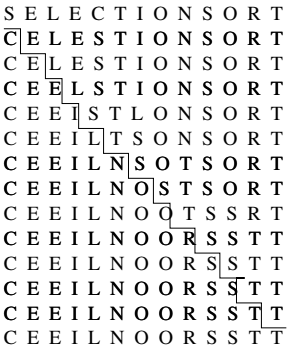

Figure 2.5: Animation of selection sort in action

```
selection_sort(int s[], int n)
{
        int i,j;                /* counters */
        int min;                /* index of minimum */

        for (i=0; i<n; i++) {
                min=i;
                for (j=i+1; j<n; j++)
                        if (s[j] < s[min]) min=j;
                swap(&s[i],&s[min]);
        }
}
```

The outer loop goes around n times. The nested inner loop goes around $n-i-1$ times, where i is the index of the outer loop. The exact number of times the *if* statement is executed is given by:

$$S(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} n - i - 1$$

What this sum is doing is adding up the integers in decreasing order starting from $n-1$, i.e.

$$S(n) = (n-1) + (n-2) + (n-3) + \ldots + 2 + 1$$

How can we reason about such a formula? We must solve the summation formula using the techniques of Section 1.3.5 (page 17) to get an exact value. But, with the Big Oh we are only interested in the *order* of the expression. One way to think about it is that we are adding up $n-1$ terms, whose average value is about $n/2$. This yields $S(n) \approx n(n-1)/2$.

Another way to think about it is in terms of upper and lower bounds. We have n terms at most, each of which is at most $n-1$. Thus, $S(n) \leq n(n-1) = O(n^2)$. We have $n/2$ terms each that are bigger than $n/2$. Thus $S(n) =\geq (n/2) \times (n/2) = \Omega(n^2)$. Together, this tells us that the running time is $\Theta(n^2)$, meaning that selection sort is quadratic.

2.5.2 Insertion Sort

A basic rule of thumb in Big Oh analysis is that worst-case running time follows from multiplying the largest number of times each nested loop can iterate. Consider the insertion sort algorithm presented on page 4, whose inner loops are repeated here:

```
for (i=1; i<n; i++) {
        j=i;
        while ((j>0) && (s[j] < s[j-1])) {
                swap(&s[j],&s[j-1]);
                j = j-1;
        }
}
```

How often does the inner *while* loop iterate? This is tricky because there are two different stopping conditions: one to prevent us from running off the bounds of the array ($j > 0$) and the other to mark when the element finds its proper place in sorted order ($s[j] < s[j-1]$). Since worst-case analysis seeks an upper bound on the running time, we ignore the early termination and assume that this loop *always* goes around i times. In fact, we can assume it *always* goes around n times since $i < n$. Since the outer loop goes around n times, insertion sort must be a quadratic-time algorithm, i.e. $O(n^2)$.

This crude "round it up" analysis always does the job, in that the Big Oh running time bound you get will always be correct. Occasionally, it might be too generous, meaning the actual worst case time might be of a lower order than implied by such analysis. Still, I strongly encourage this kind of reasoning as a basis for simple algorithm analysis.

2.5.3 String Pattern Matching

Pattern matching is the most fundamental algorithmic operation on text strings. This algorithm implements the find command available in any web browser or text editor:

Problem: Substring Pattern Matching
Input: A text string t and a pattern string p.
Output: Does t contain the pattern p as a substring, and if so where?

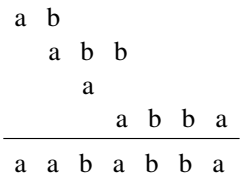

Figure 2.6: Searching for the substring *abba* in the text *aababba*.

Perhaps you are interested finding where "Skiena" appears in a given news article (well, I would be interested in such a thing). This is an instance of string pattern matching with t as the news article and p="Skiena."

There is a fairly straightforward algorithm for string pattern matching that considers the possibility that p may start at each possible position in t and then tests if this is so.

```
int findmatch(char *p, char *t)
{
        int i,j;                        /* counters */
        int m, n;                       /* string lengths */

        m = strlen(p);
        n = strlen(t);

        for (i=0; i<=(n-m); i=i+1) {
                j=0;
                while ((j<m) && (t[i+j]==p[j]))
                        j = j+1;
                if (j == m) return(i);
        }

        return(-1);
}
```

What is the worst-case running time of these two nested loops? The inner *while* loop goes around at most m times, and potentially far less when the pattern match fails. This, plus two other statements, lies within the outer *for* loop. The outer loop goes around at most $n - m$ times, since no complete alignment is possible once we get too far to the right of the text. The time complexity of nested loops multiplies, so this gives a worst-case running time of $O((n - m)(m + 2))$.

We did not count the time it takes to compute the length of the strings using the function *strlen*. Since the implementation of *strlen* is not given, we must guess how long it should take. If we explicitly count the number of characters until we

hit the end of the string; this would take time linear in the length of the string. This suggests that the running time should be $O(n + m + (n - m)(m + 2))$.

Let's use our knowledge of the Big Oh to simplify things. Since $m + 2 = \Theta(m)$, the "+2" isn't interesting, so we are left with $O(n + m + (n - m)m)$. Multiplying this out yields $O(n + m + nm - m^2)$, which still seems kind of ugly.

However, in any interesting problem we know that $n \geq m$, since it is impossible to have p as a substring of t for any pattern longer than the text itself. One consequence of this is that $n + m \leq 2n = \Theta(n)$. Thus our worst-case running time simplifies further to $O(n + nm - m^2)$.

Two more observations and we are done. First, note that $n \leq nm$, since $m \geq 1$ in any interesting pattern. Thus $n + nm = \Theta(nm)$, and we can drop the additive n, simplifying our analysis to $O(nm - m^2)$.

Finally, observe that the $-m^2$ term is negative, and thus only serves to lower the value within. Since the Big Oh gives an upper bound, we can drop any negative term without invalidating the upper bound. That $n \geq m$ implies that $mn \geq m^2$, so the negative term is not big enough to cancel any other term which is left. Thus we can simply express the worst-case running time of this algorithm as $O(nm)$.

