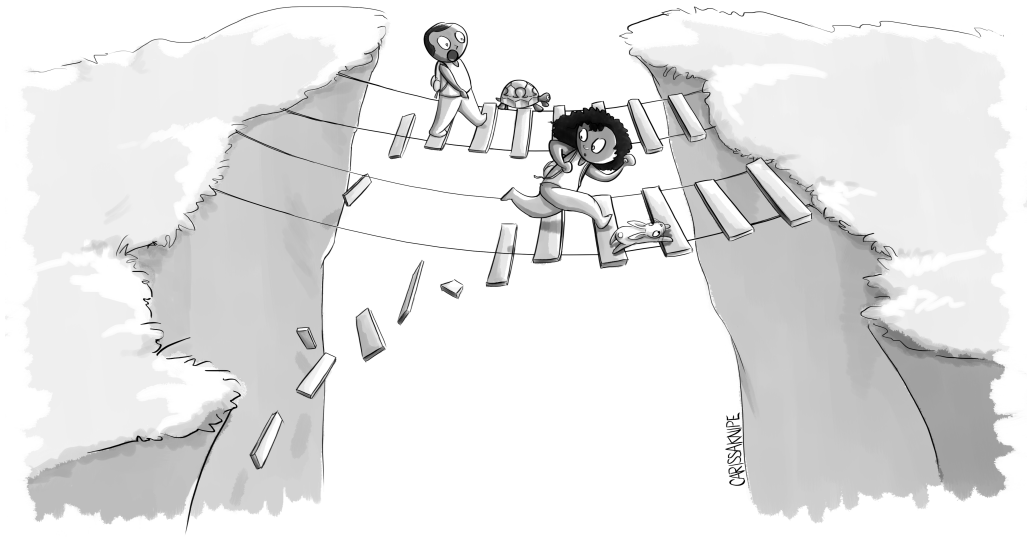
6 Analysis of Algorithms



In which our heroes stay beyond the reach of danger, by calculating precise bounds on how quickly they must move to stay safe.

6.1 Why You Might Care

> It was the best of times; it was the worst of times.
>
> Charles Dickens (1812–1870)
> *A Tale of Two Cities* (1859)

Computer scientists are speed demons. When we are confronted by a computational problem that we need to solve, we want to solve that problem as quickly as possible. That "need for speed" has driven much of the advancement in computation over the last fifty years. We discover faster ways of solving important problems: developing data structures that support apparently instantaneous search of billions of tweets or billions of users on a social networking site; or discovering new, faster algorithms that solve practical problems—such as finding shorter routes for delivery drivers or encrypting packets to be sent over the Internet. (Of course, the advances over the last fifty years have also been driven by improvements in computer hardware that ensure that *everything* we do computationally is faster!)

This chapter will introduce *asymptotic analysis,* the most common way in which computer scientists compare the speed of two possible solutions to the same problem. The basic idea is to think about the *rate of growth* of the running time of an algorithm—how much slower does the algorithm get if we double the size of the input?—in doing this analysis. We will think about "big" inputs to analyze the relative performance of the two algorithms, focusing on the long-run behavior instead of any small-input-size special cases for which one algorithm happens to perform exceptionally well. For the CS speed demon, asymptotic analysis is the speedometer. (Sometimes, instead of time, we measure the amount of space/memory or power/energy that an algorithm consumes.) We'll start in Section 6.2 with the key definitions through which we'll think about the rates of growth of functions, specifically by introducing "big oh" notation and its relatives (which formally describe what it means for one function to grow faster or slower than another). We'll start to discuss how we apply these definitions to analyze the efficiency of algorithms in Section 6.3. In Section 6.4, we'll look at how to analyze the speed of *recursive* algorithms—and then at a specific, and particularly common, type of "divide and conquer" recursive algorithm in Section 6.5.

To take one example of why this kind of analysis of running time matters, consider sorting an n-element array A. One approach is to use brute force: try all $n!$ different permutations of A, and select the one permutation whose elements are in ascending order. Sorting algorithms like Selection Sort, Insertion Sort, or Bubble Sort require $\approx c \cdot n^2$ operations, for some constant c, to sort A. You may also have seen Merge Sort,

	$n = 10$	$n = 100$	$n = 1000$	$n = 10{,}000$	maximum n solvable in one minute on a machine that completes 1,000,000,000 operations per second
$n \log n$	33	664	9966	132,877	1.94×10^9
n^2	100	10,000	1,000,000	100,000,000	244,949
$n!$	3,628,800	9.333×10^{157}	4.029×10^{2567}	$2.846 \times 10^{35{,}659}$	13

Figure 6.1 The number of operations for several algorithms, on several input sizes.

which requires $\approx c \cdot n \log n$ operations. (We'll review these sorting algorithms in Section 6.3.) Figure 6.1 shows the number of operations required by these algorithms ($n!$, n^2, and $n \log n$). Given that some estimates say that the Earth will be swallowed by the sun in merely a few billion years [9], there is plenty of reason to care about the differences in these running times. Asymptotic analysis is the first-cut approximation to making sure that our algorithms are fast enough—and that they will finish running while we're still around to view the output.

6.2 Asymptotics

> If everything seems under control, you're just not going fast enough.
>
> Mario Andretti (b. 1940)

We'll start by developing precise definitions related to the growth of functions—keeping in mind that the functions of primary interest will be those representing the running time of an algorithm based on its input size. Generally speaking, we will be interested in the behavior of algorithms *ignoring constant factors*. There are two different senses in which we ignore constants. First, we will ignore constant multiplicative factors; for our purposes, the function $f(n)$ and the function $g(n) = 2 \cdot f(n)$ "grow at the same rate." (Exercises 6.1–6.4 explore why we might evaluate efficiency of algorithms in this way.) Second, we will be interested in the long-run behavior of our algorithms, so we won't be concerned by any small input values for which the algorithm performs particularly quickly or slowly.

Example 6.1: All of these things are quite the same.
The following three functions all grow at the same rate:

$$f(n) = 3 \cdot n^2 \qquad g(n) = 0.01 \cdot n^2 \qquad h(n) = \begin{cases} 202 & \text{if } 0 < n < 100 \\ n^7 & \text{if } 100 \leq n < 1000 \\ 1776 \cdot n^2 & \text{otherwise.} \end{cases}$$

The functions f and g differ by a multiplicative factor. For $n \geq 1000$, the function h also differs by a constant multiplier from f and g; therefore for large enough n it too grows at the same rate as f and g.

This type of analysis is called *asymptotic analysis*.

> **Taking it further:** The name *asymptotic* comes from the Greek: *a* "without" + *symptotos* "falling together." In math, the *asymptote* of a function $f(n)$ is a line that $f(n)$ approaches as n gets very large. (Formally, this value is $\lim_{n\to\infty} f(n)$.) For example, the function $f(x) = \frac{1}{x}$ has an asymptote at 0: as x gets larger and larger, $f(x)$ gets closer and closer to 0. (Mathematicians also consider asymptotes where a function approaches, but does not reach, some particular value as the input approaches some point; for example, $\tan(\theta)$ has an asymptote of ∞ as $\theta \to \frac{\pi}{2}$ and $f(x) = \frac{-x}{x-2}$ has an asymptote of $-\infty$ as $x \to 2$ from below.) The asymptotic behavior of a function is similarly motivated: we're thinking about the growth rate of the function as n gets very large.

Consider two functions $f : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ and $g : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$. (We will be interested in functions whose domain and range are both nonnegative because we're primarily thinking about functions that describe the number of steps of a particular algorithm on an input of a particular size, and neither input size nor number of computational steps executed can be negative.) The key concept of asymptotic analysis will be a definition of the *growth rates* of the functions f and g, and how those growth rates compare: that is, what it means to say that f grows faster than g (or, really, no slower than g); or that f grows at the same rate as g; or that f grows slower (or no faster) than g.

6.2.1 Big O

We'll start by defining what it means for the function $f(n)$ to grow no faster than the function $g(n)$, written $f(n) = O(g(n))$. (Note: O is pronounced "big oh.")

Definition 6.1: "Big O" [O].
Consider two functions $f : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ and $g : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$. We say that *$f$ grows no faster than g* if there exist constants $c > 0$ and $n_0 \geq 0$ such that

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

In this case, we write "$f(n)$ is $O(g(n))$" or "$f(n) = O(g(n))$."

Taking it further: The "=" in "$f(n) = O(g(n))$" is odd notation, but it's also very standard. This expression means *$f(n)$ has the property of being $O(g(n))$* and not *$f(n)$ is identical to $O(g(n))$*. Philosophers sometimes distinguish between *the "is" of identity* and *the "is" of predication*. In a sentence like *Barbara Liskov is the 2008 Turing Award winner*, we are asserting that *Barbara Liskov* and *the 2008 Turing Award Winner* actually refer to the same thing—that is, they are identical. In a sentence like *Barbara Liskov is tall*, we are asserting that Barbara Liskov (the entity to which *Barbara Liskov* refers) has the property of being tall—that is, the predicate *x is tall* is true of Barbara Liskov. You should interpret the "=" in $f(n) = O(g(n))$ as an "is of predication." One reasonably accurate way to distinguish these two uses of *is* is by considering what happens if you reverse the order of the sentence: *The 2008 Turing Award Winner is Barbara Liskov* is still a (true) well-formed sentence, but *Tall is Barbara Liskov* sounds very strange. Similarly, for an "is of identity" in a mathematical context, we can say either $x^2 - 1 = (x+1)(x-1)$ or $(x+1)(x-1) = x^2 - 1$. But, while "$f(n) = O(g(n))$" is a well-formed statement, it is nonsensical to say "$O(g(n)) = f(n)$."

The intuition is that $f(n) = O(g(n))$ if, for all large enough n, we have $f(n) \leq \textit{constant} \cdot g(n)$. We get to choose what counts as "large enough" and we also get to choose the value of *constant*.

Figure 6.2 shows five different functions $f : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ that all satisfy $f(n) = O(n)$. (In the figure, the value of x is "large enough" once x is outside of the shaded box, and the multiplicative constant is equal to 3 in each subplot. For a function like $f(x) = 4x$, we'd show that $f(n) = O(n)$ by choosing some $c \geq 4$ as the multiplicative constant.) More quantitatively, here are two examples of functions that are $O(n^2)$:

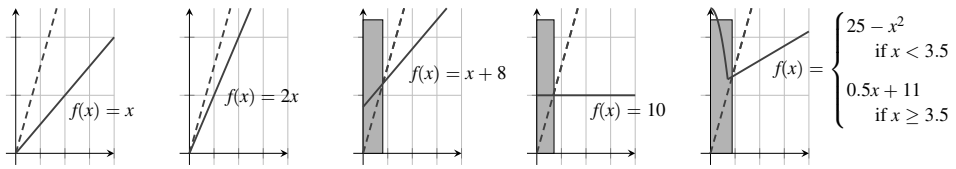
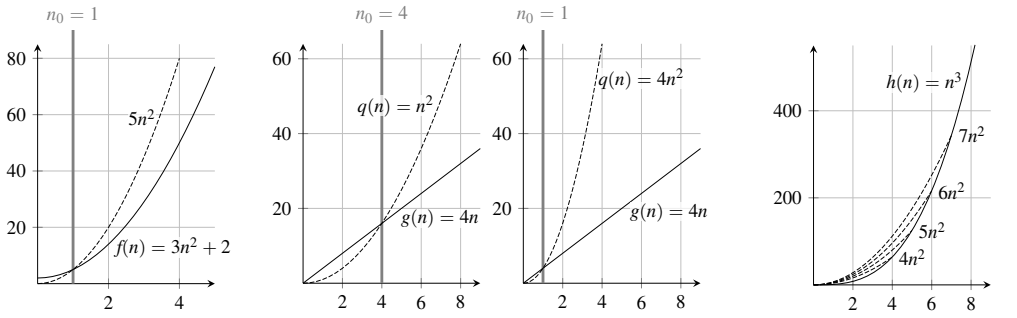


Figure 6.2 Five functions that are all $O(n)$. For any x beyond the shaded box, we have $f(x) \leq 3x$. (The first two functions satisfy $f(x) \leq 3x$ for *all* nonnegative x, so there's no shaded box necessary.) The dashed line corresponds to $3x$.



(a) $f(n)$ is $O(n^2)$ with $c = 5$ and $n_0 = 1$.

(b) $g(n)$ is $O(n^2)$ with $c = 1$ and $n_0 = 4$. Or, $g(n)$ is $O(n^2)$ with $c = 4$ and $n_0 = 1$.

(c) No value of c has $n^3 < cn^2$ for all large n.

Figure 6.3 The functions $f(n) = 3n^2 + 2$ and $g(n) = 4n$ are $O(n^2)$; $h(n) = n^3$ is not.

Example 6.2: A square function.
Prove that the function $f(n) = 3n^2 + 2$ is $O(n^2)$.

Solution. We must identify constants $c > 0$ and $n_0 \geq 0$ such that $\forall n \geq n_0 : 3n^2 + 2 \leq c \cdot n^2$. For all $n \geq 1$, observe that $2n^2 \geq 2$. Therefore, for all $n \geq 1$, we have

$$f(n) = 3n^2 + 2 \leq 3n^2 + 2n^2 = 5n^2.$$

Thus we can select $c = 5$ and $n_0 = 1$. (See Figure 6.3a.)

Example 6.3: Another square function.
Prove that the function $g(n) = 4n$ is also $O(n^2)$.

Solution. We wish to show that $4n \leq c \cdot n^2$ for all $n \geq n_0$, for constants $c > 0$ and $n_0 \geq 0$ that we get to choose. The two functions $g(n)$ and $q(n) = n^2$ are shown in Figure 6.3b. Because the functions cross (with no constant multiplier), we can pick $c = 1$. Observe that $4n \leq n^2$ whenever $n \geq 4$. (Or when $n \leq 0$: factoring $n^2 - 4n = n(n-4)$ shows that the functions $4n$ and n^2 cross at $n = 0$ and $n = 4$.) Thus $c = 1$ and $n_0 = 4$ suffice.

Note that, when $f(n) = O(g(n))$, there are many different choices of c and n_0 that satisfy the definition. For example, we could have chosen $c = 4$ and $n_0 = 1$ in Example 6.3 (again see Figure 6.3b), or $c = 2$ and $n_0 = 2$, or $c = 1$ and $n_0 = 128$, or lots of other values. (See Exercise 6.15.)

Example 6.4: One nonsquare.
Prove that the function $h(n) = n^3$ is *not* $O(n^2)$.

Solution. We need to argue that, for *all* constants n_0 and c, there exists an $n \geq n_0$ such that $h(n) > c \cdot n^2$. (See Figure 6.3c.) We'll proceed by contradiction. Imagine for a moment that we've identified values of n_0 and c that we hope satisfy the definition of $h(n) = O(n^2)$. But now define $m = \max(n_0, c+1)$. Then:

- $m \geq n_0$ (because we chose $m \geq n_0$), and
- $m^3 > cm^2$ (because we chose $m > c$, and multiplying both sides by m^2 yields $m^3 > cm^2$).

That means that we have found a value of $n \geq n_0$ such that $n^3 > c \cdot n^2$. Thus these values of n_0 and c do not satisfy the $O(\cdot)$ definition. Because n_0 and c were generic, we have shown that *no* such constants satisfying the condition $\forall n \geq n_0 : h(n) \leq cn^2$ can exist. Therefore $h(n) = n^3$ is *not* $O(n^2)$.

Some properties of $O(\cdot)$

Now that we've seen a few specific examples, let's turn to some more general results. There are many useful properties of $O(\cdot)$ that will come in handy later; we'll start here with a few of these properties, together with a proof of one.

Lemma 6.2: Asymptotic equivalence of max and sum.
We have $f(n) = O(g(n) + h(n))$ if and only if $f(n) = O(\max(g(n), h(n)))$.

Proof. We proceed by mutual implication.

For the forward direction, suppose that $f(n) = O(g(n) + h(n))$. Then by definition there exist constants $c > 0$ and $n_0 \geq 0$ such that

$$\begin{aligned}
\forall n \geq n_0 : \quad f(n) &\leq c \cdot [g(n) + h(n)] \\
&\leq c \cdot [\max(g(n), h(n)) + h(n)] && \text{for any } y, \text{ we have } g(n) \leq \max(g(n), y) \\
&\leq c \cdot [\max(g(n), h(n)) + \max(g(n), h(n))] && \text{for any } x, \text{ we have } h(n) \leq \max(x, h(n)) \\
&= 2c \cdot \max(g(n), h(n)).
\end{aligned}$$

We've now shown that $\forall n \geq n_0 : f(n) \leq 2c \cdot \max(g(n), h(n))$, and this statement just *is* the definition of $f(n) = O(\max(g(n), h(n)))$, using constants $n_0' = n_0$ and $c' = 2c$.

For the converse direction, suppose that $f(n) = O(\max(g(n), h(n)))$. Then there exist constants $c > 0$ and $n_0 \geq 0$ such that

$$\begin{aligned} \forall n \geq n_0 : \quad f(n) &\leq c \cdot \max(g(n), h(n)) \\ &\leq c \cdot [\max(g(n), h(n)) + \min(g(n), h(n))] && x \leq x + y \text{ for any value of } y \geq 0 \\ &= c \cdot [g(n) + h(n)] \,. && \min(x,y) + \max(x,y) = x + y \text{ for any values of } x \text{ and } y \end{aligned}$$

We've now shown that $\forall n \geq n_0 : f(n) \leq c[g(n) + h(n)]$, and this statement again just is the definition of $f(n) = O(g(n) + h(n))$, using the same constants $n_0' = n_0$ and $c' = c$. □

Problem-solving tip: Don't force yourself to prove more than you have to! For example, when proving that an asymptotic relationship like $f(n) = O(g(n))$ holds, all we need to do is identify *some* pair of constants c and n_0 that satisfy Definition 6.1. Don't work too hard! Choose whatever c or n_0 makes your life easiest, even if they're much bigger than necessary. For asymptotic purposes, we care that the constants c and n_0 *exist*, but we *don't* care how big they are.