After you get enough experience, you will be able to do such an algorithm analysis in your head without even writing the algorithm down. After all, algorithm design for a given task involves mentally rifling through different possibilities and selecting the best approach. This kind of fluency comes with practice, but if you are confused about why a given algorithm runs in $O(f(n))$ time, start by writing it out carefully and then employ the reasoning we used in this section.

2.5.4 Matrix Multiplication

Nested summations often arise in the analysis of algorithms with nested loops. Consider the problem of matrix multiplication:

Problem: Matrix Multiplication
Input: Two matrices, A (of dimension $x \times y$) and B (dimension $y \times z$).
Output: An $x \times z$ matrix C where $C[i][j]$ is the dot product of the ith row of A and the jth column of B.

Matrix multiplication is a fundamental operation in linear algebra, presented with an example in catalog in Section 13.3 (page 401). That said, the elementary algorithm for matrix multiplication is implemented as a tight product of three nested loops:

```
for (i=1; i<=x; i++)
        for (j=1; j<=y; j++) {
                C[i][j] = 0;
                for (k=1; k<=z; k++)
                        C[i][j] += A[i][k] * B[k][j];
        }
```

How can we analyze the time complexity of this algorithm? The number of multiplications $M(x, y, z)$ is given by the following summation:

$$M(x, y, z) = \sum_{i=1}^{x} \sum_{j=1}^{y} \sum_{k=1}^{z} 1$$

Sums get evaluated from the right inward. The sum of z ones is z, so

$$M(x, y, z) = \sum_{i=1}^{x} \sum_{j=1}^{y} z$$

The sum of y zs is just as simple, yz, so

$$M(x, y, z) = \sum_{i=1}^{x} yz$$

Finally, the sum of x yzs is xyz.

Thus the running of this matrix multiplication algorithm is $O(xyz)$. If we consider the common case where all three dimensions are the same, this becomes $O(n^3)$—i.e. , a cubic algorithm.

2.6 Logarithms and Their Applications

Logarithm is an anagram of algorithm, but that's not why we need to know what logarithms are. You've seen the button on your calculator but may have forgotten why it is there. A *logarithm* is simply an inverse exponential function. Saying $b^x = y$ is equivalent to saying that $x = \log_b y$. Further, this definition is the same as saying $b^{\log_b y} = y$.

Exponential functions grow at a distressingly fast rate, as anyone who has ever tried to pay off a credit card balance understands. Thus, inverse exponential functions—i.e. logarithms—grow refreshingly slowly. Logarithms arise in any process where things are repeatedly halved. We now look at several examples.

2.6.1 Logarithms and Binary Search

Binary search is a good example of an $O(\log n)$ algorithm. To locate a particular person p in a telephone book containing n names, you start by comparing p against the middle, or $(n/2)$nd name, say *Monroe, Marilyn*. Regardless of whether p belongs before this middle name (*Dean, James*) or after it (*Presley, Elvis*), after only one comparison you can discard one half of all the names in the book. The number of steps the algorithm takes equals the number of times we can halve n until only one name is left. By definition, this is exactly $\log_2 n$. Thus, twenty comparisons suffice to find any name in the million-name Manhattan phone book!

Binary search is one of the most powerful ideas in algorithm design. This power becomes apparent if we imagine being forced to live in a world with only unsorted

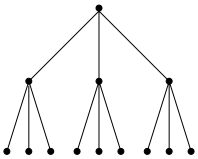

Figure 2.7: A height h tree with d children per node as d^h leaves. Here $h = 2$ and $d = 3$

telephone books. Figure 2.4 shows that $O(\log n)$ algorithms are fast enough to be used on problem instances of essentially unlimited size.

2.6.2 Logarithms and Trees

A binary tree of height 1 can have up to 2 leaf nodes, while a tree of height two can have up to four leaves. What is the height h of a rooted binary tree with n leaf nodes? Note that the number of leaves doubles every time we increase the height by one. To account for n leaves, $n = 2^h$ which implies that $h = \log_2 n$.

What if we generalize to trees that have d children, where $d = 2$ for the case of binary trees? A tree of height 1 can have up to d leaf nodes, while one of height two can have up to d^2 leaves. The number of possible leaves multiplies by d every time we increase the height by one, so to account for n leaves, $n = d^h$ which implies that $h = \log_d n$, as shown in Figure 2.7.

The punch line is that very short trees can have very many leaves, which is the main reason why binary trees prove fundamental to the design of fast data structures.

2.6.3 Logarithms and Bits

There are two bit patterns of length 1 (0 and 1) and four of length 2 (00, 01, 10, and 11). How many bits w do we need to represent any one of n different possibilities, be it one of n items or the integers from 1 to n?

The key observation is that there must be at least n different bit patterns of length w. Since the number of different bit patterns doubles as you add each bit, we need at least w bits where $2^w = n$—i.e., we need $w = \log_2 n$ bits.

2.6.4 Logarithms and Multiplication

Logarithms were particularly important in the days before pocket calculators. They provided the easiest way to multiply big numbers by hand, either implicitly using a slide rule or explicitly by using a book of logarithms.

Logarithms are still useful for multiplication, particularly for exponentiation. Recall that $\log_a(xy) = \log_a(x) + \log_a(y)$; i.e. , the log of a product is the sum of the logs. A direct consequence of this is

$$\log_a n^b = b \cdot \log_a n$$

So how can we compute a^b for any a and b using the $\exp(x)$ and $\ln(x)$ functions on your calculator, where $\exp(x) = e^x$ and $\ln(x) = \log_e(x)$? We know

$$a^b = \exp(\ln(a^b)) = \exp(b \ln a)$$

so the problem is reduced to one multiplication plus one call to each of these functions.

2.6.5 Fast Exponentiation

Suppose that we need to *exactly* compute the value of a^n for some reasonably large n. Such problems occur in primality testing for cryptography, as discussed in Section 13.8 (page 420). Issues of numerical precision prevent us from applying the formula above.