Here are statements of a few other useful facts about $O(\cdot)$. (The proofs are left to Exercises 6.18–6.20.)

Lemma 6.3: Transitivity of $O(\cdot)$.
If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

Lemma 6.4: Addition and multiplication preserve $O(\cdot)$-ness.
If $f(n) = O(h_1(n))$ and $g(n) = O(h_2(n))$, then

$$f(n) + g(n) = O(h_1(n) + h_2(n)) \qquad \text{and} \qquad f(n) \cdot g(n) = O(h_1(n) \cdot h_2(n)).$$

Asymptotics of polynomials

So far, we've discussed properties of $O(\cdot)$ that are general with respect to the form of the functions in question. But because we're typically concerned with $O(\cdot)$ in the context of the running time of algorithms—and we are generally interested in algorithms that are efficient—we'll be particularly interested in the asymptotics of polynomials. The most salient point about the growth of a polynomial $p(n)$ is that $p(n)$'s asymptotic behavior is determined by the degree of $p(n)$—that is, the polynomial $p(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_k n^k$ behaves like n^k, asymptotically:

Lemma 6.5: Asymptotics of polynomials.
Let $p(n) = \sum_{i=0}^{k} a_i n^i$ be a polynomial. Then $p(n) = O(n^k)$.

(If $a_k > 0$, then indeed $p(n) = O(n^k)$, and it is not possible to improve this bound—that is, in the notation of Section 6.2.2, we have that $p(n) = \Theta(n^k)$.)

The proof of Lemma 6.5 is deferred to Exercise 6.21, but we have already seen the intuition in previous examples: every term $a_i n^i$ satisfies $a_i n^i \leq |a_i| \cdot n^k$, for any $n \geq 1$.

Asymptotics of logarithms and exponentials

We will also often encounter logarithms and exponential functions, so it's worth identifying a few of their asymptotic properties. Again, we'll prove one of these properties as an example, and leave proofs of many of the remaining properties to the exercises. The first pair of properties is that logarithmic functions grow more slowly than polynomials, which grow more slowly than exponential functions:

Lemma 6.6: $\log n$ grows slower than $n^{0.0000001}$.
Let $\epsilon > 0$ be an arbitrary constant, and let $f(n) = \log n$. Then $f(n) = O(n^{\epsilon})$.

Lemma 6.7: $n^{1000000}$ grows slower than 1.0000001^n.
Let $b > 1$ and $k \geq 0$ be any constants, and let $p(n) = \sum_{i=0}^{k} a_i n^i$ be any polynomial. Then $p(n) = O(b^n)$.

The second pair of properties is that logarithmic functions $\log_a n$ and $\log_b n$ grow at the same rate (for any bases $a > 1$ and $b > 1$) but that exponential functions a^n and b^n do not (for any bases a and $b \neq a$):

Lemma 6.8: The base of a logarithm doesn't matter, asymptotically.
Let $b > 1$ and $k > 0$ be arbitrary constants. Then $f(n) = \log_b(n^k)$ is $O(\log n)$.

Proof of Lemma 6.8. Using standard facts about logarithms, we have that

$$\begin{aligned} \log_b(n^k) &= k \cdot \log_b(n) && \text{Theorem 2.10.5: } \log_b x^y = y \log_b x \\ &= k \cdot \tfrac{\log n}{\log b}. && \text{Theorem 2.10.6 (change of base formula): } \log_b x = \tfrac{\log_c x}{\log_c b} \end{aligned}$$

For any $n \geq 1$, then, we have that $f(n) = \frac{k}{\log b} \cdot \log n$. Therefore, we have that $f(n) = O(\log n)$, using the constants $n_0 = 1$ and $c = \frac{k}{\log b}$. □

Lemma 6.9: The base of an exponential *does* matter, asymptotically.
Let $b \geq 1$ and $c \geq 1$ be arbitrary constants. Then $f(n) = b^n$ is $O(c^n)$ if and only if $b \leq c$.

Lemma 6.8 is the reason that, for example, binary search's running time is generally described as $O(\log n)$ rather than as $O(\log_2 n)$, without any concern for writing the "2": the base of the logarithm is inconsequential asymptotically, so $O(\log_{\sqrt{2}} n)$ and $O(\log_2 n)$ and $O(\ln n)$ all mean exactly the same thing. In contrast, for exponential functions, the base of the exponent *does* affect the asymptotic behavior: Lemma 6.9 says that, for example, the functions $f(n) = 2^n$ and $g(n) = (\sqrt{2})^n$ do *not* grow at the same rate. (See Exercises 6.25–6.28.)

> **Taking it further:** Generally, exponential growth is a problem for computer scientists. Many computational problems that are important and useful to solve seem to require searching a very large space of possible answers: for example, testing the satisfiability of an n-variable logical proposition seems to require looking at about 2^n different truth assignments, and factoring an n-digit number seems to require looking at about 10^n different candidate divisors. The fact that exponential functions grow so quickly is exactly why we do not have algorithms that are practical for even moderately large instances of these problems.

(See p. 3-32.) But one of the most famous exponentially growing functions actually *helps* us to solve problems: the amount of computational power available to a "standard" user of a computer has been growing exponentially for decades: about every 18 months, the processing power of a standard computer has roughly doubled. This trend—dubbed *Moore's Law,* after Gordon Moore, the co-founder of Intel—is discussed on p. 6-16.

6.2.2 Other Asymptotic Relationships: Ω, Θ, ω, and o

There are several basic asymptotic notions (with accompanying notation), based around two core ideas (see Figure 6.4):

$f(n)$ *grows no faster than* $g(n)$. In other words, ignoring small inputs, for all n we have that $f(n) \leq constant \cdot g(n)$. This relationship is expressed by the $O(\cdot)$ notation: $f(n) = O(g(n))$. We can also say that g is an *asymptotic upper bound* for f: if we plot n against $f(n)$ and $g(n)$, then $g(n)$ will be "above" $f(n)$ for large inputs.

$f(n)$ *grows no slower than* $g(n)$. The opposite relationship, in which g is an *asymptotic lower bound* on f, is expressed by $\Omega(\cdot)$ notation. (Ω is the Greek letter Omega written in upper case; ω, which we'll see soon, is the same Greek letter written in lower case.) Again, ignoring small inputs, $f(n) = \Omega(g(n))$ if for all n we have that $f(n) \geq constant \cdot g(n)$. (Notice that the inequality swapped directions from the definition of $O(\cdot)$.)

Formal definitions

Here are the formal definitions of four other relationships based on these notions. (This notation is summarized in Figure 6.4.)

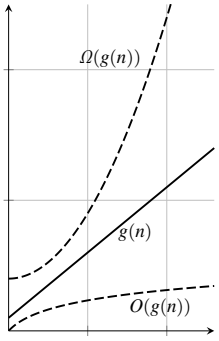
Definition 6.10: "Big Omega" [Ω].
A function f *grows no slower than* g, written $f(n) = \Omega(g(n))$, if there exist constants $d > 0$ and $n_0 \geq 0$ such that $\forall n \geq n_0 : f(n) \geq d \cdot g(n)$.

The two fundamental asymptotic relationships, $O(\cdot)$ and $\Omega(\cdot)$, are dual notions; they are related by the property that $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$. (The proof is left as Exercise 6.30.)

There are three other pieces of asymptotic notation, corresponding to the situations in which $f(n)$ is both $O(g)$ and $\Omega(g)$, or $O(g)$ but not $\Omega(g)$, or $\Omega(g)$ but not $O(g)$:

Definition 6.11: "Big Theta" [Θ].
A function f *grows at the same rate as* g, written $f(n) = \Theta(g(n))$, if $f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$.

Definition 6.12: "Little o" [o].
A function f *grows (strictly) slower than* g, written $f(n) = o(g(n))$, if $f(n) = O(g(n))$ but $f(n) \neq \Omega(g(n))$.



(a) A visualization.

$\exists c > 0, n_0 \geq 0$ such that $\forall n \geq n_0 : f(n) \leq c \cdot g(n)$ — f grows no faster than g

$\exists d > 0, n_0 \geq 0$ such that $\forall n \geq n_0 : f(n) \geq d \cdot g(n)$ — f grows no slower than g

	f grows no faster than g	f grows no slower than g	
$f(n) = O(g(n))$	*yes*	*don't care*	f grows no faster than g
$f(n) = \Omega(g(n))$	*don't care*	*yes*	f grows no slower than g
$f(n) = \Theta(g(n))$	*yes*	*yes*	f grows at the same rate as g
$f(n) = o(g(n))$	*yes*	*no*	f grows strictly slower than g
$f(n) = \omega(g(n))$	*no*	*yes*	f grows strictly faster than g

(b) A summary of the O, Ω, Θ, o, and ω asymptotic notation.

Figure 6.4 Two different ways to summarize asymptotic notation.

Definition 6.13: "Little omega" [ω].
A function f *grows (strictly) faster than* g, written $f(n) = \omega(g(n))$, if $f(n) = \Omega(g(n))$ but $f(n) \neq O(g(n))$.

Example 6.5: $f = \_\_\_(n)$.
Let $f(n) = 3n^2 + 1$. Is $f(n) = O(n)$? $\Omega(n)$? $\Theta(n)$? $o(n)$? $\omega(n)$? Prove your answers.

Solution. Once we determine whether $f(n) = O(n)$ and whether $f(n) = \Omega(n)$, we can answer all parts of the question using Figure 6.4.

Claim 1: $f(n) = \Omega(n)$. For any $n \geq 1$, we have that $n \leq n^2 \leq 3n^2 + 1 = f(n)$. Thus selecting $d = 1$ and $n_0 = 1$ satisfies Definition 6.10.

Claim 2: $f(n) \neq O(n)$. Consider any $c > 0$. For any $n \geq \frac{c}{3}$, we have that $3n^2 + 1 > 3n^2 \geq c \cdot n$. Therefore, for any $n_0 > 0$, there exists an $n \geq n_0$ such that $f(n) > c \cdot n$. (Namely, let $n = \max(n_0, \frac{c}{3})$. This value of n satisfies $n \geq n_0$ and $f(n) > c \cdot n$.) Thus, every $c > 0$ and $n_0 \geq 0$ fail to satisfy the requirements of Definition 6.1, and therefore $f(n) \neq O(n)$.

Assembling the facts that $f(n) = \Omega(n)$ and $f(n) \neq O(n)$ with Figure 6.4, we can also conclude that $f(n) = \omega(n)$, $f(n) \neq \Theta(n)$, and $f(n) \neq o(n)$.

Taking it further: We've given definitions of $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$, and $\omega(\cdot)$ that are based on nested quantifiers: there exists a multiplicative constant such that, for all sufficiently large n, etc. If you have a more calculus-based worldview, we could also give an equivalent definition in terms of limits:

- $f(n) = O(g(n))$ if $\lim_{n\to\infty} f(n)/g(n)$ is finite;
- $f(n) = \Omega(g(n))$ if $\lim_{n\to\infty} f(n)/g(n)$ is nonzero;
- $f(n) = \Theta(g(n))$ if $\lim_{n\to\infty} f(n)/g(n)$ is finite and nonzero;
- $f(n) = o(g(n))$ if $\lim_{n\to\infty} f(n)/g(n) = 0$; and
- $f(n) = \omega(g(n))$ if $\lim_{n\to\infty} f(n)/g(n) = \infty$.

For the function $f(n) = 3n^2+1$ in Example 6.5, for example, observe that $\lim_{n\to\infty} \frac{f(n)}{n} = \infty$. Thus $f(n) = \Omega(n)$ and $f(n) = \omega(n)$, but none of the other asymptotic relationships holds.

A (possibly counterintuitive) example

Intuitively, the asymptotic symbols O, Ω, Θ, o, and ω correspond to the numerical comparison symbols $\leq$, $\geq$, $=$, $<$, and $>$—but the correspondence isn't perfect, as we'll see in this example:

Example 6.6: Finding functions, to spec.
For each of (a), (b), (c), and (d) in the following table, fill in the blank with an example of a function f that satisfies the stated conditions.

	$f(n) = O(n^2)$	$f(n) \neq O(n^2)$
$f(n) = \Omega(n^2)$	(a)	(b)
$f(n) \neq \Omega(n^2)$	(c)	(d)

Solution. Three of these cells are fairly straightforward to complete:

(a) $f(n) = n^2$ is $\Theta(n^2)$—that is, it satisfies both $O(n^2)$ and $\Omega(n^2)$.
(b) $f(n) = n$ is $o(n^2)$—that is, it satisfies $O(n^2)$ but not $\Omega(n^2)$.
(c) $f(n) = n^3$ is $\omega(n^2)$—that is, it satisfies $\Omega(n^2)$ but not $O(n^2)$.

But cell (d)—a function $f(n)$ that is *neither* $O(n^2)$ nor $\Omega(n^2)$—appears more challenging (perhaps even impossible). But let's look at the definitions carefully:

For $f(n) \neq O(g(n))$ to be true, we need, for any constants $c > 0$ and $n_0 \geq 0$, that there exists $\overline{n} \geq n_0$ such that $f(\overline{n}) > c\overline{n}^2$.

Similarly, for $f(n) \neq \Omega(n^2)$ to be true, we need, for any constants $d > 0$ and $n_0 \geq 0$, there to exist $\underline{n} \geq n_0$ such that $f(\underline{n}) < d\underline{n}^2$.

(In other words, intuitively, if I tell you a value of n_0, you have to be sure that f both exceeds n^2 at some point $n \geq n_0$ *and* is exceeded by n^2 for some point $n \geq n_0$. The "trick" is that it doesn't have to be the same value of n!) Here's one way to simultaneously achieve these conditions. We'll define the function f in a *piecewise* manner, so that for, say, even values of n the function grows faster than n^2, and for odd values it grows slower:

$$f(n) = n^{2+(-1)^n} = \begin{cases} n^3 & \text{if } n \text{ is even} \\ n & \text{if } n \text{ is odd.} \end{cases}$$

(See Figure 6.5b for a plot of this function.) Below, we'll argue formally that $f(n) \neq O(n^2)$. Together with the proof that $f(n) \neq \Omega(n^2)$, which is left to you as Exercise 6.44, this function will allow us to finish the required table. (See Figure 6.5a.)

To prove that $f(n) \neq O(n^2)$, let $c > 0$ and $n_0 \geq 0$ be arbitrary. Let $\overline{n}$ be the smallest even number strictly greater than $\max(c, n_0)$. Then $f(\overline{n}) = \overline{n}^3$ and $\overline{n}^3 > c \cdot \overline{n}^2$ because we chose $\overline{n} > c$. But then it is not the case that $\forall n \geq n_0 : f(n) \leq cn^2$. Because this argument holds for arbitrary $c > 0$ and $n_0 \geq 0$, we conclude that $f(n) \neq O(n^2)$.

Problem-solving tip: When you're confronted with a problem with seemingly contradictory constraints, as in the bottom-right cell of the table in Example 6.6, very carefully write down what the constraints require. This process can help you see why the constraints aren't actually contradictory.

Some properties of Ω, Θ, o, and ω

Many of the properties of $O(\cdot)$ also hold for the other four asymptotic notions; for example, all five of $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$, and $\omega(\cdot)$ obey transitivity, and several obey reflexivity. See Exercises 6.45–6.53.

One of the subtlest aspects of asymptotic notation is the fact that two functions can be *incomparable* with respect to their rates of growth: we can identify two functions f and g such that none of the asymptotic relationships holds. (That is, it's possible for all five of these statements to be true: $f \neq O(g)$, $f \neq \Omega(g)$, $f \neq \Theta(g)$, $f \neq o(g)$, and $f \neq \omega(g)$.) Intuitively,

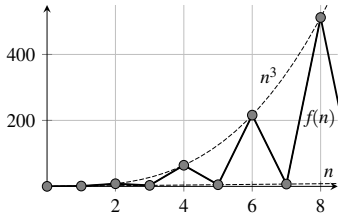
$$f(n) = O(g(n)) \text{ means (roughly)} \quad \text{“the growth rate of } f \leq \text{ the growth rate of } g.\text{”} \tag{A}$$

$$f(n) = \Omega(g(n)) \text{ means (roughly)} \quad \text{“the growth rate of } f \geq \text{ the growth rate of } g.\text{”} \tag{B}$$

Definitions 6.11, 6.12, and 6.13 correspond to three of the four combinations of truth values for these two statements: (A) and (B) is Θ; (A) but not (B) is o; and (B) but not (A) is ω. But be careful! For two real numbers $a \in \mathbb{R}$ and $b \in \mathbb{R}$, it's impossible for $a \leq b$ and $a \geq b$ to both be false. But it *is* possible for both of the inequalities (A) *and* (B) to be false! The functions $g(n) = n^2$ and the function $f(n)$ from Example 6.6 that equals either n^3 or n depending on the parity of n are an example of a pair of functions for which

	$f(n) = O(n^2)$	$f(n) \neq O(n^2)$
$f(n) = \Omega(n^2)$	$f(n) = n^2$	$f(n) = n^3$
$f(n) \neq \Omega(n^2)$	$f(n) = n$	$f(n) = \begin{cases} n^3 & \text{if } n \text{ is even} \\ n & \text{if } n \text{ is odd.} \end{cases}$

(a) The summary of a solution to Example 6.6.



(b) A plot of the function $f(n)$ from (a) that is neither $O(n^2)$ nor $\Omega(n^2)$.

Figure 6.5 A solution, and a plot, for Example 6.6.

neither (A) nor (B) is satisfied. (Thus f was neither $O(n^2)$ nor $\Omega(n^2)$, and therefore not $\omega(n^2)$, $o(n^2)$, or $\Theta(n^2)$, either.)

> **Taking it further:** The real numbers satisfy the mathematical property of *trichotomy* (Greek: "division into three parts"): for $a, b \in \mathbb{R}$, exactly one of $\{a < b, a = b, a > b\}$ holds. Functions compared asymptotically do not obey trichotomy: for two functions f and g, it's possible for *none* of $\{f = o(g), f = \Theta(g), f = \omega(g)\}$ to hold.

Before we begin to apply asymptotic notation to the analysis of algorithms, we'll close this section with a few notes about the use (and abuse) of asymptotic notation.

Using asymptotics in arithmetic expressions

It is often convenient to use asymptotic notation in arithmetic expressions. We permit ourselves to write something like $O(n \log n) + O(n^3) = O(n^3)$, which intuitively means that, given functions that grow no faster than $n \log n$ and n^3, their sum grows no faster than n^3 too. When asymptotic notation like $O(n^2)$ appears on the left-hand side of an equality, we interpret it to mean an arbitrary unnamed function that grows no faster than n^2. For example, making $\log n$ calls to an algorithm whose running time is $O(n)$ requires $\log n \cdot O(n) = O(n \log n)$ time.

Using asymptotics with multiple variables

It will also occasionally turn out to be convenient to be able to write asymptotic expressions that depend on more than one variable. Giving a precise technical definition of multivariate asymptotic notation is a bit subtle, but the intuition precisely matches the univariate definitions we've already given. We'll use the notation $g(n, m) = O(f(n, m))$ to mean "for all sufficiently large n and m, there exists a constant c such that $g(n, m) \leq c \cdot f(n, m)$." For example, the function $f(n, m) = n^2 + 3m - 5$ satisfies $f(n, m) = O(n^2 + m)$.

A common mistake and some meaningless language

There is a widespread—and incorrect—sloppy use of asymptotic notation: it is unfortunately common for people to use $O(\cdot)$ when they mean $\Theta(\cdot)$. You will sometimes encounter claims like:

$$\text{"I prefer } f \text{ to } g\text{, because } f(n) = O(n^2) \text{ and } g(n) = O(n^3)\text{."} \tag{1}$$

But this logic doesn't make sense: $O(\cdot)$ defines only an upper bound, so either of f or g might grow more slowly than the other! Saying (1) is like saying

"Alice is richer than Bob, because Alice has at most \$1,000,000,000 and Bob has at most \$1,000,000." (2)

(Alice *might* be richer than Bob, sure. But perhaps they both have twenty bucks each, or perhaps Bob has \$1,000,000 and Alice has nothing.) Use $O(\cdot)$ when you mean $O(\cdot)$, and to use $\Theta(\cdot)$ when you mean $\Theta(\cdot)$—and be aware that others may use $O(\cdot)$ improperly. (And, gently, correct them if they're doing so.)

There's a related imprecise use of asymptotics that leads to statements that don't mean anything. For example, consider statements like "$f(n)$ is at least $O(n^3)$" or "$f(n)$ is at most $\Omega(n^2)$." These sentences have no meaning: they say "$f(n)$ grows at least as fast as at most as fast as n^3" and "$f(n)$ grows at most as fast as at least as fast as n^2." (?!?) Be careful: use upper bounds as upper bounds, and use lower bounds as lower bounds! Again, by analogy, consider these sentences (thanks to Tom Wexler for suggesting (5)):

"My weight is more than ≤ 100 kilograms" (3)

or "I am shorter than some person who is taller than 4 feet tall." (4)

or "You could save up to 50% or more!" (5)

None of these sentences says anything!

Computer Science Connections

Vacuum Tubes, Transistors, and Moore's Law

The earliest electronic computers—machines like the Colossus at Bletchley Park and the ENIAC at the University of Pennsylvania—were initially developed in the early-to-mid 1940s. These first electronic machines used *vacuum tubes* (or *valves*) as the core of their circuitry. Vacuum tubes can be used for logical switching are devices that look a bit like a light bulb—and they also burn out at roughly similar rates to light bulbs. (These early computers contained tens of thousands of these devices, which meant that a vacuum tube would fail every day or two.) The discovery of the *transistor* was a major breakthrough of the late 1940s; transistors can do the same kinds of things as vacuum tubes, but they were substantially more energy efficient and smaller—and have gotten smaller and smaller as time has gone on. (Vacuum tubes are ≈centimeters in size; transistors are now ≈10 nanometers in size.)

In 1965, Gordon Moore, one of the co-founders of Intel, published an article making a basic prediction—and it's been reinterpreted many times—that processing power would double roughly once every 18–24 months [91]. (It's been debated and revised over time, by, for example, interpreting "processing power" as the number of transistors on the processor rather than what it can actually compute.) This prediction later came to be known as *Moore's Law*—not a real "law" like Ohm's Law or the Law of Large Numbers, of course, but rather simply a prediction. That said, it's proven to be a remarkably robust prediction: for something like 40 to 50 years, it has proven to be a consistent guide to the massive increase in processing power for a typical computer user over many decades. (See Figure 6.6.)

Claims that "Moore's Law is just about to end!" have been made for many decades, and yet Moore's Law has still proven to be remarkably accurate over time. Its imminent demise is still being predicted today, and yet it's still been a pretty good model of computing power [92]. One probable reason that Moore's Law has held for as long as it has is a little bizarre: the repeated publicity surrounding Moore's Law! Because chip manufacturing companies "know" that the public generally expects processors to have twice as many transistors in two years, these companies may

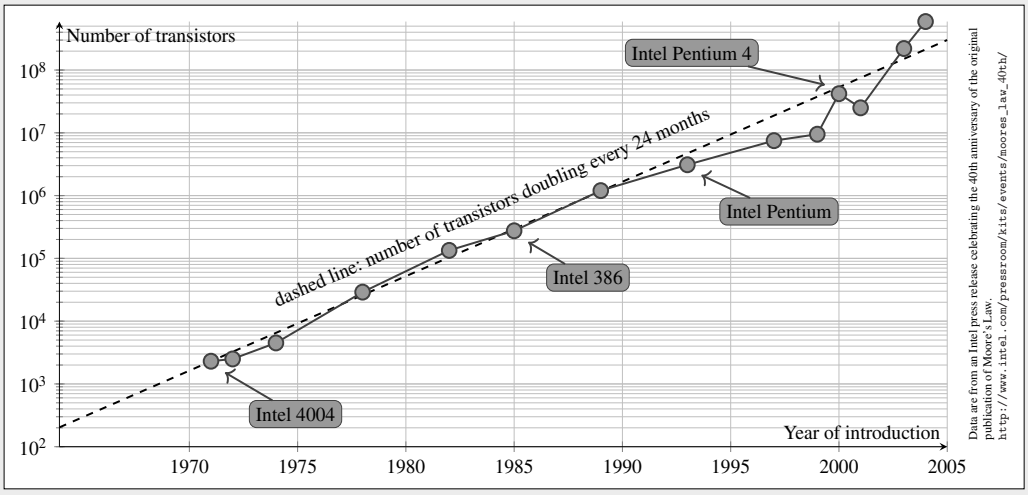


Figure 6.6 The number of transistors per processor, for Intel brand processors introduced over the last 50 years. The dashed line indicates the rate of growth we'd see if the number of transistors per processor doubled every two years (starting with the Intel 4004 in 1971).

actually be setting research-and-development targets based on meeting Moore's Law. (Just as in a physical system, we cannot observe a phenomenon without changing it!) But Moore's Law–level growth *must* come to an end at some point—we're beginning to run up against physical limits in the size of transistors; modern transistors aren't much bigger than the size of a silicon atom—and so we're entering what people tend to call a "post-Moore world," in which the now-expected performance improvement over time will have to come from some other kind of innovation.

EXERCISES

Part of the motivation for asymptotic analysis was that algorithms are typically analyzed ignoring constant factors. Ignoring constant factors in analyzing an algorithm may seem strange: if algorithm **A** *runs twice as fast as* **B**, *then* **A** *is way faster! But the reason we care more about asymptotic running time is that even an improvement by a factor of two is quickly swamped by an asymptotic improvement for even slightly larger inputs. Here are a few examples:*

6.1 Suppose that linear search can find an element in a sorted list of n elements in n steps on a particular machine. Binary search (perhaps not implemented especially efficiently) requires $100 \log n$ steps. For what values of $n \geq 2$ is linear search faster?

Alice implements Merge Sort so, on a particular machine, it requires exactly $\lceil 8n \log n \rceil$ *steps to sort n elements. Bob implements Heap Sort so it requires exactly* $\lceil 5n \log n \rceil$ *steps to sort n elements. Charlie implements Selection Sort so it requires exactly* $2n^2$ *steps to sort n elements. Suppose that Alice can sort* 1000 *elements in* 1 *minute.*

6.2 How many elements can Bob sort in a minute? How many can Charlie sort in a minute?

6.3 What is the largest value of n that Charlie can sort faster than Alice?

6.4 Charlie, devastated by the news from Exercise 6.3, buys a computer that's twice the speed of Alice's. What is the largest value of n that Charlie can sort faster than Alice now?

Let $f(n) = 9n + 3$ *and let* $g(n) = 3n^3 - n^2$. *(See Figure 6.7.)*

Let $a(n) = 7n$, *let* $b(n) = 3n^2 + \sin n$, *and let* $c(n) = 128$. *(See Figure 6.8.)*

6.5 Prove that $f(n) = O(n)$.

6.6 Prove that $f(n) = O(n^2)$.

6.7 Prove that $f(n) = O(g(n))$.

6.8 Prove that $g(n) = O(n^3)$.

6.9 Prove that $g(n) = O(n^4)$.

6.10 Prove that $g(n)$ is not $O(n^2)$.

6.11 Prove that $g(n)$ is not $O(n^{3-\epsilon})$, for any $\epsilon > 0$.

6.12 Prove that $a(n)$ is $O(n^2)$.

6.13 Prove that $b(n)$ is $O(n^2)$.

6.14 Prove that $c(n)$ is $O(n^2)$.

6.15 Consider two functions $f : \mathbb{Z}^{\geq 0} \to \mathbb{Z}^{\geq 0}$ and $g : \mathbb{Z}^{\geq 0} \to \mathbb{Z}^{\geq 0}$. We defined $O(\cdot)$ notation as follows:

$$f(n) = O(g(n)) \text{ if there exist constants } c > 0 \text{ and } n_0 \geq 0 \text{ such that } \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Suppose $f(n) = O(g(n))$. Explain why there are infinitely many choices of c *and* infinitely many choices of n_0 that satisfy the definition of $O(\cdot)$.

6.16 It turns out that both c and n_0 are necessary to the definition of $O(\cdot)$. (See Exercise 6.15 for a reminder of the definition.) Define the following alternative asymptotic notation, leaving out c (using $c = 1$) from the definition:

$$f(n) = P(g(n)) \text{ if there exists a constant } n_0 \geq 0 \text{ such that } \forall n \geq n_0 : f(n) \leq g(n).$$

Prove that $P(\cdot)$ and $O(\cdot)$ mean different things: prove that there exist functions f and g such that either (i) $f = O(g)$ but $f \neq P(g)$, or (ii) $f \neq O(g)$ but $f = P(g)$.

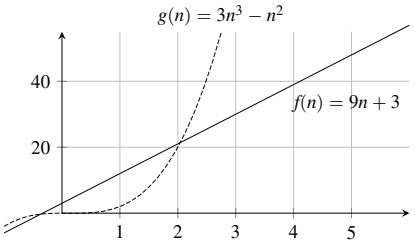


Figure 6.7 Two functions for Exercises 6.5–6.11.

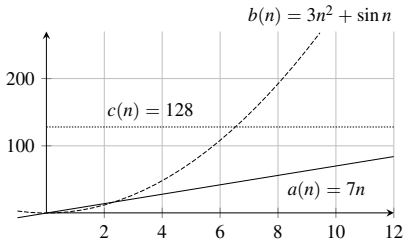


Figure 6.8 Three functions for Exercises 6.12–6.14.

6.17 Repeat Exercise 6.16, this time showing that we cannot omit n_0 from the definition. Define the following asymptotic notation (which implicitly uses $n_0 = 1$):

$$f(n) = Q(g(n)) \text{ if there exists a constant } c > 0 \text{ such that } \forall n \geq 1 : f(n) \leq c \cdot g(n).$$

Prove that there exist functions f and g such that either (i) $f = O(g)$ but $f \neq Q(g)$, or (ii) $f \neq O(g)$ but $f = Q(g)$.

The next several exercises ask you to prove some of properties of $O(\cdot)$ that we stated without proof earlier in the section. (For a model of a proof of this type of property, see Lemma 6.2 and its proof in this section.)

6.18 Prove Lemma 6.3, the transitivity of $O(\cdot)$: if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

6.19 Prove the first half of Lemma 6.4: if $f(n) = O(h_1(n))$ and $g(n) = O(h_2(n))$, then $f(n) + g(n) = O(h_1(n) + h_2(n))$.

6.20 Prove the second half of Lemma 6.4: if $f(n) = O(h_1(n))$ and $g(n) = O(h_2(n))$, then $f(n) \cdot g(n) = O(h_1(n) \cdot h_2(n))$.

6.21 Prove Lemma 6.5: if $p(n) = \sum_{i=0}^{k} a_i n^i$ is a polynomial, then $p(n) = O(n^k)$.

6.22 Prove that the bound from Exercise 6.21 cannot be improved. That is, consider $p(n) = \sum_{i=0}^{k} a_i n^i$ with $a_k > 0$. Prove that $p(n)$ is not $O(n^{k-\epsilon})$ for any $\epsilon > 0$.

Lemmas 6.6 and 6.7 state that all logarithmic functions grow slower than all polynomial functions, which grow slower than all exponential functions. (For example, $\log n = O(n^{0.000001})$ and $n^{1000000} = O(1.000001^n)$.) While fully general proofs are more calculus-intensive than we want to be in this book, here are a few simpler results to prove:

6.23 Assuming Lemma 6.6, prove that any polylogarithmic function $f(n) = \log^k(n)$ satisfies $f(n) = O(n^\epsilon)$ for any $\epsilon > 0$ and any integer $k \geq 0$. (A *polylogarithmic* function is one that's a polynomial where the terms are powers of $\log n$ instead of powers of n—hence a poly(nomial of the)log function.)

6.24 Prove the special case of Lemma 6.6 for $\epsilon = 1$: that is, prove that $\log n = O(n)$. Specifically, do so by proving that $\log n \leq n$ for all integers $n \geq 1$, using strong induction.

The next three exercises explore whether the asymptotic properties of two functions f and g "transfer over" to the functions $\log f$ and $\log g$. Specifically, consider two functions $f : \mathbb{Z}^{\geq 0} \to \mathbb{Z}^{\geq 1}$ and $g : \mathbb{Z}^{\geq 0} \to \mathbb{Z}^{\geq 1}$. (Note: the outputs of f and g are always positive, so that $\log(f(n))$ and $\log(g(n))$ are well defined.)

6.25 Assume that, for all n, we have $f(n) \geq n$ and $g(n) \geq n$. Furthermore assume that $f(n) = O(g(n))$. Prove that the function $\ell(n) = \log(f(n))$ satisfies $\ell(n) = O(\log(g(n)))$.

6.26 Prove that the converse of Exercise 6.25 is *not* true: identify functions $f(n)$ and $g(n)$ where $f(n) \geq n$ and $g(n) \geq n$ such that $\log(f(n)) = O(\log(g(n)))$ but $f(n) \neq O(g(n))$. *(Hint: what's $\log n^2$?)*

6.27 Prove that assuming $f(n) \geq n$ and $g(n) \geq n$ in Exercise 6.25 was necessary: identify functions $f : \mathbb{Z}^{\geq 0} \to \mathbb{Z}^{\geq 1}$ and $g : \mathbb{Z}^{\geq 0} \to \mathbb{Z}^{\geq 1}$ where $f(n) = O(g(n))$ but $\ell(n) \neq O(\log(g(n)))$ for the function $\ell(n) = \log(f(n))$.

6.28 For a real number $b \geq 1$, define the function $f(n) = b^n$. Prove Lemma 6.9: $f(n) = O(c^n)$ if and only if $b \leq c$.

6.29 Just as with a virus (as we now know all too well), an idea "going viral"—a video, a joke, a hashtag, an app—can be reasonably modeled as a form of exponential growth: if each person who "adopts" the idea on a particular day causes two others to adopt that idea the next day, then 1 adopter on Day Zero means 2 new ones on Day One (for a total of 3), and 4 new ones on Day Two (for a total of 7), etc. Here we might call 2 the *spreading rate,* the number of people "infected" by each new adopter. Let $r_0 \in \mathbb{Z}^{\geq 1}$ be a spreading rate. Define $f(n) = \sum_{i=1}^{n} (r_0)^i$ to be the number of people who have adopted by Day n. Is $f(n) = O((r_0)^n)$? Prove your answer.

6.30 Prove that $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

Consider the function $f(n) = n + \frac{1}{n}$. (See Figure 6.9.) Because $f(0)$ is undefined and the output $f(n)$ is not an integer for any integer $n \geq 2$, treat f as a function from $\mathbb{Z}^{\geq 1}$ to $\mathbb{R}$. Prove all of your answers to the following questions:

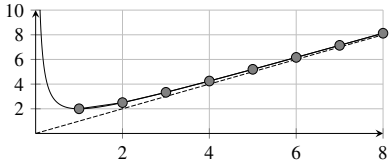


Figure 6.9 The function $f(n) = n + \frac{1}{n}$, for Exercises 6.31–6.33.