The simplest algorithm performs $n - 1$ multiplications, by computing $a \times a \times \ldots \times a$. However, we can do better by observing that $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$. If n is even, then $a^n = (a^{n/2})^2$. If n is odd, then $a^n = a(a^{\lfloor n/2 \rfloor})^2$. In either case, we have halved the size of our exponent at the cost of, at most, two multiplications, so $O(\lg n)$ multiplications suffice to compute the final value.

```
function power(a, n)
    if (n = 0) return(1)
    x = power(a, ⌊n/2⌋)
    if (n is even) then return(x^2)
        else return(a × x^2)
```

This simple algorithm illustrates an important principle of divide and conquer. It always pays to divide a job as evenly as possible. This principle applies to real life as well. When n is not a power of two, the problem cannot always be divided perfectly evenly, but a difference of one element between the two sides cannot cause any serious imbalance.

2.6.6 Logarithms and Summations

The *Harmonic numbers* arise as a special case of arithmetic progression, namely $H(n) = S(n, -1)$. They reflect the sum of the progression of simple reciprocals, namely,

$$H(n) = \sum_{i=1}^{n} 1/i \sim \ln n$$

Loss (apply the greatest)	Increase in level
(A) $2,000 or less	no increase
(B) More than $2,000	add 1
(C) More than $5,000	add 2
(D) More than $10,000	add 3
(E) More than $20,000	add 4
(F) More than $40,000	add 5
(G) More than $70,000	add 6
(H) More than $120,000	add 7
(I) More than $200,000	add 8
(J) More than $350,000	add 9
(K) More than $500,000	add 10
(L) More than $800,000	add 11
(M) More than $1,500,000	add 12
(N) More than $2,500,000	add 13
(O) More than $5,000,000	add 14
(P) More than $10,000,000	add 15
(Q) More than $20,000,000	add 16
(R) More than $40,000,000	add 17
(Q) More than $80,000,000	add 18

Figure 2.8: The Federal Sentencing Guidelines for fraud

The Harmonic numbers prove important because they usually explain "where the log comes from" when one magically pops out from algebraic manipulation. For example, the key to analyzing the average case complexity of Quicksort is the summation $S(n) = n \sum_{i=1}^{n} 1/i$. Employing the Harmonic number identity immediately reduces this to $\Theta(n \log n)$.

2.6.7 Logarithms and Criminal Justice

Figure 2.8 will be our final example of logarithms in action. This table appears in the Federal Sentencing Guidelines, used by courts throughout the United States. These guidelines are an attempt to standardize criminal sentences, so that a felon convicted of a crime before one judge receives the same sentence that they would before a different judge. To accomplish this, the judges have prepared an intricate point function to score the depravity of each crime and map it to time-to-serve.

Figure 2.8 gives the actual point function for fraud—a table mapping dollars stolen to points. Notice that the punishment increases by one level each time the amount of money stolen roughly doubles. That means that the level of punishment (which maps roughly linearly to the amount of time served) grows logarithmically with the amount of money stolen.

Think for a moment about the consequences of this. Many a corrupt CEO certainly has. It means that your total sentence grows *extremely* slowly with the amount of money you steal. Knocking off five liquor stores for $10,000 each will get you more time than embezzling $1,000,000 once. The corresponding benefit of stealing really large amounts of money is even greater. The moral of logarithmic growth is clear: *"If you are gonna do the crime, make it worth the time!"*

Take-Home Lesson: Logarithms arise whenever things are repeatedly halved or doubled.

2.7 Properties of Logarithms

As we have seen, stating $b^x = y$ is equivalent to saying that $x = \log_b y$. The b term is known as the *base* of the logarithm. Three bases are of particular importance for mathematical and historical reasons:

- *Base* $b = 2$ – The *binary logarithm*, usually denoted $\lg x$, is a base 2 logarithm. We have seen how this base arises whenever repeated halving (i.e., binary search) or doubling (i.e., nodes in trees) occurs. Most algorithmic applications of logarithms imply binary logarithms.

- *Base* $b = e$ – The *natural log*, usually denoted $\ln x$, is a base $e = 2.71828\ldots$ logarithm. The inverse of $\ln x$ is the exponential function $\exp(x) = e^x$ on your calculator. Thus, composing these functions gives us

$$\exp(\ln x) = x$$

- *Base* $b = 10$ – Less common today is the base-10 or *common logarithm*, usually denoted as $\log x$. This base was employed in slide rules and logarithm books in the days before pocket calculators.

We have already seen one important property of logarithms, namely that

$$\log_a(xy) = \log_a(x) + \log_a(y)$$

The other important fact to remember is that it is easy to convert a logarithm from one base to another. This is a consequence of the formula:

$$\log_a b = \frac{\log_c b}{\log_c a}$$

Thus, changing the base of $\log b$ from base-a to base-c simply involves dividing by $\log_c a$. It is easy to convert a common log function to a natural log function, and vice versa.

Two implications of these properties of logarithms are important to appreciate from an algorithmic perspective:

- *The base of the logarithm has no real impact on the growth rate-* Compare the following three values: $\log_2(1,000,000) = 19.9316$, $\log_3(1,000,000) = 12.5754$, and $\log_{100}(1,000,000) = 3$. A big change in the base of the logarithm produces little difference in the value of the log. Changing the base of the log from a to c involves dividing by $\log_c a$. This conversion factor is lost to the Big Oh notation whenever a and c are constants. Thus we are usually justified in ignoring the base of the logarithm when analyzing algorithms.

- *Logarithms cut any function down to size-* The growth rate of the logarithm of any polynomial function is $O(\lg n)$. This follows because

  $$\log_a n^b = b \cdot \log_a n$$

  The power of binary search on a wide range of problems is a consequence of this observation. Note that doing a binary search on a sorted array of n^2 things requires only twice as many comparisons as a binary search on n things.

  Logarithms efficiently cut any function down to size. It is hard to do arithmetic on factorials except for logarithms, since

  $$n! = \Pi_{i=1}^{n} i \rightarrow \log n! = \sum_{i=1}^{n} \log i = \Theta(n \log n)$$

  provides another way for logarithms to pop up in algorithm analysis.

Stop and Think: Importance of an Even Split

Problem: How many queries does binary search take on the million-name Manhattan phone book if each split was 1/3 to 2/3 instead of 1/2 to 1/2?

Solution: When performing binary searches in a telephone book, how important is it that each query split the book exactly in half? Not much. For the Manhattan telephone book, we now use $\log_{3/2}(1,000,000) \approx 35$ queries in the worst case, not a significant change from $\log_2(1,000,000) \approx 20$. The power of binary search comes from its logarithmic complexity, not the base of the log. ∎

2.8 War Story: Mystery of the Pyramids

That look in his eyes should have warned me even before he started talking.