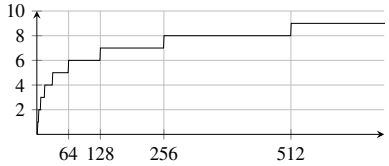


Figure 6.10 The function $k(n)$, where $k(n)$ is the nonnegative integer such that $2^{k(n)} \leq n < 2^{k(n)+1}$, for Exercises 6.34–6.36.

6.31 Is $f(n) = O(1)$? $\Omega(1)$? $\Theta(1)$? $o(1)$? $\omega(1)$?

6.32 Is $f(n) = O(n)$? $\Omega(n)$? $\Theta(n)$? $o(n)$? $\omega(n)$?

6.33 Is $f(n) = O(n^2)$? $\Omega(n^2)$? $\Theta(n^2)$? $o(n^2)$? $\omega(n^2)$?

For an integer $n \geq 0$, let $k(n)$ denote the nonnegative integer such that $2^{k(n)} \leq n < 2^{k(n)+1}$. That is, $2^{k(n)}$ takes n and "rounds down" to a power of two: for example, $2^{k(4)} = 2^2 = 4$ and $2^{k(5)} = 2^2 = 4$ and $2^{k(202)} = 2^7 = 128$ and $2^{k(55,057)} = 2^{15} = 32{,}768$. (See Figure 6.10.)

6.34 Prove that $2^{k(n)}$ and $2^{k(n)+1}$ are both $\Theta(n)$.

6.35 Prove that $k(n) = \Theta(\log n)$.

6.36 Let $b > 1$ be an arbitrary constant. Let $k_b(n)$ denote the nonnegative integer such that $b^{k_b(n)} \leq n < b^{k_b(n)+1}$. Prove that $k_b(n) = \Theta(\log n)$ *for any constant value* $b > 1$.

6.37 In Chapter 11, we'll talk about graphs and the "density" of graphs. If $f(n)$ denotes the number of edges in an n-node graph (we'll define those terms later), then a graph is called *sparse* if $f(n) = O(n)$ and it's called *dense* if $f(n) = \Theta(n^2)$. Prove that there exists a function $f : \mathbb{Z}^{\geq 0} \to \mathbb{Z}^{\geq 0}$ satisfying $0 \leq f(n) \leq n^2$ such that neither $f(n) = \Theta(n^2)$ nor $f(n) = O(n)$.

6.38 Prove or disprove: the all-zero function $f(n) = 0$ is the *only* function that is $\Theta(0)$.

6.39 Give an example of a function $f(n)$ such that $f(n) = \Theta(f(n)^2)$.

6.40 Let $k \in \mathbb{Z}^{\geq 0}$ be any constant. Prove that $n^k = o(n!)$.

6.41 Let $f : \mathbb{Z}^{\geq 0} \to \mathbb{Z}^{\geq 0}$ be an arbitrary function. Define the function $g(n) = f(n) + 1$. Prove that $g(n) = O(f(n))$ *if and only if* $f(n) = \Omega(1)$.

6.42 For each of the four blanks (a), (b), (c), and (d) in the following table, identify an example of a function f that satisfies the stated conditions, or argue that it's impossible to satisfy both conditions.

	$f(n) = o(n^2)$	$f(n) \neq o(n^2)$
$f(n) = \omega(n^2)$	(a)	(b)
$f(n) \neq \omega(n^2)$	(c)	(d)

6.43 Let f and g be arbitrary functions. Prove that *at most one* of the three properties $f(n) = o(g(n))$ and $f(n) = \Theta(g(n))$ and $f(n) = \omega(g(n))$ can hold.

6.44 Complete the proof in Example 6.6: prove that $f(n) \neq \Omega(n^2)$, where $f(n)$ is the function

$$f(n) = \begin{cases} n^3 & \text{if } n \text{ is even} \\ n & \text{if } n \text{ is odd.} \end{cases}$$

Many (but not all) of the properties of $O(\cdot)$ also hold for the other four asymptotic notions. Consider the following properties, for arbitrary functions f, g, and h:

6.45 Prove that Ω is transitive: if $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$.

6.46 Prove that Θ is transitive: if $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.

6.47 Prove that o is transitive: if $f(n) = o(g(n))$ and $g(n) = o(h(n))$, then $f(n) = o(h(n))$.

6.48 Is Ω symmetric? Prove or disprove: if $f(n) = \Omega(g(n))$, then $g(n) = \Omega(f(n))$.

6.49 Is Θ symmetric? Prove or disprove: if $f(n) = \Theta(g(n))$, then $g(n) = \Theta(f(n))$.

6.50 Is ω symmetric? Prove or disprove: if $f(n) = \omega(g(n))$, then $g(n) = \omega(f(n))$.

6.51 Is O reflexive? Prove or disprove: $f(n) = O(f(n))$.

6.52 Is Ω reflexive? Prove or disprove: $f(n) = \Omega(f(n))$.

6.53 Is ω reflexive? Prove or disprove: $f(n) = \omega(f(n))$.

6.54 Consider the false claim below, and the bogus proof that follows. Where, precisely, does the proof go wrong?

False Claim. The function $f(n) = n^2$ satisfies $f(n) = O(n)$.

Bogus proof. We proceed by induction on n.

For the base case ($n = 1$), we have $n^2 = 1$. Thus $f(1) = O(n)$ because $1 \leq n$ for all $n \geq 1$. (Choose $c = 1$ and $n_0 = 1$.)

For the inductive case ($n \geq 2$), we assume the inductive hypothesis—namely, assume that $(n-1)^2 = O(n)$. We must show that $n^2 = O(n)$. Here is the proof:

$$
\begin{aligned}
n^2 &= (n-1)^2 + 2n - 1 && \textit{by factoring} \\
&= O(n) + 2n - 1 && \textit{by the inductive hypothesis} \\
&= O(n) + O(n) && \textit{by definition of } O(\cdot) \textit{ and Lemma 6.4} \\
&= O(n). && \square
\end{aligned}
$$

6.3 Asymptotic Analysis of Algorithms

> But why this idle toil to paint that day,
> This time elaborately thrown away?
>
> Edward Young (1683–1765)
> "A Poem On The Last Day" (1713)

The main reason that computer scientists are interested in asymptotic analysis is for its application to the *analysis of algorithms*. When, for example, we compare different algorithms that solve the same problem—say, Merge Sort, Selection Sort, and Insertion Sort—we want to be able to give a meaningful answer to the question *which algorithm is the fastest?* (And different inputs may trigger different behaviors in the algorithms under consideration: when the input array is sorted, for example, Insertion Sort is faster than Merge Sort and Selection Sort; when the input is very far from sorted, Merge Sort is fastest. But typically we still would like to identify a single answer to the question of which algorithm is the fastest.)

When evaluating the running time of an algorithm, we generally follow asymptotic principles. Specifically, we will generally ignore constants in the two ways that $O(\cdot)$ and its asymptotic siblings do:

(1) We don't care much about what happens for small inputs. There might be small special-case inputs for which an algorithm is particularly fast, but this fast performance on a few special inputs doesn't mean that the algorithm is fast in general. For example, imagine an algorithm for primality testing that returns true if it's given as input any of 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, or 97; it returns false if it's given any other input less than 100; and for numbers greater than 100 it calls **isPrime** from Figure 4.32. Despite its speed on a few special cases, we wouldn't consider this algorithm to be faster *in general* than **isPrime**. (Incidentally, it's also harder to persuade yourself that it's correct—it's so easy to have made a typo in that list of small primes!) We seek *general* answers to the question *which algorithm is faster?,* which leads us to pay little heed to special cases.

(2) We typically evaluate the running time of an algorithm not by measuring elapsed time on the "wall clock," but rather by counting the number of steps that the algorithm takes to complete. (How long a program takes on your laptop, in terms of the hypothetical clock hanging on your wall, is affected by all sorts of things unrelated to the algorithm, like how many videos you're watching while the algorithm executes.) We will generally ignore multiplicative constants in counting the number of steps consumed by an algorithm. One reason is so that we can give a machine-independent answer to the *which algorithm is faster?* question; how much is accomplished by one instruction on an Intel processor may be different from one instruction on an ARM processor, and ignoring constants allows us to compare algorithms in a way that doesn't depend on grungy details about the particular machine.

Definition 6.14: Running time of an algorithm on a particular input.
Consider an algorithm $\mathcal{A}$ and an input x. The *running time of algorithm $\mathcal{A}$ on input x* is the number of primitive steps that $\mathcal{A}$ takes when it's run on input x.

For example, imagine running **binarySearch** on the input $x = \langle [2,3,5,7,11,13,17,19,23,29,31], 4\rangle$. The precise number of primitive steps in this execution depends on the particular machine on which the algorithm is being run, but it involves successively comparing 4 to 13, then 5, then 2, and finally 3.

Taking it further: Definition 6.14 is intentionally vague about what a "primitive step" is, but it's probably easiest to think of a single machine instruction as a primitive step. That single machine instruction might add or compare two numbers, increment a counter, return a value, etc. Different hardware systems might have different granularity in their "primitive steps"—perhaps a Mac desktop can "do more" in one machine instruction than an iPhone can do—but, as we just indicated, we'll look to analyze algorithms independently of this detail. Theoretical computer scientists who work on algorithmic analysis study a very precise model of computation, in which a "primitive step" is very carefully defined. By far the most common model is the *Turing machine,* named after the British code-breaking, artificial intelligence, and computing pioneer Alan Turing (1912–1954), who defined the model in a groundbreaking paper in the 1930s [126].

We typically evaluate an algorithm's efficiency by counting asymptotically of the number of primitive steps used by an algorithm's execution, rather than by using a stopwatch to measure how long the algorithm actually takes to run on a particular input on a particular machine. One reason is that it's very difficult to properly measure this type of performance; see p. 6-33 for some discussion about why.

In certain applications, particularly in *scientific computing* (the subfield of CS devoted to processing and analyzing real-valued data, where we have to be concerned with issues like accumulated rounding errors in long calculations), it is typical to use a variation on asymptotic analysis. Calculations on integers are substantially cheaper than those involving floating point values (see p. 2-20); thus in this field one typically doesn't bother counting integer operations, and instead we only track floating point operations, or *flops*. Because flops are substantially more expensive, often we'll keep track of the constant on the leading (highest-degree) term—for example, an algorithm might require $\frac{3}{2}n^2 + O(n\log n)$ flops or $2n^2 + O(n)$ flops. (We'd choose the former.)

6.3.1 Worst-Case Analysis

We will generally evaluate the efficiency of an algorithm $\mathcal{A}$ by thinking about its performance as the input gets large: what happens to the number of steps consumed by $\mathcal{A}$ as a function of the input size n? Furthermore, we generally assume the worst: when we ask about the running time of an algorithm $\mathcal{A}$ on an input of size n, we are interested in the running time of $\mathcal{A}$ *on the input of size n for which* $\mathcal{A}$ *is the slowest*.

Definition 6.15: Worst-case running time of an algorithm.
The *worst-case running time of an algorithm* $\mathcal{A}$ is

$$T_{\mathcal{A}}(n) = \max_{x:|x|=n} \big[\text{the number of primitive steps used by } \mathcal{A} \text{ on input } x\big].$$

We will be interested in the asymptotic behavior of the function $T_{\mathcal{A}}(n)$.

When we perform *worst-case analysis* of an algorithm—analyzing the asymptotic behavior of the function $T_{\mathcal{A}}(n)$—we seek to understand the rate at which the running time of the algorithm increases as the input size increases. Because a primary goal of algorithmic analysis is to provide a *guarantee* on the running time of an algorithm, we will be pessimistic, and think about how quickly $\mathcal{A}$ performs on the input of size n that's the worst for algorithm $\mathcal{A}$.

Taking it further: Occasionally we will perform *average-case analysis* instead of worst-case analysis, by computing the *expected* (average) performance of algorithm $\mathcal{A}$ for inputs drawn from an appropriate distribution. It can be difficult to decide on an appropriate distribution, but sometimes this approach makes more sense than being purely pessimistic. See Section 6.3.2.

It's also worth noting that using asymptotic, worst-case analysis can sometimes be misleading. There are occasions in which an algorithm's performance in practice is very poor despite a "good" asymptotic running time—for example, because the multiplicative constant suppressed by the $O(\cdot)$ is massive. (And conversely: sometimes an algorithm that's asymptotically slow in the worst case might perform very well on problem instances that actually show up in real applications.) Asymptotics capture the high-level performance of an algorithm, but constants matter too!

Some examples: three sorting algorithms

Figure 6.11 shows a sampling of worst-case running times for a number of the algorithms you may have encountered earlier in this book or in previous CS classes. In the rest of this section, we'll prove some of these results as examples. We'll start our analysis with Selection Sort, shown in Figure 6.12.

Example 6.7: Selection Sort.
What is the worst-case running time of Selection Sort?

Solution. The outer **for** loop's body (Lines 2–6) is executed n times, once each for $i = 1, 2, \ldots, n$. We complete the body of the inner **for** loop (Lines 4–5) a total of $n - i$ times in iteration i. Thus the total number of times that we execute Lines 4–5 is

$$\sum_{i=1}^{n} n - i = n^2 - \sum_{i=1}^{n} i = n^2 - \frac{n(n+1)}{2} = \frac{n^2-n}{2},$$

worst-case running time	*sample algorithm(s)*
$\Theta(1)$	push/pop in a stack with n elements
$\Theta(\log n)$	binary search in an array with n elements
$\Theta(n)$	linear search in an array with n elements
$\Theta(n \log n)$	merge sort of an array of n elements
$\Theta(n^2)$	selection sort, insertion sort, or bubble sort of an array of n elements
$\Theta(n^3)$	naïve matrix multiplication of two n-by-n matrices
$\Theta(2^n)$	brute-force satisfiability algorithm for a proposition with n variables

Figure 6.11 The running time of some sample algorithms.

```
selectionSort(A[1 . . . n])
1 for i := 1 to n:
2     minIndex := i
3     for j := i + 1 to n:
4         if A[j] < A[minIndex] then
5             minIndex := j
6     swap A[i] and A[minIndex]
```

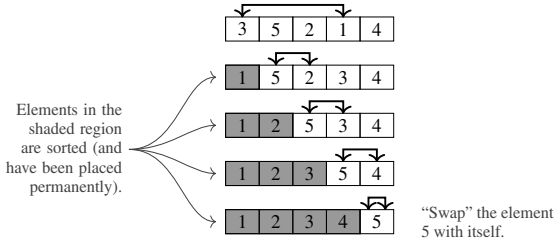


Figure 6.12 Selection Sort: pseudocode and an example. We repeatedly find the minimum element in the unsorted region of the array A, and swap it into the first slot of the unsorted segment.

where $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ by Theorem 5.3.

Notice that the only variation in the running time of Selection Sort based on the particular input array $A[1 \ldots n]$ is in the number of times Line 5 is executed; the number of times that *minIndex* is reassigned in the ith iteration of the inner loop can vary from as low as 0 to as high as $n - i$. If x represents the *total* number of executions of Line 5, then x can be as low as 0 and as high as $\sum_{i=1}^{n} n - i$ (which, as argued above, is less than n^2). The remainder of the algorithm behaves identically regardless of the input array.

Thus, for some constants $c_1 > 0$ and $c_2 > 0$ and $c_3 > 0$, the total number of primitive steps used by the algorithm is $c_1 n + c_2 n^2$ (for all lines except Line 5) plus $c_3 x$ (for Line 5), where $0 \leq x < n^2$. Thus the total running time is between $c_1 n + c_2 n^2$ and $c_1 n + (c_2 + c_3) n^2$, and therefore the asymptotic worst-case running time of Selection Sort is $\Theta(n^2)$.

We are generally interested in the asymptotic performance of algorithms, so the particular values of the constants c_1, c_2, and c_3 from Example 6.7, which reflect the number of primitive steps corresponding to each line of the pseudocode in Figure 6.12, are irrelevant to our final answer. (One exception is that we may sometimes try to count exactly the number of *comparisons* between elements of A, or *swaps* of elements of A; see Exercises 6.55–6.63.)

We'll now turn to our second sorting algorithm, Insertion Sort, which slowly increases the size of a sorted subarray by swapping adjacent elements. (See Figure 6.13.)

Example 6.8: Insertion Sort.
Insertion Sort is more sensitive to the structure of its input than Selection Sort: if A is in sorted order, then, in every iteration of the outer **for** loop, the **while** loop in Lines 3–5 terminates immediately (because the test $A[j] > A[j-1]$ fails). On the other hand, if the input array is in *reverse* sorted order, then the **while** loop in Lines 3–5 always completes $i - 1$ iterations. In fact, the reverse-sorted array is the worst-case input for Insertion Sort: there can be as many as $i - 1$ iterations of the **while** loop, and there cannot be more than $i - 1$ iterations.

```
insertionSort(A[1 ... n])
1 for i := 2 to n:
2     j := i
3     while j > 1 and A[j] < A[j − 1]:
4         swap A[j] and A[j − 1]
5         j := j − 1
```

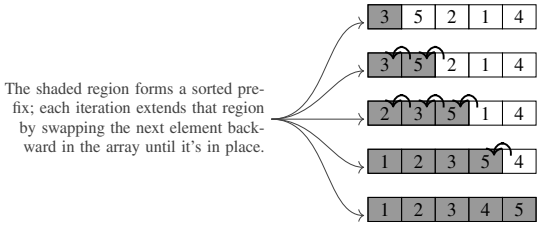


Figure 6.13 Insertion Sort: pseudocode and an example. We maintain a sorted prefix of A (initially consisting only of the first element), and we repeatedly expand the sorted prefix by one element.