"We want to use a parallel supercomputer for a numerical calculation up to 1,000,000,000, but we need a faster algorithm to do it."

I'd seen that distant look before. Eyes dulled from too much exposure to the raw horsepower of supercomputers—machines so fast that brute force seemed to eliminate the need for clever algorithms; at least until the problems got hard enough.

"I am working with a Nobel prize winner to use a computer on a famous problem in number theory. Are you familiar with Waring's problem?"

I knew some number theory. "Sure. Waring's problem asks whether every integer can be expressed at least one way as the sum of at most four integer squares. For example, $78 = 8^2 + 3^2 + 2^2 + 1^2 = 7^2 + 5^2 + 2^2$. I remember proving that four squares suffice to represent any integer in my undergraduate number theory class. Yes, it's a famous problem but one that got solved about 200 years ago."

"No, we are interested in a different version of Waring's problem. A *pyramidal number* is a number of the form $(m^3 - m)/6$, for $m \geq 2$. Thus the first several pyramidal numbers are 1, 4, 10, 20, 35, 56, 84, 120, and 165. The conjecture since 1928 is that every integer can be represented by the sum of at most five such pyramidal numbers. We want to use a supercomputer to prove this conjecture on all numbers from 1 to 1,000,000,000."

"Doing a billion of anything will take a substantial amount of time," I warned. "The time you spend to compute the minimum representation of each number will be critical, because you are going to do it one billion times. Have you thought about what kind of an algorithm you are going to use?"

"We have already written our program and run it on a parallel supercomputer. It works very fast on smaller numbers. Still, it takes much too much time as soon as we get to 100,000 or so."

Terrific, I thought. Our supercomputer junkie had discovered asymptotic growth. No doubt his algorithm ran in something like quadratic time, and he got burned as soon as n got large.

"We need a faster program in order to get to one billion. Can you help us? Of course, we can run it on our parallel supercomputer when you are ready."

I am a sucker for this kind of challenge, finding fast algorithms to speed up programs. I agreed to think about it and got down to work.

I started by looking at the program that the other guy had written. He had built an array of all the $\Theta(n^{1/3})$ pyramidal numbers from 1 to n inclusive.[2] To test each number k in this range, he did a brute force test to establish whether it was the sum of two pyramidal numbers. If not, the program tested whether it was the sum of three of them, then four, and finally five, until it first got an answer. About 45% of the integers are expressible as the sum of three pyramidal numbers. Most of the remaining 55% require the sum of four, and usually each of these can be represented in many different ways. Only 241 integers are known to require the sum of five pyramidal numbers, the largest being 343,867. For about half of the n numbers, this algorithm presumably went through all of the three-tests and at least

[2]Why $n^{1/3}$? Recall that pyramidal numbers are of the form $(m^3 - m)/6$. The largest m such that the resulting number is at most n is roughly $\sqrt[3]{6n}$, so there are $\Theta(n^{1/3})$ such numbers.

some of the four-tests before terminating. Thus, the total time for this algorithm would be at least $O(n \times (n^{1/3})^3) = O(n^2)$ time, where $n = 1{,}000{,}000{,}000$. No wonder his program cried "Uncle."

Anything that was going to do significantly better on a problem this large had to avoid explicitly testing all triples. For each value of k, we were seeking the smallest set of pyramidal numbers that add up to exactly to k. This problem is called the *knapsack problem*, and is discussed in Section 13.10 (page 427). In our case, the weights are the set of pyramidal numbers no greater than n, with an additional constraint that the knapsack holds exactly k items.

A standard approach to solving knapsack precomputes the sum of smaller subsets of the items for use in computing larger subsets. If we have a table of all sums of two numbers and want to know whether k is expressible as the sum of three numbers, we can ask whether k is expressible as the sum of a single number plus a number in this two-table.

Therefore I needed a table of all integers less than n that can be expressed as the sum of two of the 1,818 pyramidal numbers less than 1,000,000,000. There can be at most $1,818^2 = 3{,}305{,}124$ of them. Actually, there are only about half this many after we eliminate duplicates and any sum bigger than our target. Building a sorted array storing these numbers would be no big deal. Let's call this sorted data structure of all pair-sums the *two*-table.

To find the minimum decomposition for a given k, I would first check whether it was one of the 1,818 pyramidal numbers. If not, I would then check whether k was in the sorted table of the sums of two pyramidal numbers. To see whether k was expressible as the sum of three such numbers, all I had to do was check whether $k - p[i]$ was in the *two*-table for $1 \leq i \leq 1,818$. This could be done quickly using binary search. To see whether k was expressible as the sum of four pyramidal numbers, I had to check whether $k - two[i]$ was in the two-table for any $1 \leq i \leq |two|$. However, since almost every k was expressible in many ways as the sum of four pyramidal numbers, this test would terminate quickly, and the total time taken would be dominated by the cost of the threes. Testing whether k was the sum of three pyramidal numbers would take $O(n^{1/3} \lg n)$. Running this on each of the n integers gives an $O(n^{4/3} \lg n)$ algorithm for the complete job. Comparing this to his $O(n^2)$ algorithm for $n = 1{,}000{,}000{,}000$ suggested that my algorithm was a cool *30,000* times faster than his original!

My first attempt to code this solved up to $n = 1,000,000$ on my ancient Sparc ELC in about 20 minutes. From here, I experimented with different data structures to represent the sets of numbers and different algorithms to search these tables. I tried using hash tables and bit vectors instead of sorted arrays, and experimented with variants of binary search such as interpolation search (see Section 14.2 (page 441)). My reward for this work was solving up to n =1,000,000 in under three minutes, a factor of six improvement over my original program.

With the real thinking done, I worked to tweak a little more performance out of the program. I avoided doing a sum-of-four computation on any k when $k - 1$ was

the sum-of-three, since 1 is a pyramidal number, saving about 10% of the total run time using this trick alone. Finally, I got out my profiler and tried some low-level tricks to squeeze a little more performance out of the code. For example, I saved another 10% by replacing a single procedure call with in line code.