In this worst case, when the **while** loop completes $i-1$ swaps for every iteration i, then the total amount of work done by the algorithm is

$$\sum_{i=1}^{n} \underbrace{c}_{\substack{c \text{ is a constant representing the work in a single for-loop iteration } \textit{aside} \text{ from what's in the while loop}}} + (i-1)\underbrace{d}_{\substack{d \text{ is a constant representing the work in one iteration of Lines 3–5}}} = (c-d)n + \sum_{i=1}^{n} id \qquad \textit{rearranging terms}$$

$$= (c-d)n + d \cdot \tfrac{n(n+1)}{2} \qquad \textit{by Theorem 5.3}$$

$$= (c-\tfrac{d}{2})n + \tfrac{d}{2}n^2. \qquad \textit{collecting like terms}$$

This function is $\Theta(n^2)$, so Insertion Sort's worst-case running time is $\Theta(n^2)$.

Finally, we will analyze a third sorting algorithm: Bubble Sort (Figure 6.14), which makes n left-to-right passes through the array; in each pass, adjacent elements that are out of order are swapped. Bubble Sort is a comparatively simpler sorting algorithm to analyze. (But, in practice, it is also a comparatively slow sorting algorithm to run!)

Example 6.9: Bubble Sort.
Bubble Sort repeatedly compares $A[j]$ and $A[j+1]$ (swapping the two elements if necessary) for many different values of j. Every time the body of the inner loop is executed, the algorithm does a constant amount of work: exactly one comparison and either zero or one swaps. Thus any particular execution of the body of the inner loop takes $\Theta(1)$ time—or, more precisely, an amount of time t satisfying $c \le t \le d$, for two constants $c > 0$ and $d > 0$.

Therefore the total running time of Bubble Sort is between $\sum_{i=1}^{n}\sum_{j=1}^{n-i} c$ and $\sum_{i=1}^{n}\sum_{j=1}^{n-i} d$. The summation $\sum_{i=1}^{n} n-i$ is $\Theta(n^2)$, precisely as in Example 6.7, and thus Bubble Sort's running time is $\Omega(cn^2) = \Omega(n^2)$ and $O(dn^2) = O(n^2)$. Therefore Bubble Sort is $\Theta(n^2)$.

Problem-solving tip: Precisely speaking, the number of primitive steps required to execute, for example, Lines 3–4 of Bubble Sort varies based on whether a swap has to occur. In Example 6.9, we carried through the analysis considering two different constants representing this difference. But, more simply, we could say that Lines 3–4 of Bubble Sort take $\Theta(1)$ time, without caring about the particular constants. You can use this simpler approach to streamline arguments like the one in Example 6.9.

Before we move on from sorting, we'll mention one more algorithm, Merge Sort, which proceeds recursively by splitting the input array in half, recursively sorting each half, and then "merging" the sorted

```
bubbleSort(A[1 . . . n])
1 for i := 1 to n:
2     for j := 1 to n − i:
3         if A[j] > A[j + 1] then
4             swap A[j] and A[j + 1]
```

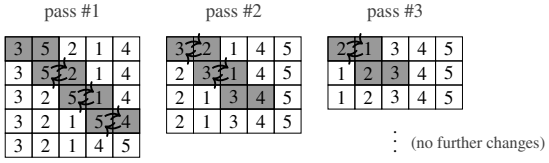


Figure 6.14 Bubble Sort: pseudocode and an example. We make n left-to-right passes through $A[1 \ldots n]$; in each pass, we swap each pair of adjacent elements that are out of order.

subarrays into a single sorted array. But we will defer the analysis of Merge Sort to Section 6.4: to analyze recursive algorithms like Merge Sort, we will use *recurrence relations* which represent *the algorithm's running time itself* as a recursive function.

Some more examples: search algorithms

We'll now analyze two search algorithms, both of which determine whether a particular value x appears in an array A. We'll start with Linear Search (Figure 6.15a), which simply walks through the (possibly unsorted) array A and successively compares each element to the sought value x. We'll then look at Binary Search (Figure 6.15b and 6.15c), which relies on the array $A[1 \ldots n]$ being sorted. It proceeds by defining a range of the array in which x would be found if it is present, and then repeatedly halving the size of that range by comparing x to the middle entry in that range.

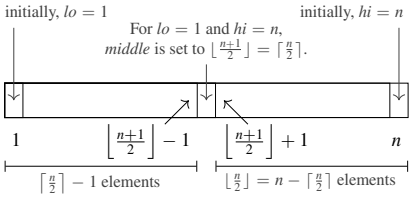
Unless otherwise specified (and we will rarely specify otherwise), we are interested in the worst-case behavior of algorithms. *This concern with worst-case behavior includes lower bounds!* Here's an example of the analysis of an algorithm that suffers from this confusion:

Example 6.10: Linear Search, unsatisfactorily analyzed.
What is incomplete or incorrect in the following analysis of the worst-case running time of Linear Search?

The running time of Linear Search is obviously $O(n)$: we at most iterate over every element of the array, performing a constant number of operations per element. And it's obviously $\Omega(1)$: no matter what the inputs A and x are, the algorithm certainly at least does one operation (setting $i := 1$ in the first line), even if it immediately returns because $A[1] = x$.

```
linearSearch(A[1 ... n], x):
Input: an array A[1 ... n] and an element x
Output: is x in the (possibly unsorted) array A?
1 for i := 1 to n:
2     if A[i] = x then
3         return True
4 return False
```

(a) Linear Search.



(c) An illustration of the first split in binary search.

```
binarySearch(A[1 ... n], x):
Input: a sorted array A[1 ... n]; an element x
Output: is x in the (sorted) array A?
1  lo := 1
2  hi := n
3  while lo ≤ hi:
4      middle := ⌊(lo+hi)/2⌋
5      if A[middle] = x then
6          return True
7      else if A[middle] > x then
8          hi := middle − 1
9      else
10         lo := middle + 1
11 return False
```

(b) The pseudocode for Binary Search.

Figure 6.15 Linear and Binary Search.

Solution. The analysis is correct, but it gives a looser lower bound than can be shown: specifically, the running time of Linear Search is $\Omega(n)$, and not just $\Omega(1)$. If we call **linearSearch**$(A, 42)$ for an array $A[1 \ldots n]$ that does not contain the number 42, then the total number of steps required will be at least n, because every element of A is compared to 42. Performing n comparisons takes $\Omega(n)$ time.

Here is a terser writeup of the analysis of Linear Search:

Example 6.11: Linear Search.
The worst case for Linear Search is an array $A[1 \ldots n]$ that doesn't contain the element x. In this case, the algorithm compares x to all n elements of A, taking $\Theta(n)$ time.

Taking it further: When we're analyzing an algorithm $\mathcal{A}$'s running time, we can generally prove several different lower and upper bounds for $\mathcal{A}$. For example, we might be able to prove that the running time is $\Omega(1)$, $\Omega(\log n)$, $\Omega(n)$, $O(n^2)$, and $O(n^3)$. The bound $\Omega(1)$ is a *loose bound*, because it is superseded by the bound $\Omega(\log n)$. (That is, if $f(n) = \Omega(\log n)$ then $f(n) = \Omega(1)$.) Similarly, $O(n^3)$ is a loose bound, because it is implied by $O(n^2)$. (Example 6.10 established a loose lower bound on the running time of linear search; it was superseded by the result in Example 6.11.)

We seek asymptotic bounds that are as tight as possible—so we always want to prove $f(n) = \Omega(g(n))$ and $f(n) = O(h(n))$ for the fastest-growing function g and slowest-growing function h that we can. If $g = h$, then we have proven a *tight bound*, or, equivalently, that $f(n) = \Theta(g(n))$. Sometimes there are algorithms for which we don't know a tight bound; we can prove $\Omega(n)$ and $O(n^2)$, but the algorithm might be $\Theta(n)$ or $\Theta(n^2)$ or $\Theta(n \log n \log\log \log n)$ or whatever. In general, we want to give upper and lower bounds that are as close together as possible.

Now let's analyze the running time of Binary Search:

Example 6.12: Binary Search.
Here is the intuition: in every iteration of the **while** loop in binary search, we halve the range of elements under consideration. (In other words, we halve $|\{i : lo \le i \le hi\}|$.) We can halve a set of size n only $\log_2 n$ times, and therefore binary search completes $O(\log_2 n)$ iterations of the **while** loop. Each of those iterations takes a constant amount of time, and therefore the total running time is $O(\log n)$.

Let's translate this intuition into a somewhat more formal proof. Suppose that the range of elements under consideration at the beginning of an iteration of the **while** loop is $A[lo, \ldots, hi]$. Let k denote the number of elements in this range—that is, $k = hi - lo + 1$. There are $\lceil \frac{k}{2} \rceil - 1$ elements in $A[lo, \ldots, middle - 1]$. There are $\lfloor \frac{k}{2} \rfloor$ elements in $A[middle + 1, \ldots, hi]$. (See Figure 6.15c.) Thus, after comparing x to $A[middle]$, one of three things happens:

- we find $x = A[middle]$, and the algorithm terminates.
- we find $x < A[middle]$ and continue in the left part of the array, which contains $\lceil \frac{k}{2} \rceil - 1 \le \frac{k}{2}$ elements.
- we find $x > A[middle]$ and continue in the right part of the array, which contains $\lfloor \frac{k}{2} \rfloor \le \frac{k}{2}$ elements.

In all three cases, we have at most $\frac{k}{2}$ elements under consideration in the next iteration of the loop.

Initially, there are n elements under consideration. Therefore, the above argument implies that there are at most $n/2^i$ elements left under consideration after i iterations. (This claim can be proven by induction.) So, after $\log_2 n$ iterations, there is at most one element left under consideration. Once the range contains only one element, we complete at most one more iteration of the **while** loop. Thus the total number of iterations is at most $1 + \log_2 n$. Each iteration takes a constant number of steps, and thus the total running time is $O(\log n)$.

Notice that analyzing the running time of any single iteration of the **while** loop in the algorithm was not too hard; the challenge in determining the running time of **binarySearch** lies in figuring out how many iterations occur.

Here we have only shown an upper bound on Binary Search's running time; in Example 6.26, we'll prove that Binary Search takes $\Omega(\log n)$ time, too. (Just as for Linear Search, the worst-case input for Binary Search is an n-element array that does not contain the sought value x.)

6.3.2 Some Other Types of Analysis

So far we have focused on asymptotically analyzing the worst-case running time of algorithms. While this type of analysis is the one most commonly used in the analysis of algorithms, there are other interesting types of questions that we can ask about algorithms. We'll sketch two of them in this section: instead of being completely pessimistic about the particular input that we get, we might instead consider either the *best* possible case or the "average" case.

Best-case analysis of running time

Best-case running time simply replaces the "max" from Definition 6.15 with a "min":

Definition 6.16: Best-case running time of an algorithm.
The *best-case running time of an algorithm* $\mathcal{A}$ on an input of size n is

$$T_{\mathcal{A}}^{\text{best}}(n) = \min_{x:|x|=n} \left[\text{the number of primitive steps used by } \mathcal{A} \text{ on input } x\right].$$

Best-case analysis is rarely used; knowing that an algorithm *might* be fast (on inputs for which it is particularly well tuned) doesn't help much in drawing generalizable conclusions about its performance (on the input that it's actually called on). (Ambrose Bierce (1842–≈1913) defined "optimism" in his *The Devil's Dictionary* (1911) as "the doctrine or belief that everything is beautiful, including what is ugly" [14]. Best-case analysis adheres to that definition more or less perfectly.)

Average-case analysis of running time

The "average" running time of an algorithm $\mathcal{A}$ is subtler to state formally, because "average" means that we have to have a notion of which values are more or less likely to be chosen as inputs. (For example,

consider sorting. In many settings, an already-sorted array is the most common input type to the sorting algorithm; a programmer might just want to "make sure" that the input was sorted, even though they might have been pretty confident that it already was.) The simplest way to do average-case analysis is to consider inputs that are chosen *uniformly at random* from the space of all possible inputs. For example, for sorting algorithms, we would consider each of the $n!$ different orderings of $\{1, 2, \ldots, n\}$ to be equally likely inputs of size n.

Definition 6.17: Average-case running time of an algorithm.
Let X denote the set of all possible inputs to an algorithm $\mathcal{A}$. The *average-case running time of an algorithm $\mathcal{A}$ for a uniformly chosen input* of size n is

$$T_{\mathcal{A}}^{\text{avg}}(n) = \frac{1}{|\{y \in X : |y| = n\}|} \cdot \sum_{x \in X : |x| = n} \big[\text{number of primitive steps used by } \mathcal{A} \text{ on } x\big].$$

Taking it further: Let ρ_n be a probability distribution over $\{x \in X : |x| = n\}$—that is, ρ_n is a function where $\rho_n(x)$ denotes the fraction of the time that the input to $\mathcal{A}$ is x, out of all of the times that the input to $\mathcal{A}$ is of size n. Definition 6.17 considers the uniform distribution, where $\rho_n(x) = 1/|\{x \in X : |x| = n\}|$.

The average-case running time of $\mathcal{A}$ on inputs of size n is the *expected running time* of $\mathcal{A}$ for an input x of size n chosen according to the probability distribution ρ_n. We will explore both probability distributions and expectation in detail in Chapter 10, which is devoted to probability. (If someone refers to the average case of an algorithm without specifying the probability distribution ρ, then they probably mean that ρ is the uniform distribution, as in Definition 6.17.)

We will still consider the asymptotic behavior of the best-case and average-case running times, for the same reasons that we are generally interested in the asymptotic behavior in the worst case.

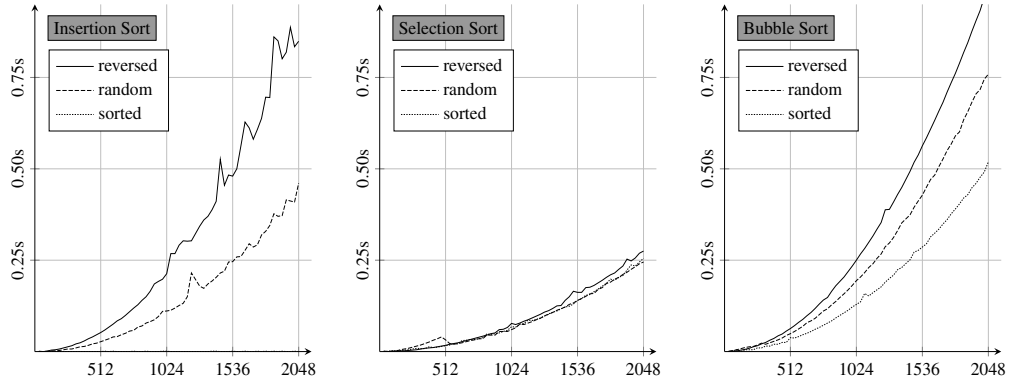
Best- and average-case analysis of sorting algorithms

We'll close this section with the best- and average-case analyses of our three sorting algorithms. (See Figure 6.16 for a reminder of the algorithms.)

Example 6.13: Insertion Sort, best- and average-case.
In Example 6.8, we showed that the worst-case running time of Insertion Sort is $\Theta(n^2)$. Let's analyze the best- and average-case running times of Insertion Sort.

The best-case running time for Insertion Sort is much faster than the worst-case running time: if the input array is already in sorted order, the inner **while** loop that swaps each $A[i]$ into place terminates immediately without doing any swaps, because $A[i] > A[i-1]$. Each iteration of the **for** loop therefore takes $\Theta(1)$ time, so the total running time is $\Theta(n)$.

We will defer a fully formal analysis of the average-case running time of Insertion Sort to Chapter 10 (see Example 10.47), but here is an informal analysis. Consider iteration the ith of the **for** loop. When that iteration starts, the first $i-1$ elements of A—that is, $A[1, \ldots, i-1]$—are in sorted order. The next element $A[i]$ has an equal chance of falling into any one of the i "slots" in the sorted subarray $A[1, \ldots, i-1]$: before




```
insertionSort(A[1 . . . n])
1 for i := 2 to n:
2     j := i
3     while j > 1 and A[j] < A[j − 1]:
4         swap A[j] and A[j − 1]
5         j := j − 1
```

```
selectionSort(A[1 . . . n])
1 for i := 1 to n:
2     minIndex := i
3     for j := i + 1 to n:
4         if A[j] < A[minIndex] then
5             minIndex := j
6     swap A[i] and A[minIndex]
```

```
bubbleSort(A[1 . . . n])
1 for i := 1 to n:
2     for j := 1 to n − i:
3         if A[j] > A[j + 1] then
4             swap A[j] and A[j + 1]
```

Figure 6.16 The elapsed-time running time for Insertion, Selection, and Bubble Sorts, and a reminder of the pseudocode for each.

$A[1]$, between $A[1]$ and $A[2]$, ..., between $A[i-2]$ and $A[i-1]$, and after $A[i-1]$. On average, then, the number of swaps in the ith iteration of the **for** loop is $\frac{i}{2}$. Thus the total average running time will be $\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4}$, which is $\Theta(n^2)$.

While we typically use formal mathematical analysis to address the performance of algorithms (whether we're interested in the worst, best, or average case), sometimes a kind of empirical analysis—where we measure an algorithm's performance by running it on an actual computer on an actual input and measuring how much time elapses before the algorithm terminates—can also be useful.