At this point, I turned the code over to the supercomputer guy. What he did with it is a depressing tale, which is reported in Section 7.10 (page 268).

In writing up this war story, I went back to rerun my program more than ten years later. On my desktop SunBlade 150, getting to 1,000,000 now took 27.0 seconds using the gcc compiler without turning on any compiler optimization. With Level 4 optimization, the job ran in just 14.0 seconds—quite a tribute to the quality of the optimizer. The run time on my desktop machine improved by a factor of about three over the four-year period prior to my first edition of this book, with an additional 5.3 times over the last 11 years. These speedups are probably typical for most desktops.

The primary lesson of this war story is to show the enormous potential for algorithmic speedups, as opposed to the fairly limited speedup obtainable via more expensive hardware. I sped his program up by about 30,000 times. His million-dollar computer had 16 processors, each reportedly five times faster on integer computations than the $3,000 machine on my desk. That gave a maximum potential speedup of less than 100 times. Clearly, the algorithmic improvement was the big winner here, as it is certain to be in any sufficiently large computation.

2.9 Advanced Analysis (*)

Ideally, each of us would be fluent in working with the mathematical techniques of asymptotic analysis. And ideally, each of us would be rich and good looking as well.

In this section I will survey the major techniques and functions employed in advanced algorithm analysis. I consider this optional material—it will not be used elsewhere in the textbook section of this book. That said, it will make some of the complexity functions reported in the Hitchhiker's Guide far less mysterious.

2.9.1 Esoteric Functions

The bread-and-butter classes of complexity functions were presented in Section 2.3.1 (page 39). More esoteric functions also make appearances in advanced algorithm analysis. Although we will not see them much in this book, it is still good business to know what they mean and where they come from:

- *Inverse Ackerman's function* $f(n) = \alpha(n)$ – This function arises in the detailed analysis of several algorithms, most notably the Union-Find data structure discussed in Section 6.1.3 (page 198).

  The exact definition of this function and why it arises will not be discussed further. It is sufficient to think of it as geek talk for the slowest-growing

complexity function. Unlike the constant function $f(n) = 1$, it eventually gets to infinity as $n \to \infty$, but it certainly takes its time about it. The value of $\alpha(n) < 5$ for any value of n that can be written in this physical universe.

- $f(n) = \log \log n$ – The "log log" function is just that—the logarithm of the logarithm of n. One natural example of how it might arise is doing a binary search on a sorted array of only $\lg n$ items.

- $f(n) = \log n / \log \log n$ – This function grows a little slower than $\log n$ because it is divided by an even slower growing function.

  To see where this arises, consider an n-leaf rooted tree of degree d. For binary trees, i.e. when $d = 2$, the height h is given

  $$n = 2^h \to h = \lg n$$

  by taking the logarithm of both sides of the equation. Now consider the height of such a tree when the degree $d = \log n$. Then

  $$n = (\log n)^h \to h = \log n / \log \log n$$

- $f(n) = \log^2 n$ – This is the product of log functions—i.e., $(\log n) \times (\log n)$. It might arise if we wanted to count the bits looked at in doing a binary search on n items, each of which was an integer from 1 to (say) n^2. Each such integer requires a $\lg(n^2) = 2 \lg n$ bit representation, and we look at $\lg n$ of them, for a total of $2 \lg^2 n$ bits.

  The "log squared" function typically arises in the design of intricate nested data structures, where each node in (say) a binary tree represents another data structure, perhaps ordered on a different key.

- $f(n) = \sqrt{n}$ – The square root is not so esoteric, but represents the class of "sublinear polynomials" since $\sqrt{n} = n^{1/2}$. Such functions arise in building d-dimensional grids that contain n points. A $\sqrt{n} \times \sqrt{n}$ square has area n, and an $n^{1/3} \times n^{1/3} \times n^{1/3}$ cube has volume n. In general, a d-dimensional hypercube of length $n^{1/d}$ on each side has volume d.

- $f(n) = n^{(1+\epsilon)}$ – Epsilon (ϵ) is the mathematical symbol to denote a constant that can be made arbitrarily small but never quite goes away.

  It arises in the following way. Suppose I design an algorithm that runs in $2^c n^{(1+1/c)}$ time, and I get to pick whichever c I want. For $c = 2$, this is $4n^{3/2}$ or $O(n^{3/2})$. For $c = 3$, this is $8n^{4/3}$ or $O(n^{4/3})$, which is better. Indeed, the exponent keeps getting better the larger I make c.

  The problem is that I cannot make c arbitrarily large before the 2^c term begins to dominate. Instead, we report this algorithm as running in $O(n^{1+\epsilon})$, and leave the best value of ϵ to the beholder.

2.9.2 Limits and Dominance Relations

The dominance relation between functions is a consequence of the theory of limits, which you may recall from Calculus. We say that $f(n)$ *dominates* $g(n)$ if $\lim_{n\to\infty} g(n)/f(n) = 0$.

Let's see this definition in action. Suppose $f(n) = 2n^2$ and $g(n) = n^2$. Clearly $f(n) > g(n)$ for all n, but it does not dominate since

$$\lim_{n\to\infty} g(n)/f(n) = \lim_{n\to\infty} n^2/2n^2 = \lim_{n\to\infty} 1/2 \neq 0$$

This is to be expected because both functions are in the class $\Theta(n^2)$. What about $f(n) = n^3$ and $g(n) = n^2$? Since

$$\lim_{n\to\infty} g(n)/f(n) = \lim_{n\to\infty} n^2/n^3 = \lim_{n\to\infty} 1/n = 0$$

the higher-degree polynomial dominates. This is true for any two polynomials, namely that n^a dominates n^b if $a > b$ since

$$\lim_{n\to\infty} n^b/n^a = \lim_{n\to\infty} n^{b-a} \to 0$$

Thus $n^{1.2}$ dominates $n^{1.1999999}$.

Now consider two exponential functions, say $f(n) = 3^n$ and $g(n) = 2^n$. Since

$$\lim_{n\to\infty} g(n)/f(n) = 2^n/3^n = \lim_{n\to\infty} (2/3)^n = 0$$

the exponential with the higher base dominates.