Figure 6.16 shows the elapsed time on an aging laptop during executions of my Python implementations of Insertion, Selection, and Bubble Sorts on a sorted array, a reverse-sorted array, and a randomly shuffled array. For Insertion Sort, Figure 6.16 confirms the formal analysis from Example 6.13: Insertion Sort's worst case is about twice as slow as its average case, and both are $\Theta(n^2)$; the best case of Insertion Sort is virtually invisible along the x-axis. On the other hand, Figure 6.16 suggests that Selection Sort's performance does not seem to depend very much on the structure of its input. Let's analyze this algorithm formally:

Example 6.14: Selection Sort, best- and average-case.
In Selection Sort, the only effect of the input array's structure is the number of times that *minIndex* is updated. (That's why the reverse-sorted input tends to perform ever-so-slightly worse in Figure 6.16.)

Thus the best- and average-case running time of Selection Sort is $\Theta(n^2)$, just like the worst-case running time established in Example 6.7.

Figure 6.16 suggests that Bubble Sort's performance varies only by a constant factor and, indeed, the worst-, average-, and best-case running times are all $\Theta(n^2)$:

Example 6.15: Bubble Sort, best- and average-case.
Again, the only difference in running time based on the structure of the input array is in how many times the body of the inner conditional is executed—that is, how many swaps occur. (The number of swaps ranges between 0 for a sorted array and $\frac{n(n-1)}{2}$ for a reverse-sorted array.) But there are $\Theta(n^2)$ comparisons of adjacent elements in any case, and $\Theta(n^2) + 0$ and $\Theta(n^2) + n^2$ are both $\Theta(n^2)$.

Careful examination (and some small tweaks) of Bubble Sort shows that we can improve the algorithm's best-case performance without affecting the worst- and average-case performance asymptotically; see Exercise 6.65.

Taking it further: The tools from this chapter can be used to analyze the consumption of any resource by an algorithm. So far, the only resource that we have considered is *time*: how many primitive steps are used by the algorithm on an particular input? The other resource whose consumption is most commonly analyzed is the *space* used by the algorithm—that is, the amount of memory it uses. As with time, we almost always consider the worst-case space use of the algorithm. See p. 6-35 for more on the subfield of CS called *computational complexity*, which seeks to understand the resources required to solve any particular problem. While time and space are the resources most frequently analyzed by complexity theorists, there are other resources that are interesting to track, too. For example, *randomized algorithms* "flip coins" as they run—that is, they make decisions about how to continue based on a randomly generated bit. Generating a truly random bit is expensive, and so we can view randomness itself as a resource, and try to minimize the number of random bits used. And, particularly in mobile processors, *power consumption*—and therefore the amount of battery life consumed, and the amount of heat generated—may be a more limiting resource than time or space. Thus energy can also be viewed as a resource that an algorithm might consume. (For some of the research on power-aware computing from an architecture perspective, see [70].)

Computer Science Connections

Multitasking, Garbage Collection, and Wall Clocks

One reason that we typically measure the running time of algorithms by counting (asymptotically) the number of primitive operations consumed by the algorithm on (worst-case) inputs is that measuring running time by so-called *wall-clock time* can be difficult to interpret—and potentially misleading.

All modern operating systems (everything that's been widely deployed for several decades: Windows, MacOS, Linux, iOS, Android, ...) are *multitasking* operating systems. That is, the user is typically running many applications simultaneously—perhaps an application to play music, a web browser, a programming environment, a word processor, a virus checker, and that sorting program that you wrote for your CS class. While it appears to the user that these applications are all running simultaneously, the operating system is actually pulling off a trick. There's typically only one processor (or a small number of processors, in multicore machines), and the operating system uses *time-sharing* to allow each running application to have a "turn" using the processor. (When it's the next application's turn and there's no currently idle processor, the operating system *swaps out* one application, and *swaps in* the next one that gets a slice of time on the processor.) If there were more processes running when you ran Merge Sort than when you ran Bubble Sort, then the elapsed time for Merge Sort could look worse than it should.

Many operating systems can report the total amount of processor time that a particular process consumed, so we can avoid the multitasking concern—but even within a single process, total processor time consumed can be misleading. While a program in Python or Java, for example, is running, periodically the *garbage collector* runs to reclaim "garbage" memory (previously allocated memory that won't be used again) for future use. When the garbage collector runs, the code that you were executing stops running. (See p. 11-51.)

Figure 6.17 shows the elapsed time while running four sorting algorithms, written in Python, executed on sorted inputs $[1, 2, \ldots, n]$, reverse sorted inputs $[n, n-1, \ldots, 1]$, and a randomly permuted n-element array. The "spikiness" of the elapsed times within the reverse-sorted plots may be because I launched a large presentation-editing application while the Insertion Sort test was running on inputs in descending sorted order, or because the garbage collector happened to start running during those trials.

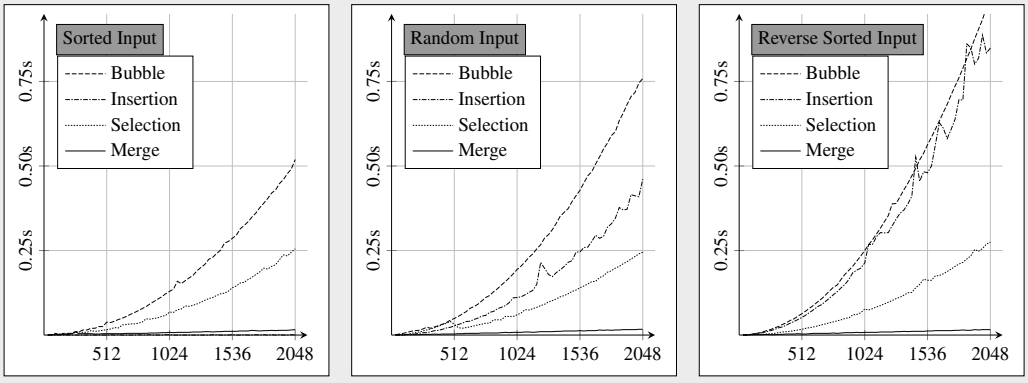


Figure 6.17 The wall-clock running time of four sorting algorithms on three different types of input. For n-element inputs of each type, the plot shows the number of seconds elapsed for the given sorting algorithms.

Even putting aside the difficulty of measuring running times accurately, there's another fundamental issue that we must address: we have to decide *on what* inputs to run the algorithms. The three panels of Figure 6.17 show why this choice can be significant. When the input is in sorted order, Insertion Sort is the best algorithm (in fact, it's barely visually distinguishable from the x-axis!). When the input is in reverse sorted order, Insertion Sort is terrible, and Merge Sort is the fastest. When the input is randomized, Insertion Sort is somewhere in the middle, and Merge Sort is again the fastest. Selection Sort is essentially unaffected by which type of input we consider.

The fact that we get such different pictures from the three different input types says that we have to decide which input to consider. (Typically we choose *the worst-case input for the particular algorithm,* as we've discussed.)

Computer Science Connections

Time, Space, and Complexity

Computational complexity is the subfield of computer science devoted to the study of the resources required to solve computational problems. Computational complexity is the domain of the most important open question in all of computer science, the P-versus-NP problem. That problem is described elsewhere in this book (see p. 3-32), but here we'll describe some of the basic entities that are studied by complexity theorists. (For much more, see any good textbook on computational complexity (also known as complexity theory), such as [120, 96].)

A *complexity class* is a set of problems that can be solved using a given constraint on resources consumed. Those resources are most typically the *time* or *space* used by an algorithm that solves the problem. For example, the complexity class EXPTIME includes precisely those problems solvable in exponential time—that is, $O(2^{n^k})$ time for some constant integer k. (There's a wide range of other resources, other constraints, or other models of computation that are studied by complexity theorists. For example, what extra power—if any—comes from using a computer that can make truly random choices as part of its execution? Or, what extra power—if any—comes from using a quantum computer instead of a classical one?)

One of the most important complexity classes is P, which denotes the set of all problems Π for which there is a polynomial-time algorithm $\mathcal{A}$ that solves Π. In other words,

$$\Pi \in \mathsf{P} \Leftrightarrow \text{there exists an algorithm } \mathcal{A} \text{ and an integer } k \in \mathbb{Z}^{\geq 0} \text{ such that } \mathcal{A} \text{ solves } \Pi \text{ and the worst-case running time of } \mathcal{A} \text{ on an input of size } n \text{ is } O(n^k).$$

Although the practical efficiency of an algorithm that runs in time $\Theta(n^{1000})$ is highly suspect, it has turned out that essentially any (non-contrived) problem that has been shown to be in P has actually also had a reasonably efficient algorithm—almost always $O(n^5)$ or better. As a result, one might think of the entire subfield of CS devoted to algorithms as really being devoted to understanding what problems can be solved in polynomial time. (Of course, improving the exponent of the polynomial is always a goal!)

Other complexity classes that are commonly studied are defined in terms of the space (memory) that they use:

PSPACE consists of those problems solvable using a polynomial amount of space.

L consists of those problems solvable using $O(\log n)$ space (beyond the input itself).

EXPSPACE consists of problems solvable using a exponential amount of space.

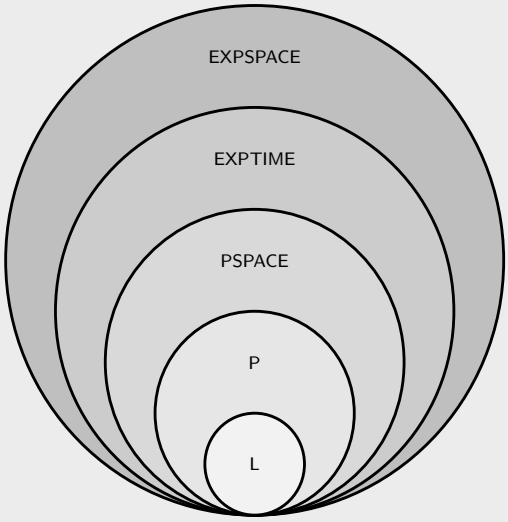


Figure 6.18 A few complexity classes, and their relationships.

While a great deal of effort has been devoted to complexity theory over the last half century, surprisingly little is known about how much time or space is actually required to solve problems—including some very important problems! It is reasonably easy to prove the relationships among the complexity classes shown in Figure 6.18, namely

$$\mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME} \subseteq \mathsf{EXPSPACE}.$$

Although the proofs are trickier, it has also been known since the 1960s that $\mathsf{P} \neq \mathsf{EXPTIME}$ (using the "time hierarchy theorem"), and that both $\mathsf{L} \neq \mathsf{PSPACE}$ and $\mathsf{PSPACE} \neq \mathsf{EXPSPACE}$ (using the "space hierarchy theorem"). But that's just about all that we know about the relationship among these complexity classes! For example, for all we know $\mathsf{L} = \mathsf{P}$ or $\mathsf{P} = \mathsf{PSPACE}$—but not both, because we *do* know that $\mathsf{L} \neq \mathsf{PSPACE}$. These foundational complexity-theoretic questions remain open—awaiting the insights of a new generation of computer scientists!

EXERCISES

A comparison-based *sorting algorithm reorders its input array $A[1 \ldots n]$ with two fundamental operations:*

- *the* comparison *of a pair of elements (to determine which one is bigger); and*
- *the* swap *of a pair of elements (to exchange their positions in the array).*

See Figure 6.19 for a reminder of three comparison-based sorting algorithms (Selection, Insertion, and Bubble Sort), with comparisons and swaps highlighted. For the worst-case input array $A[1 \ldots n]$ of size n, how many of the listed operations are done by each? Give an exact *answer, not an asymptotic one, and prove your answer.*

6.55 (worst-case) comparisons for **selectionSort**

6.56 (worst-case) comparisons for **insertionSort**

6.57 (worst-case) comparisons for **bubbleSort**

6.58 (worst-case) swaps for **selectionSort**

6.59 (worst-case) swaps for **insertionSort**

6.60 (worst-case) swaps for **bubbleSort**

Repeat Exercises 6.55–6.60 for the best *case: for the array $A[1 \ldots n]$ on which the given algorithm performs the best, how many comparisons/swaps does it do? (If the best-case array for swaps is different from the best-case array for comparisons, say so and explain why, and analyze the number of comparisons/swaps in the two different "best" arrays.)*

6.61 What is the best-case number of comparisons for **selectionSort**$(A[1 \ldots n])$? The best-case number of swaps?

6.62 What is the best-case number of comparisons for **insertionSort**$(A[1 \ldots n])$? The best-case number of swaps?

6.63 What is the best-case number of comparisons for **bubbleSort**$(A[1 \ldots n])$? The best-case number of swaps?

Two variations of the basic **bubbleSort** *algorithm are shown in Figure 6.20. In the next few exercises, you'll explore whether they're asymptotic improvements.*

```
selectionSort(A[1...n])
1 for i := 1 to n:
2     minIndex := i
3     for j := i + 1 to n:
4         if A[j] < A[minIndex] then
5             minIndex := j
6     swap A[i] and A[minIndex]
```

```
insertionSort(A[1...n])
1 for i := 2 to n:
2     j := i
3     while j > 1 and A[j] < A[j − 1]:
4         swap A[j] and A[j − 1]
5         j := j − 1
```

```
bubbleSort(A[1...n])
1 for i := 1 to n:
2     for j := 1 to n − i:
3         if A[j] > A[j + 1] then
4             swap A[j] and A[j + 1]
```

Figure 6.19 A reminder of three sorting algorithms, with swaps and comparisons of elements in the input array highlighted. Count as a "swap" any execution of the swap operation, including the exchange of an element with itself: the last line of **selectionSort** still counts as performing a swap even if $i = minIndex$.

```
early-stopping-bubbleSort(A[1...n]):
1 for i := 1 to n:
2     swapped := False
3     for j := 1 to n − i:
4         if A[j] > A[j + 1] then
5             swap A[j] and A[j + 1]
6             swapped := True
7     if swapped = False then
8         return A
```

```
forward-backward-bubbleSort(A[1...n]):
1 Construct R[1...n], where R[i] := A[n − i + 1] for each i.
2 for i := 1 to n:
3     Run one iteration of lines 2–8 of early-stopping-bubbleSort on A.
4     Run one iteration of lines 2–8 of early-stopping-bubbleSort on R.
5     if either A or R is now sorted then
6         return whichever is sorted
```

Figure 6.20 Two potentially improving variations on Bubble Sort: one that halts once no more swaps are possible, and one that that simultaneously works on the input and its reverse.

6.64 What's the worst-case running time of **early-stopping-bubbleSort**? Prove your answer.

6.65 Show that the *best-case* running time of **early-stopping-bubbleSort** is asymptotically better than the best-case running time of **bubbleSort**.

6.66 Show that the running time of **forward-backward-bubbleSort** on a reverse-sorted array $A[1 \ldots n]$ is $\Theta(n)$. (The reverse-sorted input is the worst case for both **bubbleSort** and **early-stopping-bubbleSort**.)

6.67 Prove that the worst-case running time of **forward-backward-bubbleSort** is $O(n^2)$.

6.68 Prove that—despite the apparent improvement—the worst-case running time of **forward-backward-bubbleSort** is $\Omega(n^2)$. To prove this claim, explicitly describe an array $A[1 \ldots n]$ for which **early-stopping-bubbleSort** requires $\Omega(n^2)$ on *both* A and the reverse of A.

6.69 *(programming required.)* Implement the three versions of Bubble Sort (including the two in Figure 6.20) in a programming language of your choice.

6.70 *(programming required.)* Modify your implementations from Exercise 6.69 to count the number of swaps and comparisons performed. Then run all three algorithms on each of the $8! = 40{,}320$ different orderings of $\{1, 2, \ldots, 8\}$. How do the algorithms' performances compare, on average?

In Chapter 9 (see p. 9-26), we will meet a sorting algorithm called Counting Sort *that sorts an array $A[1 \ldots n]$* where each $A[i]$ is an element of $\{1, 2, \ldots, k\}$ *as follows: for each possible value $x \in \{1, 2, \ldots, k\}$, we walk through A to compute the number c_x of occurrences of the value x—that is, we compute $c_x := |\{i : A[i] = x\}|$. (We can compute all k values of $c_1, \ldots, c_k$ in a single pass through A.) The output array consists of c_1 copies of 1, followed by c_2 copies of 2, and so forth, ending with c_k copies of k. (See Figure 6.21.) Counting sort is particularly good when k is small.*

6.71 In terms of n, what is the worst-case running time of **countingSort** on an input array of n letters from the alphabet (so $k = 26$, and n is arbitrary)?

6.72 *(programming required.)* Implement Counting Sort and one of the $\Theta(n^2)$-time sorting algorithms from this section. Collect some data to determine, on a particular computer, for what values of k you'd generally prefer Counting Sort over the $\Theta(n^2)$-time algorithm when $n = 4096 = 2^{12}$ elements are each chosen uniformly at random from $\{1, 2, \ldots, k\}$.

6.73 *Radix Sort* is a sorting algorithm based on Counting Sort that proceeds by repeatedly applying Counting Sort to the ith-most significant bit in the input integers, for increasing i. Do some online research to learn more about Radix Sort, then write pseudocode for Radix Sort and compare its running time (in terms of n and k) to Counting Sort.

In Example 5.14, we proved the correctness of Quick Sort, *a recursive sorting algorithm (see Figure 6.21). The basic idea is to choose a* pivot *element of the input array A, then* partition *A into those elements smaller than the pivot and those elements larger than the pivot. We can then recursively sort the two "halves" and paste them together, around the pivot, to produce a sorted version of A. The algorithm performs very well if the two "halves" are genuinely about half the size of A; it performs very poorly if one*

```
countingSort(A[1...n]) :
// Assume that each A[i] ∈ {1, 2, ..., k}
for v := 1 to k:
    count[v] := 0
for i := 1 to n:
    count[A[i]] := count[A[i]] + 1
i := 1
for v := 1 to k:
    for t := 1 to count[v]:
        A[i] := v
        i := i + 1
```

```
quickSort(A[1...n]):
if n ≤ 1 then
    return A
else
    Choose pivotIndex ∈ {1, ..., n}, somehow.
    Let less (those elements smaller than A[pivotIndex]), same and greater
    be empty arrays.
    for i := 1 to n:
        compare A[i] to A[pivotIndex], and append A[i] to the appropriate
        array less, same, or greater.
    return quickSort(less) + same + quickSort(greater).
```

Figure 6.21 Counting Sort, and a high-level reminder of Quick Sort (see Figure 5.19a for more detail).