Our ability to prove dominance relations from scratch depends upon our ability to prove limits. Let's look at one important pair of functions. Any polynomial (say $f(n) = n^\epsilon$) dominates logarithmic functions (say $g(n) = \lg n$). Since $n = 2^{\lg n}$,

$$f(n) = (2^{\lg n})^\epsilon = 2^{\epsilon \lg n}$$

Now consider

$$\lim_{n\to\infty} g(n)/f(n) = \lg n/2^{\epsilon \lg n}$$

In fact, this does go to 0 as $n \to \infty$.

Take-Home Lesson: By interleaving the functions here with those of Section 2.3.1 (page 39), we see where everything fits into the dominance pecking order:

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\epsilon} \gg n \log n \gg n \gg \sqrt{n} \gg$$
$$\log^2 n \gg \log n \gg \log n/\log\log n \gg \log\log n \gg \alpha(n) \gg 1$$

Chapter Notes

Most other algorithm texts devote considerably more efforts to the formal analysis of algorithms than we have here, and so we refer the theoretically-inclined reader

elsewhere for more depth. Algorithm texts more heavily stressing analysis include [CLRS01, KT06].

The book *Concrete Mathematics* by Knuth, Graham, and Patashnik [GKP89] offers an interesting and thorough presentation of mathematics for the analysis of algorithms. Niven and Zuckerman [NZ80] is an nice introduction to number theory, including Waring's problem, discussed in the war story.

The notion of dominance also gives rise to the "Little Oh" notation. We say that $f(n) = o(g(n))$ iff $g(n)$ dominates $f(n)$. Among other things, the Little Oh proves useful for asking questions. Asking for an $o(n^2)$ algorithm means you want one that is better than quadratic in the worst case—and means you would be willing to settle for $O(n^{1.999} \log^2 n)$.

2.10 Exercises

Program Analysis

2-1. *[3]* What value is returned by the following function? Express your answer as a function of n. Give the worst-case running time using the Big Oh notation.

```
function mystery(n)
    r := 0
    for i := 1 to n − 1 do
        for j := i + 1 to n do
            for k := 1 to j do
                r := r + 1
    return(r)
```

2-2. *[3]* What value is returned by the following function? Express your answer as a function of n. Give the worst-case running time using Big Oh notation.

```
function pesky(n)
    r := 0
    for i := 1 to n do
        for j := 1 to i do
            for k := j to i + j do
                r := r + 1
    return(r)
```

2-3. *[5]* What value is returned by the following function? Express your answer as a function of n. Give the worst-case running time using Big Oh notation.

```
function prestiferous(n)
    r := 0
    for i := 1 to n do
        for j := 1 to i do
            for k := j to i + j do
                for l := 1 to i + j − k do
```

```
                    r := r + 1
            return(r)
```

2-4. *[8]* What value is returned by the following function? Express your answer as a function of n. Give the worst-case running time using Big Oh notation.

```
function conundrum(n)
    r := 0
    for i := 1 to n do
        for j := i + 1 to n do
            for k := i + j − 1 to n do
                r := r + 1
    return(r)
```

2-5. *[5]* Suppose the following algorithm is used to evaluate the polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

```
p := a_0;
xpower := 1;
for i := 1 to n do
        xpower := x * xpower;
        p := p + a_i * xpower
end
```

(a) How many multiplications are done in the worst-case? How many additions?

(b) How many multiplications are done on the average?

(c) Can you improve this algorithm?

2-6. *[3]* Prove that the following algorithm for computing the maximum value in an array $A[1..n]$ is correct.

```
function max(A)
                m := A[1]
                for i := 2 to n do
                        if A[i] > m then m := A[i]
                return (m)
```

Big Oh

2-7. *[3]* True or False?

(a) Is $2^{n+1} = O(2^n)$?

(b) Is $2^{2n} = O(2^n)$?

2-8. *[3]* For each of the following pairs of functions, either $f(n)$ is in $O(g(n))$, $f(n)$ is in $\Omega(g(n))$, or $f(n) = \Theta(g(n))$. Determine which relationship is correct and briefly explain why.

(a) $f(n) = \log n^2$; $g(n) = \log n + 5$

(b) $f(n) = \sqrt{n}$; $g(n) = \log n^2$

(c) $f(n) = \log^2 n$; $g(n) = \log n$

(d) $f(n) = n$; $g(n) = \log^2 n$

(e) $f(n) = n \log n + n$; $g(n) = \log n$

(f) $f(n) = 10$; $g(n) = \log 10$

(g) $f(n) = 2^n$; $g(n) = 10n^2$

(h) $f(n) = 2^n$; $g(n) = 3^n$

2-9. *[3]* For each of the following pairs of functions $f(n)$ and $g(n)$, determine whether $f(n) = O(g(n))$, $g(n) = O(f(n))$, or both.

(a) $f(n) = (n^2 - n)/2$, $g(n) = 6n$

(b) $f(n) = n + 2\sqrt{n}$, $g(n) = n^2$

(c) $f(n) = n \log n$, $g(n) = n\sqrt{n}/2$

(d) $f(n) = n + \log n$, $g(n) = \sqrt{n}$

(e) $f(n) = 2(\log n)^2$, $g(n) = \log n + 1$

(f) $f(n) = 4n \log n + n$, $g(n) = (n^2 - n)/2$

2-10. *[3]* Prove that $n^3 - 3n^2 - n + 1 = \Theta(n^3)$.

2-11. *[3]* Prove that $n^2 = O(2^n)$.

2-12. *[3]* For each of the following pairs of functions $f(n)$ and $g(n)$, give an appropriate positive constant c such that $f(n) \leq c \cdot g(n)$ for all $n > 1$.

(a) $f(n) = n^2 + n + 1$, $g(n) = 2n^3$

(b) $f(n) = n\sqrt{n} + n^2$, $g(n) = n^2$

(c) $f(n) = n^2 - n + 1$, $g(n) = n^2/2$

2-13. *[3]* Prove that if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

2-14. *[3]* Prove that if $f_1(N) = \Omega(g_1(n))$ and $f_2(n) = \Omega(g_2(n))$, then $f_1(n) + f_2(n) = \Omega(g_1(n) + g_2(n))$.