"half" contains almost all the elements of A. The running time of the algorithm therefore hinges on how we select the pivot, in Line 4. (A very good choice of pivot is actually a random *element of A, but here we'll think only about deterministic rules for choosing a pivot.)*

6.74 Suppose that we always choose $pivotIndex := 1$. (That is, the first element of the array is the pivot value.) Describe (for an arbitrary n) an input array $A[1 \ldots n]$ that causes **quickSort** under this pivot rule to make either *less* or *greater* empty.

6.75 Argue that, for the array you found in Exercise 6.74, the running time of Quick Sort is $\Theta(n^2)$.

6.76 Suppose that we always choose $pivotIndex := \lfloor \frac{n}{2} \rfloor$. (That is, the middle element of the array is the pivot value.) What input array $A[1 \ldots n]$ causes worst-case performance (that is, one of the two sides of the partition—*less* or *greater*—is empty) for this pivot rule?

6.77 A fairly commonly used pivot rule is called the *Median of Three* rule: we choose $pivotIndex \in \{1, \lfloor \frac{n}{2} \rfloor, n\}$ so that $A[pivotIndex]$ is the median of the three values $A[1]$, $A[\lfloor \frac{n}{2} \rfloor]$, and $A[n]$. Argue that there is still an input array of size n that results in $\Omega(n^2)$ running time for Quick Sort.

6.78 Earlier we described a linear-search algorithm that looks for an element x in an array $A[1 \ldots n]$ by comparing x to $A[i]$ for each $i = 1, 2, \ldots n$. (See Figure 6.15a.) But if A is sorted, we can determine that x is not in A earlier, as shown in Figure 6.22: once we've passed where x "should" be, we know that it's not in A. (Our original version omitted lines 4–5.) What is the worst-case running time of the early-stopping version of linear search?

6.79 Consider the algorithm in Figure 6.22 for counting the number of times the letter Z appears in a given string s. What is the worst-case running time of this algorithm on an input string of length n?

```
early-stopping-linearSearch(A[1...n], x)
1 for i := 1 to n:
2     if A[i] = x then
3         return True
4     else if A[i] < x then
5         return False
6 return False
```

```
countZ(s):                                  Assume that (i) testing whether Z is in s
                                            and (ii) removing a single letter from s
                                            both take c · |s| time, for some constant c.
1 z := 0
2 while there exists i such that s_i = Z:
3     z := z + 1
4     remove s_i from s                     // that is, set s := s_1 ... s_{i-1} s_{i+1} ... s_n
5 return z
```

Figure 6.22 An early-stopping variations on Linear Search, and an algorithm for counting ZZZs.

6.4 Recurrence Relations: Analyzing Recursive Algorithms

> What did this mean? Who was I? What was I? Whence did I come? What was my destination? These questions continually recurred, but I was unable to solve them.
>
> Mary Wollstonecraft Shelley (1797–1851)
> *Frankenstein* (1818)

The nonrecursive algorithms in Section 6.3 could be analyzed "just" by counting and manipulation of summations. (Sometimes the summations are easier to handle, and sometimes they're harder—but they're just summations.) First we figured out the number of iterations of each loop, and then figured out how long each iteration takes. By summing this work over the iterations and simplifying the summation, we were able to compute the running time of the algorithm. Determining the running time of a recursive algorithm is harder. Instead of "merely" containing loops that can be analyzed in this way, the algorithm's running time on an input of size n depends on the same algorithm's running time for inputs of size smaller than n.

We'll use the classical recursive sorting algorithm Merge Sort (Figure 6.23) as an example. Merge Sort sorts an n-element array by recursively sorting the first half, recursively sorting the second half, and finally "merging" the resulting sorted lists. (On an input array of size 1, Merge Sort just returns the array as is.) Merging two $\frac{n}{2}$-element arrays takes $\Theta(n)$ time, but what does that mean for the overall running time of Merge Sort? We can think about Merge Sort's running time by drawing a picture of all of the work that is done in its execution, in the form of a *recursion tree*:

Definition 6.18: Recursion tree.
The *recursion tree* for a recursive algorithm $\mathcal{A}$ is a tree that shows all of the recursive calls spawned by a call to $\mathcal{A}$ on an input of size n. Each node in the tree is annotated with the amount of work, aside from any recursive calls, done by that call.

Figure 6.24 shows the recursion tree for Merge Sort. For ease, assume that n is an exact power of 2. Let $c \cdot n$ represent the amount of time needed to process an n-element array *aside from the recursive calls*—that is, the time to split and merge. There are many different ways to analyze the total amount of work done by Merge Sort on an n-element input array, but one of the easiest is to use the recursion tree itself:

```
mergeSort(A[1 . . . n]):
1 if n = 1 then
2     return A
3 else
4     L := mergeSort(A[1 . . . ⌊n/2⌋])
5     R := mergeSort(A[⌊n/2⌋ + 1 . . . n])
6     return merge(L, R)
```

The **merge** function combines two sorted arrays into a single sorted array. For example, **merge**([2, 4, 6, 8], [5, 7, 9, 11]) yields [2, 4, 5, 6, 7, 8, 9, 11]. You'll argue in Exercise 6.100 that merging two $\frac{n}{2}$-element arrays takes $\Theta(n)$ time.

Figure 6.23 Merge Sort.

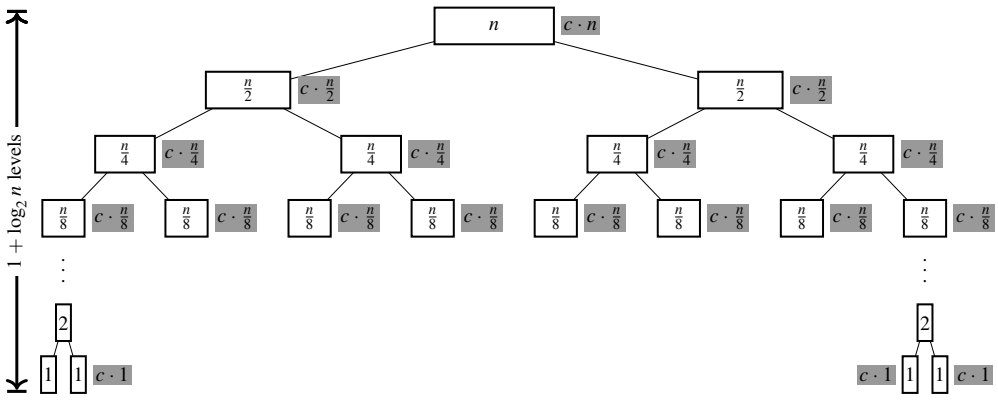


Figure 6.24 The recursion tree for Merge Sort. Each input's size is shown in the node; the linear amount of time for splitting/merging it is shown in the shaded box adjacent to the corresponding node.

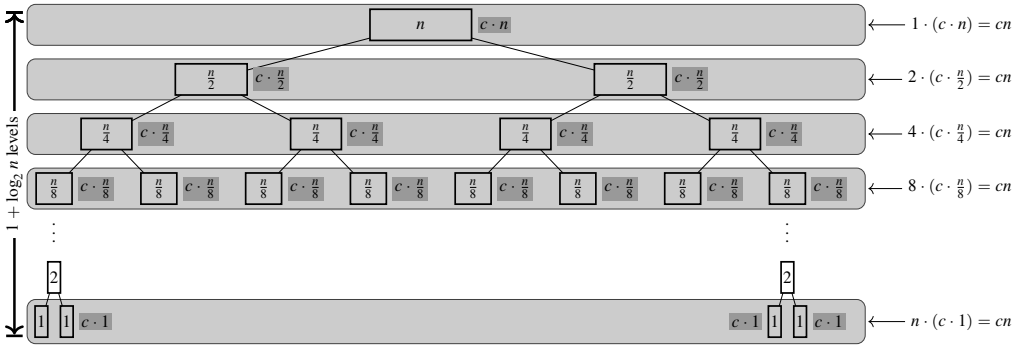


Figure 6.25 The row-wise sum of the tree in Figure 6.24.

Example 6.16: Analyzing Merge Sort via recursion tree.
How quickly does Merge Sort run on an n-element input array? (Assume that n is a power of two.)

Solution. Let's look at the recursion tree for Merge Sort. The total amount of work done by the algorithm is precisely the sum of the identified work—the quantities written in the shaded boxes—for each node in the tree. (That's exactly how $c \cdot n$ was defined.) The easiest way to sum up the work in the tree is to sum "row-wise" (see Figure 6.25):

- The first "row" of the tree (one call on an input of size n) generates cn work.
- The second row (two calls on inputs of size $\frac{n}{2}$) generates $2 \cdot (c \cdot \frac{n}{2}) = cn$ work.
- The third row (four calls on inputs of size $\frac{n}{4}$) generates $4 \cdot (c \cdot \frac{n}{4}) = cn$ work.

- In general, the kth row of the tree contains 2^{k-1} nodes, each of which represents a call to Merge Sort on an input of size $n/2^{k-1}$. Each of these calls is generates $c \cdot n/2^{k-1}$ work. In total, then, the kth row contains a total of $2^{k-1} \cdot (c \cdot n/2^{k-1}) = cn$ work—no matter what the value of k was.

Putting these pieces together: the total work done by Merge Sort on an n-element input array is

$$\begin{aligned}
&\sum_{k=1}^{1+\log_2 n} (\text{the work done in the } k\text{th row of the tree}) && \textit{there are } 1+\log_2 n \textit{ rows in the recursion tree} \\
&= \sum_{k=1}^{1+\log_2 n} \left[2^{k-1}\right] \cdot \left[c \cdot \tfrac{n}{2^{k-1}}\right] && \textit{there are } 2^{k-1} \textit{ nodes in row k, and each corresponds to } c \cdot \tfrac{n}{2^{k-1}} \textit{ work} \\
&= \sum_{k=1}^{1+\log_2 n} cn = cn(1+\log_2 n), && 2^{k-1} \cdot (c \cdot \tfrac{n}{2^{k-1}}) = cn \textit{ for any value of k (see above)}
\end{aligned}$$

and thus is $\Theta(n \log n)$ in total.

Taking it further: We'll be analyzing algorithms by examining recursion trees throughout this section, but—if you prefer—here's a different argument as to why Merge Sort requires $\Theta(n \log n)$ time. (This approach looks at the individual experience of a particular element that's being sorted, rather than looking at the entire array all at once.) *Every* individual element of the input array is merged once into an array of size 2, once into an array of size 4, once into an array of size 8, etc. So each element is merged $\log_2 n$ times, so thus the total work is $\Theta(n \cdot \log_2 n)$.

6.4.1 Recurrence Relations

Recursion trees are an excellent way to gain intuition about the running time of a recursive algorithm, and to analyze it. Here, we'll look at another way of thinking about recursion trees, which suggests a rigorous (and in many ways easier to use) approach to analyzing recursive algorithms: the *recurrence relation.* Because at least one of the steps in a recursive algorithm $\mathcal{A}$ is to call $\mathcal{A}$ on a smaller input, the running time of $\mathcal{A}$ on an input of size n depends on $\mathcal{A}$'s running time for inputs of size smaller than n. We will therefore express $\mathcal{A}$'s running time recursively, too:

Definition 6.19: Recurrence relation.
A *recurrence relation* (sometimes simply called a *recurrence*) is a function $T(n)$ that is defined (for some values of n) in terms of the values of $T(k)$ for input values $k < n$.

(The name comes from the fact that *T recurs* ("occurs again") on the right-hand side of the equation. That's the same reason that recursion is called recursion.) Here's a first example, about compounding interest:

Example 6.17: Compound interest.
Suppose that, in year 0, Alice puts \$1000 in a bank account that pays 2% annual compound interest. Writing $A(n)$ to denote the balance of Alice's account in year n, we have

$$A(0) = 1000 \qquad\qquad A(n) = 1.02 \cdot A(n-1).$$

If Bob opens a bank account with the same interest rate, and deposits \$10 into the account each year (starting in year 0), then Bob's balance is given by the recurrence

$$B(0) = 10 \qquad\qquad B(n) = 1.02 \cdot B(n-1) + 10.$$

We'll most frequently encounter recurrence relations in which $T(n)$ denotes the worst-case number of steps taken by a particular recursive algorithm on an input of size n. Here are a few examples:

Example 6.18: Factorial.
Let $T(n)$ denote the worst-case running time of **fact** (see Figure 6.26). Then

$$T(1) = d \qquad \text{and} \qquad T(n) = T(n-1) + c,$$

where c is a constant denoting the work of the comparison, conditional, multiplication, and return; and d is a constant denoting the work of the comparison, conditional, and return.

Example 6.19: Merge Sort.
Let $T(n)$ denote the worst-case running time of Merge Sort (see Figure 6.23) on an input array containing n elements. Then, for a constant c, we have:

$$T(1) = c \qquad \text{and} \qquad T(n) = T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + cn.$$

Taking it further: Just as for nonrecursive algorithms, we will generally be interested in the asymptotic running times of these recursive algorithms, so we will usually not fret about the particular values of the constants in recurrences. We will often abuse notation and use a single constant to represent different $\Theta(1)$-time operations, for example. In Example 6.19, for instance, we are being sloppy in our recurrence, using a single variable c to represent two different values. The use of one constant to have two different meanings (plus the '=' sign) is an abuse of notation, but when we care about asymptotic values, this abuse doesn't matter. We will even sometimes write 1 to stand for this constant. (See Exercise 6.126.)

Here's another recurrence relation, for the recursive version of Binary Search:

```
fact(n):
1 if n = 1 then
2     return 1
3 else
4     return n · fact(n − 1)
```

```
binarySearch(A, loIndex, hiIndex, x):
1 if loIndex > hiIndex then
2     return False
3 middle := ⌊(loIndex + hiIndex)/2⌋
4 if A[middle] = x then
5     return True
6 else if A[middle] > x then
7     return binarySearch(A, loIndex, middle − 1, x)
8 else
9     return binarySearch(A, middle + 1, hiIndex, x)
```

To avoid the inefficiency of copying portions of A when a recursive call is made, this code uses four parameters instead of two: the array A, the left- and right-most indices in A to search, and the sought element x. You'd call the algorithm **binarySearch**$(A[1 \ldots n], 1, n, x)$ to start the recursive search for x in A.

Figure 6.26 Two recursive algorithms: factorial and binary search.

Example 6.20: Binary Search.
Let $T(n)$ denote the worst-case running time of the recursive **binarySearch** (see Figure 6.26) on an n-element array. Then:

$$T(0) = c$$

$$T(n) = \begin{cases} T(\frac{n}{2}) + c & \text{if } n \text{ is even} \\ T(\frac{n-1}{2}) + c & \text{if } n \text{ is odd.} \end{cases}$$

Although our interest in recurrence relations will be almost exclusively about the running times of recursive algorithms, there are other interesting recurrence relations, too. The most famous of these is the recurrence for the *Fibonacci numbers* (which turn out to have some interesting CS applications, as we'll see):

Example 6.21: Fibonacci numbers.
The Fibonacci numbers are defined by

$$f_1 = 1$$

$$f_2 = 1$$

$$f_n = f_{n-1} + f_{n-2} \qquad \text{for } n \geq 3.$$

The first several Fibonacci numbers are $1, 1, 2, 3, 5, 8, 13, 21, 34, 55$, and 89.

6.4.2 Solving Recurrences: Induction

When we *solve* a recurrence relation, we find a closed-form (that is, nonrecursive) equivalent expression. Because recurrence relations are recursively defined quantities, induction is the easiest way to prove that a conjectured solution is correct. (The hard part is figuring out what solution to conjecture.) Here, we'll solve the recurrences from Section 6.4.1, starting with Alice and Bob and their bank accounts:

Example 6.22: Compound interest.
Let's solve the recurrences from Example 6.17:

$$A(0) = 1000 \qquad A(n) = 1.02 \cdot A(n-1) \qquad \textit{(Alice)}$$

$$B(0) = 10 \qquad B(n) = 1.02 \cdot B(n-1) + 10. \qquad \textit{(Bob)}$$

The recurrence for Alice is the easier of the two to solve: we can prove relatively straightforwardly by induction that $A(n) = 1000 \cdot 1.02^n$ for any $n \geq 1$.

For Bob, the analysis is a little trickier. Here's some intuition: in year n, Bob has had \$10 sitting in his account since year 0 (earning interest for n years); he has had \$10 in his account since year 1 (earning interest for $n-1$ years); and so forth. Ten dollars that has accumulated interest for i years has, as with

Alice, grown to $10 \cdot 1.02^i$. Thus the total amount of money in Bob's account in year n will be

$$\sum_{i=0}^{n} \left[10 \cdot 1.02^i\right] = 10 \cdot \sum_{i=0}^{n} 1.02^i \overset{\text{by Theorem 5.5 (the analysis of a geometric series)}}{=} 10 \cdot \tfrac{1.02^{n+1}-1}{1.02-1} = 510 \cdot 1.02^n - 500.$$

Using this intuition, let's prove that $B(n) = 510 \cdot 1.02^n - 500$, by induction on n:

Base case ($n = 0$). The recurrence defines $B(0)$ as 10, and indeed $510 \cdot 1.02^0 - 500 = 510 - 500 = 10$.