2-15. *[3]* Prove that if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$

2-16. *[5]* Prove for all $k \geq 1$ and all sets of constants $\{a_k, a_{k-1}, \ldots, a_1, a_0\} \in R$,

$$a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0 = O(n^k)$$

2-17. *[5]* Show that for any real constants a and b, $b > 0$

$$(n + a)^b = \Theta(n^b)$$

2-18. *[5]* List the functions below from the lowest to the highest order. If any two or more are of the same order, indicate which.

n	2^n	$n \lg n$	$\ln n$
$n - n^3 + 7n^5$	$\lg n$	$\sqrt{n}$	e^n
$n^2 + \lg n$	n^2	2^{n-1}	$\lg \lg n$
n^3	$(\lg n)^2$	$n!$	$n^{1+\varepsilon}$ where $0 < \varepsilon < 1$

2-19. *[5]* List the functions below from the lowest to the highest order. If any two or more are of the same order, indicate which.

$\sqrt{n}$	n	2^n
$n \log n$	$n - n^3 + 7n^5$	$n^2 + \log n$
n^2	n^3	$\log n$
$n^{\frac{1}{3}} + \log n$	$(\log n)^2$	$n!$
$\ln n$	$\frac{n}{\log n}$	$\log \log n$
$(1/3)^n$	$(3/2)^n$	6

2-20. *[5]* Find two functions $f(n)$ and $g(n)$ that satisfy the following relationship. If no such f and g exist, write "None."

(a) $f(n) = o(g(n))$ and $f(n) \neq \Theta(g(n))$

(b) $f(n) = \Theta(g(n))$ and $f(n) = o(g(n))$

(c) $f(n) = \Theta(g(n))$ and $f(n) \neq O(g(n))$

(d) $f(n) = \Omega(g(n))$ and $f(n) \neq O(g(n))$

2-21. *[5]* True or False?

(a) $2n^2 + 1 = O(n^2)$

(b) $\sqrt{n} = O(\log n)$

(c) $\log n = O(\sqrt{n})$

(d) $n^2(1 + \sqrt{n}) = O(n^2 \log n)$

(e) $3n^2 + \sqrt{n} = O(n^2)$

(f) $\sqrt{n} \log n = O(n)$

(g) $\log n = O(n^{-1/2})$

2-22. *[5]* For each of the following pairs of functions $f(n)$ and $g(n)$, state whether $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \Theta(g(n))$, or none of the above.

(a) $f(n) = n^2 + 3n + 4$, $g(n) = 6n + 7$

(b) $f(n) = n\sqrt{n}$, $g(n) = n^2 - n$

(c) $f(n) = 2^n - n^2$, $g(n) = n^4 + n^2$

2-23. *[3]* For each of these questions, briefly explain your answer.

(a) If I prove that an algorithm takes $O(n^2)$ worst-case time, is it possible that it takes $O(n)$ on some inputs?

(b) If I prove that an algorithm takes $O(n^2)$ worst-case time, is it possible that it takes $O(n)$ on all inputs?

(c) If I prove that an algorithm takes $\Theta(n^2)$ worst-case time, is it possible that it takes $O(n)$ on some inputs?

(d) If I prove that an algorithm takes $\Theta(n^2)$ worst-case time, is it possible that it takes $O(n)$ on all inputs?

(e) Is the function $f(n) = \Theta(n^2)$, where $f(n) = 100n^2$ for even n and $f(n) = 20n^2 - n\log_2 n$ for odd n?

2-24. *[3]* For each of the following, answer *yes*, *no*, or *can't tell*. Explain your reasoning.

(a) Is $3^n = O(2^n)$?

(b) Is $\log 3^n = O(\log 2^n)$?

(c) Is $3^n = \Omega(2^n)$?

(d) Is $\log 3^n = \Omega(\log 2^n)$?

2-25. *[5]* For each of the following expressions $f(n)$ find a simple $g(n)$ such that $f(n) = \Theta(g(n))$.

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

(b) $f(n) = \sum_{i=1}^{n} \lceil \frac{1}{i} \rceil$.

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

(d) $f(n) = \log(n!)$.

2-26. *[5]* Place the following functions into increasing asymptotic order.

$f_1(n) = n^2 \log_2 n$, $f_2(n) = n(\log_2 n)^2$, $f_3(n) = \sum_{i=0}^{n} 2^i$, $f_4(n) = \log_2(\sum_{i=0}^{n} 2^i)$.

2-27. *[5]* Place the following functions into increasing asymptotic order. If two or more of the functions are of the same asymptotic order then indicate this.

$f_1(n) = \sum_{i=1}^{n} \sqrt{i}$, $f_2(n) = (\sqrt{n}) \log n$, $f_3(n) = n\sqrt{\log n}$, $f_4(n) = 12n^{\frac{3}{2}} + 4n$,

2-28. *[5]* For each of the following expressions $f(n)$ find a simple $g(n)$ such that $f(n) = \Theta(g(n))$. (You should be able to prove your result by exhibiting the relevant parameters, but this is not required for the homework.)

(a) $f(n) = \sum_{i=1}^{n} 3i^4 + 2i^3 - 19i + 20$.

(b) $f(n) = \sum_{i=1}^{n} 3(4^i) + 2(3^i) - i^{19} + 20$.

(c) $f(n) = \sum_{i=1}^{n} 5^i + 3^{2i}$.

2-29. *[5]* Which of the following are true?

(a) $\sum_{i=1}^{n} 3^i = \Theta(3^{n-1})$.

(b) $\sum_{i=1}^{n} 3^i = \Theta(3^n)$.

(c) $\sum_{i=1}^{n} 3^i = \Theta(3^{n+1})$.

2-30. *[5]* For each of the following functions f find a simple function g such that $f(n) = \Theta(g(n))$.

(a) $f_1(n) = (1000)2^n + 4^n$.

(b) $f_2(n) = n + n\log n + \sqrt{n}$.

(c) $f_3(n) = \log(n^{20}) + (\log n)^{10}$.

(d) $f_4(n) = (0.99)^n + n^{100}$.

2-31. *[5]* For each pair of expressions (A, B) below, indicate whether A is O, o, Ω, ω, or Θ of B. Note that zero, one or more of these relations may hold for a given pair; list all correct ones.

	A	B
(a)	n^{100}	2^n
(b)	$(\lg n)^{12}$	$\sqrt{n}$
(c)	$\sqrt{n}$	$n^{\cos(\pi n/8)}$
(d)	10^n	100^n
(e)	$n^{\lg n}$	$(\lg n)^n$
(f)	$\lg(n!)$	$n \lg n$

Summations

2-32. *[5]* Prove that:

$$1^2 - 2^2 + 3^2 - 4^2 + \ldots + (-1)^{k-1}k^2 = (-1)^{k-1}k(k+1)/2$$

2-33. *[5]* Find an expression for the sum of the ith row of the following triangle, and prove its correctness. Each entry is the sum of the three entries directly above it. All non existing entries are considered 0.

				1				
			1	1	1			
		1	2	3	2	1		
	1	3	6	7	6	3	1	
1	4	10	16	19	16	10	4	1

2-34. *[3]* Assume that Christmas has n days. Exactly how many presents did my "true love" send me? (Do some research if you do not understand this question.)

2-35. *[5]* Consider the following code fragment.

```
for i=1 to n do
    for j=i to 2*i do
        output ‘‘foobar’’
```

Let $T(n)$ denote the number of times 'foobar' is printed as a function of n.

a. Express $T(n)$ as a summation (actually two nested summations).

b. Simplify the summation. Show your work.

2-36. *[5]* Consider the following code fragment.

```
for i=1 to n/2 do
    for j=i to n-i do
        for k=1 to j do
output ‘‘foobar’’
```

Assume n is even. Let $T(n)$ denote the number of times 'foobar' is printed as a function of n.

(a) Express $T(n)$ as three nested summations.

(b) Simplify the summation. Show your work.

2-37. *[6]* When you first learned to multiply numbers, you were told that $x \times y$ means adding x a total of y times, so $5 \times 4 = 5+5+5+5 = 20$. What is the time complexity of multiplying two n-digit numbers in base b (people work in base 10, of course, while computers work in base 2) using the repeated addition method, as a function of n and b. Assume that single-digit by single-digit addition or multiplication takes $O(1)$ time. (Hint: how big can y be as a function of n and b?)

2-38. *[6]* In grade school, you learned to multiply long numbers on a digit-by-digit basis, so that $127 \times 211 = 127 \times 1 + 127 \times 10 + 127 \times 200 = 26,397$. Analyze the time complexity of multiplying two n-digit numbers with this method as a function of n (assume constant base size). Assume that single-digit by single-digit addition or multiplication takes $O(1)$ time.

Logarithms

2-39. *[5]* Prove the following identities on logarithms:

(a) Prove that $\log_a(xy) = \log_a x + \log_a y$

(b) Prove that $\log_a x^y = y \log_a x$

(c) Prove that $\log_a x = \frac{\log_b x}{\log_b a}$

(d) Prove that $x^{\log_b y} = y^{\log_b x}$

2-40. *[3]* Show that $\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$

2-41. *[3]* Prove that that the binary representation of $n \geq 1$ has $\lfloor \lg_2 n \rfloor + 1$ bits.

2-42. *[5]* In one of my research papers I give a comparison-based sorting algorithm that runs in $O(n \log(\sqrt{n}))$. Given the existence of an $\Omega(n \log n)$ lower bound for sorting, how can this be possible?

Interview Problems

2-43. *[5]* You are given a set S of n numbers. You must pick a subset S' of k numbers from S such that the probability of each element of S occurring in S' is equal (i.e., each is selected with probability k/n). You may make only one pass over the numbers. What if n is unknown?

2-44. *[5]* We have 1,000 data items to store on 1,000 nodes. Each node can store copies of exactly three different items. Propose a replication scheme to minimize data loss as nodes fail. What is the expected number of data entries that get lost when three random nodes fail?

2-45. *[5]* Consider the following algorithm to find the minimum element in an array of numbers $A[0, \ldots, n]$. One extra variable *tmp* is allocated to hold the current minimum value. Start from A[0]; "tmp" is compared against $A[1]$, $A[2]$, $\ldots$, $A[N]$ in order. When $A[i] < tmp$, $tmp = A[i]$. What is the expected number of times that the assignment operation $tmp = A[i]$ is performed?

2-46. *[5]* You have a 100-story building and a couple of marbles. You must identify the lowest floor for which a marble will break if you drop it from this floor. How fast can you find this floor if you are given an infinite supply of marbles? What if you have only two marbles?

2-47. *[5]* You are given 10 bags of gold coins. Nine bags contain coins that each weigh 10 grams. One bag contains all false coins that weigh one gram less. You must identify this bag in just one weighing. You have a digital balance that reports the weight of what is placed on it.

2-48. *[5]* You have eight balls all of the same size. Seven of them weigh the same, and one of them weighs slightly more. How can you find the ball that is heavier by using a balance and only two weighings?

2-49. *[5]* Suppose we start with n companies that eventually merge into one big company. How many different ways are there for them to merge?

2-50. *[5]* A *Ramanujam number* can be written two different ways as the sum of two cubes—i.e. , there exist distinct a, b, c, and d such that $a^3 + b^3 = c^3 + d^3$. Generate all Ramanujam numbers where $a, b, c, d < n$.

2-51. *[7]* Six pirates must divide $300 dollars among themselves. The division is to proceed as follows. The senior pirate proposes a way to divide the money. Then the pirates vote. If the senior pirate gets at least half the votes he wins, and that division remains. If he doesn't, he is killed and then the next senior-most pirate gets a chance to do the division. Now you have to tell what will happen and why (i.e. , how many pirates survive and how the division is done)? All the pirates are intelligent and the first priority is to stay alive and the next priority is to get as much money as possible.

2-52. *[7]* Reconsider the pirate problem above, where only one indivisible dollar is to be divided. Who gets the dollar and how many are killed?

Programming Challenges

These programming challenge problems with robot judging are available at *http://www.programming-challenges.com* or *http://online-judge.uva.es*.

2-1. "Primary Arithmetic" – Programming Challenges 110501, UVA Judge 10035.

2-2. "A Multiplication Game" – Programming Challenges 110505, UVA Judge 847.

2-3. "Light, More Light" – Programming Challenges 110701, UVA Judge 10110.