Inductive case ($n \geq 0$). We assume the inductive hypothesis, which states that $B(n-1) = 510 \cdot 1.02^{n-1} - 500$; we must show that $B(n) = 510 \cdot 1.02^n - 500$:

$$\begin{aligned}
B(n) &= 1.02 \cdot B(n-1) + 10 && \textit{definition of } B(n) \\
&= 1.02 \cdot \left[510 \cdot 1.02^{n-1} - 500\right] + 10 && \textit{inductive hypothesis} \\
&= 1.02 \cdot 510 \cdot 1.02^{n-1} - 1.02 \cdot 500 + 10 && \textit{multiplying through} \\
&= 510 \cdot 1.02^n - 510 + 10 && \textit{simplifying} \\
&= 510 \cdot 1.02^n - 500.
\end{aligned}$$

Taking it further: As Example 6.22 suggests, familiar summations like geometric series can be expressed using recurrence relations. Other familiar summations can also be expressed using recurrence relations; for example, the sum of the first n integers is given by the recurrence $T(1) = 1$ and $T(n) = T(n-1) + n$. (See Section 5.2 for some closed-form solutions.)

Factorial

One good way to generate a conjecture (to then prove correct by induction) is by "iterating" the recurrence: expand out a few layers of the recursion to see the values of $T(n)$ for a few small values of n. We'll illustrate this technique with the simplest recurrence from the last section, for the recursive factorial function.

Example 6.23: Factorial.
Recall the recurrence from Example 6.18:

$$T(1) = d \qquad T(n) = T(n-1) + c.$$

Give an exact closed-form (nonrecursive) solution for $T(n)$.

Solution. Let's iterate the recurrence a few times:

$$\begin{aligned}
T(1) &= d \\
T(2) &= c + T(1) = c + d \\
T(3) &= c + T(2) = 2c + d \\
T(4) &= c + T(3) = 3c + d
\end{aligned}$$

From the values of this recurrence for small n, we might conjecture that $T(n) = (n-1)c + d$. Let's prove this conjecture by induction.

For the base case ($n = 1$), we have $T(1) = d$ by definition of the recurrence, and $d = 0 \cdot c + d$, as desired.

For the inductive case ($n \geq 1$), assume the inductive hypothesis $T(n-1) = (n-2)c + d$. We want to show that $T(n) = (n-1)c + d$. Here's the proof:

$$\begin{aligned} T(n) &= T(n-1) + c && \textit{by definition of the recurrence} \\ &= [(n-2)c + d] + c && \textit{by the inductive hypothesis} \\ &= (n-1)c + d. && \textit{by algebraic manipulation} \end{aligned}$$

Thus $T(n) = (n-1)c + d$.

Problem-solving tip: Try iterating a recurrence to generate its first few values. Once we have a few values, we can often conjecture a general solution (which we then prove correct via induction).

Merge Sort

Recall the Merge Sort recurrence, where $T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + cn$ and $T(1) = c$. We'll solve this recurrence in this section. But it will be easier to address the case in which n is an exact power of 2 first (so that the floors and ceilings don't complicate the picture), so we'll start with that case:

Example 6.24: Merge Sort, for powers of two.
Give an exact closed-form (nonrecursive) solution for the Merge Sort recurrence from Example 6.19:

$$T(1) = c \qquad T(n) = T(\lceil \tfrac{n}{2} \rceil) + T(\lfloor \tfrac{n}{2} \rfloor) + cn.$$

Assume that n is an exact power of two.

Solution. Because n is an exact power of two, we can write $n = 2^k$ for some $k \in \mathbb{Z}^{\geq 0}$. (For an exact power of two, we have $\lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor = \frac{n}{2} = 2^{k-1}$.)

We'll solve the given recurrence by thinking about the recursive relationship involving k, instead of n. Specifically, define $R(k) = T(2^k)$. Then $R(0) = T(1) = c$ and $R(k) = T(2^k) = 2 \cdot T(2^{k-1}) + c \cdot 2^k = 2 \cdot R(k-1) + c \cdot 2^k$ for $k \geq 1$. Thus we can instead solve the recurrence

$$R(0) = c \qquad R(k) = 2 \cdot R(k-1) + c \cdot 2^k.$$

Iterating R a few times, we see

$$\begin{aligned} R(0) &= c \\ R(1) &= c \cdot 2^1 + 2 \cdot R(0) = 4c \\ R(2) &= c \cdot 2^2 + 2 \cdot R(1) = 12c \\ R(3) &= c \cdot 2^3 + 2 \cdot R(2) = 32c. \end{aligned}$$

On the basis of these values, we might conjecture

$$R(k) = (1+k)2^k \cdot c. \qquad (*)$$

How might we get to the conjecture $(*)$? The pattern from iterating R matches it, or looking at the recursion tree in Figure 6.25 might help; that tree had $1 + \log_2 n = 1 + k$ levels, with $c \cdot n = c \cdot 2^k$ work per level, or $(1+k)2^k \cdot c$ in total. Alternatively, we'd expect a solution that's the product of $\approx k$ and $\approx 2^k$ so that we get $T(n) \approx n \log n$; if we check the $k = 0$ case—$R(0) = 1$—it looks like we'd better multiply by $k+1$ rather than k.

Let's prove $(*)$, by induction on k.

In the base case, $R(0) = c$ by the definition of the recurrence, and indeed $(1+0)2^0 \cdot c = 1 \cdot 1 \cdot c$.

In the inductive case, we assume the inductive hypothesis $R(k-1) = k2^{k-1} \cdot c$. Then,

$$\begin{aligned} R(k) &= 2R(k-1) + c \cdot 2^k && \textit{by definition of the recurrence} \\ &= 2 \cdot [k \cdot 2^{k-1} \cdot c] + c \cdot 2^k && \textit{by the inductive hypothesis} \\ &= (k+1)2^k \cdot c. && \textit{factoring and simplifying} \end{aligned}$$

Thus $R(k) = (k+1)2^k \cdot c$, completing the inductive case—and the proof of $(*)$. Because we defined $R(k) = T(2^k)$, we can conclude that $T(n) = R(\log_2 n)$, by substituting. Therefore

$$T(n) = (1 + \log_2 n) \cdot 2^{\log_2 n} \cdot c = (1 + \log_2 n) \cdot n \cdot c.$$

Problem-solving tip: A useful technique for solving recurrences is to do a *variable substitution*. If you can express the recurrence in terms of a different variable and solve the new recurrence easily, you can then substitute back into the original recurrence to solve it. Transforming an unfamiliar recurrence into a familiar one will make life easy!

Thinking only about powers of two in Example 6.24 made our life simpler, but it means that our analysis was incomplete: what is the running time of Merge Sort when the input array's length is *not* precisely a power of two? The more general analysis is actually not too complicated, given the result we just derived:

Example 6.25: Merge Sort, for general n.
Solve the Merge Sort recurrence (asymptotically), for any integer $n \geq 1$:

$$T(1) = c \qquad T(n) = T(\lceil \tfrac{n}{2} \rceil) + T(\lfloor \tfrac{n}{2} \rfloor) + cn.$$

Solution. We'll use the fact that $T(n) \geq T(n')$ if $n \geq n'$—that is, T is *monotonic*. (See Exercise 6.101.) So let k be the nonnegative integer such that $2^k \leq n < 2^{k+1}$. Then

$$\begin{aligned} T(n) &\geq T(2^k) && \text{and} \quad T(n) < T(2^{k+1}) && \textit{monotonicity and } 2^k \leq n < 2^{k+1} \\ &= (1 + \log_2 2^k) \cdot 2^k \cdot c && = (1 + \log_2 2^{k+1}) \cdot 2^{k+1} \cdot c && \textit{Example 6.24} \\ &> (1 + \log_2 \tfrac{n}{2}) \cdot \tfrac{n}{2} \cdot c. && \leq (1 + \log_2 2n) \cdot 2n \cdot c && \tfrac{n}{2} < 2^k \textit{ and } 2n \geq 2^{k+1} \textit{ (by definition)} \\ &= \Omega(n \log n) && = O(n \log n). \end{aligned}$$

Combining these facts yields that $T(n) = \Theta(n \log n)$.

Binary Search

The logic that we used to argue that Binary Search takes logarithmic time in Example 6.12 is reasonably intuitive: *In the worst case, when the sought item x isn't in the array, we repeatedly compare x to the middle of the valid range of the array, and halve the size of that valid range. We can halve an n-element range exactly* $\log_2 n$ *times, and thus the running time of Binary Search is logarithmic.*

But, while this intuitive argument correctly establishes that Binary Search's running time is $O(\log n)$, there's a slightly subtle issue that we've glossed over: the so-called "halving" in this description isn't actually *exactly* halving. If there are n elements in the valid range, then after comparing x to the middle element of the range, we will end up with a worst-case valid range of size either $\frac{n}{2}$ or $\frac{n-1}{2}$, depending on the parity of n. (So it's sometimes, but not always, *exactly* $\frac{n}{2}$. The argument in Example 6.12 only relied on the fact that we end up with a valid range of size *at most* $\frac{n}{2}$. But we have not yet ruled out the possibility that the running time might be *faster* than $\Theta(\log n)$, if we "slightly better than halve" at every stage.)

We can resolve this issue by rigorously analyzing the correct recurrence relation—and we can prove that the running time *is* in fact $\Theta(\log n)$.

Example 6.26: Binary Search.
Solve the Binary Search recurrence:

$$T(0) = 1 \qquad T(n) = \begin{cases} T(\frac{n}{2}) + 1 & \text{if } n \text{ is even} \\ T(\frac{n-1}{2}) + 1 & \text{if } n \text{ is odd.} \end{cases}$$

(Note that we've changed the additive constants to 1 instead of c; changing it back to c would only have the effect of multiplying the entire solution by c.)

Solution. We conjecture that $T(n) = \lfloor \log_2 n \rfloor + 2$ for all $n \geq 1$. We'll prove the conjecture by strong induction on n.

For the base case ($n = 1$), we have $T(1) = T(0) + 1 = 1 + 1 = 2$ by definition of the recurrence, and indeed $2 = \lfloor 0 \rfloor + 2 = \lfloor \log_2 1 \rfloor + 2$.

For the inductive case ($n \geq 2$), assume the inductive hypothesis, namely that $T(k) = \lfloor \log_2 k \rfloor + 2$ for any $k < n$. We'll proceed in two cases:

Case I: n is even. Then
$T(n) = T(\frac{n}{2}) + 1$ *by definition of the recurrence*
$= \lfloor \log_2(\frac{n}{2}) \rfloor + 2 + 1$ *by the inductive hypothesis*
$= \lfloor (\log_2 n) - 1 \rfloor + 3$ *because* $\log(\frac{a}{b}) = \log a - \log b$, *and* $\log_2 2 = 1$
$= \lfloor \log_2 n \rfloor + 2.$ *because* $\lfloor x + 1 \rfloor = \lfloor x \rfloor + 1$

Case II: n is odd. Then $T(n) = T(\frac{n-1}{2}) + 1$ *by definition of the recurrence*

$$= \lfloor \log_2(\tfrac{n-1}{2}) \rfloor + 2 + 1 \qquad \textit{by the inductive hypothesis}$$
$$= \lfloor \log_2(n-1) \rfloor + 2 \qquad \textit{by the same manipulations as in the even case}$$
$$= \lfloor \log_2 n \rfloor + 2. \qquad \textit{because } \lfloor \log_2(n-1) \rfloor = \lfloor \log_2 n \rfloor \textit{ for any odd integer } n > 1$$

Because $T(n) = \lfloor \log_2 n \rfloor + 2$ in both cases, we've proven the claim. Therefore $T(n) = \Theta(\log n)$.

Problem-solving tip: When solving a new recurrence, we can generate conjectures (to prove correct via induction) by iterating the recurrence, drawing out the recursion tree, or by straight-up guessing a solution (or recognizing a familiar pattern). To generate my conjecture for Example 6.26, I wrote a program implementing the recurrence. I ran the program for $n \in \{1, 2, \ldots, 2000\}$ and printed out the smallest integer n for which $T(n) = 1$, then the smallest for which $T(n) = 2$, etc. (See Figure 6.10 for a graph of the function.) The conjecture followed from the observation that the breakpoints all happened at $n = 2^k - 1$ for an integer k.

As a general matter, the appearance of floors and ceilings inside a recurrence won't matter to the asymptotic running time, nor will small additive adjustments inside the recursive term. For example, $T(n) = T(\lceil \frac{n}{2} \rceil) + 1$ and $T(n) = T(\lfloor \frac{n}{2} \rfloor - 2) + 1$ both have $T(n) = \Theta(\log n)$ solutions. Typically, understanding the running time for the "pure" version of the recurrence will give a correct understanding of the more complicated version. As such, we'll often be sloppy in our notation, and write $T(n) = T(\frac{n}{2}) + 1$ when we really mean $T(\lfloor \frac{n}{2} \rfloor)$ or $T(\lceil \frac{n}{2} \rceil)$. (This abuse of notation is fairly common.)

Taking it further: Intuitively, floors and ceilings don't change this type of recurrence because they don't affect the total depth of the recursion tree by more than a $\Theta(1)$ number of calls, and a $\Theta(1)$ difference in depth is asymptotically irrelevant. There's a general theorem called the *"sloppiness" theorem*, which states precise conditions under which it is safe to ignore floors and ceilings in recurrence relations. (As long as we actually prove inductively that our conjectured solution to a recurrence relation is correct, it's always fine in generating conjectures.) As a rough guideline, as long as $T(n)$ is monotonic (if $n \leq n'$, then $T(n) \leq T(n')$) and doesn't grow too quickly ($T(n)$ is $O(n^k)$ for some constant k), then this "sloppiness" is fine. The details of the theorem, and its precise assumptions, are presented in many algorithms textbooks; see [33] for a full proof, for example.

6.4.3 The Fibonacci Numbers

We'll close with another example of a recurrence relation—the Fibonacci recurrence—that we will analyze using induction. But this time we will solve the recurrence exactly (that is, nonasymptotically):

Example 6.27: The Fibonacci Numbers.
Recall the *Fibonacci numbers*, defined by the recurrence

$$f_1 = 1 \qquad f_2 = 2 \qquad f_n = f_{n-1} + f_{n-2}.$$

Prove that f_n grows exponentially: that is, prove that there exist $a \in \mathbb{R}^{>0}$ and $r \in \mathbb{R}^{>1}$ such that $f_n \geq ar^n$.

Some brainstorming: Let's start in the middle of a hypothetical proof. Suppose that we've somehow magically figured out values of a and r to make the base cases work ($n = 1$ and $n = 2$—there are two base cases because f_2 is not defined recursively, either). And suppose that we're in the middle of a inductive proof:

$$f_n \overset{\text{definition of the recurrence}}{=} f_{n-1} + f_{n-2} \overset{\text{inductive hypothesis}}{\geq} ar^{n-1} + ar^{n-2} \overset{\text{algebra}}{=} ar^{n-2}(r+1).$$

But what we *want* to prove is $f_n \geq ar^n$. So we'd be done if only $r + 1 = r^2$—that is, if $r^2 - r - 1 = 0$. But the value of r isn't specified by the problem—so we get to choose its value! Using the quadratic formula, we find that there are two solutions to this equation, which we'll name ϕ and $\hat{\phi}$:

$$\phi = \tfrac{1+\sqrt{5}}{2} \qquad\qquad \hat{\phi} = \tfrac{1-\sqrt{5}}{2}.$$

Let's use $r = \phi$. To get the base cases to work, we would need to have $f_1 \geq a\phi$ and $f_2 \geq a\phi^2 = a(1+\phi)$—in other words, $1 \geq a\phi$ and $1 \geq a(1+\phi)$. Because $1 + \phi > \phi$, the latter is the harder one to achieve. To ensure that $a(1+\phi) \leq 1$, we must have

$$a \leq \tfrac{1}{1+\phi} = \tfrac{1}{1+\frac{1+\sqrt{5}}{2}} = \tfrac{2}{3+\sqrt{5}}.$$

We've now identified a value of r and a constraint on a so the proof might work. Let's try it!

Solution. The brainstorming above identifies a value $\phi = \frac{1+\sqrt{5}}{2}$ such that $\phi + 1 = \phi^2$ and a corresponding value of $a = \frac{2}{3+\sqrt{5}}$. Using these values, we'll prove the following claim:

Claim: $f_n \geq \frac{2}{3+\sqrt{5}} \cdot \phi^n$, where $\phi = \frac{1+\sqrt{5}}{2}$.

Proof (by strong induction on n). There are two base cases:

For $n = 1$, we have $\frac{2}{3+\sqrt{5}} \cdot \phi^1 = \frac{2}{3+\sqrt{5}} \cdot \frac{1+\sqrt{5}}{2} = \frac{1+\sqrt{5}}{3+\sqrt{5}} = 0.6180\cdots < 1 = f_1$.

For $n = 2$, we have $\frac{2}{3+\sqrt{5}} \cdot \phi^2 = \frac{2}{3+\sqrt{5}} \cdot (1+\phi)$ *we chose ϕ so that $\phi + 1 = \phi^2$*

$= \frac{2}{3+\sqrt{5}} \cdot \frac{3+\sqrt{5}}{2} = 1 = f_2$.

For the inductive case ($n \geq 3$), we assume the inductive hypothesis, namely that $f_k \geq \frac{2}{3+\sqrt{5}} \cdot \phi^k$ for any $1 \leq k \leq n-1$. Then:

$f_n = f_{n-1} + f_{n-2}$ *definition of the Fibonaccis*

$\geq \frac{2}{3+\sqrt{5}} \cdot \phi^{n-1} + \frac{2}{3+\sqrt{5}} \cdot \phi^{n-2}$ *inductive hypothesis, applied twice*

$$= \frac{2}{3+\sqrt{5}} \cdot \phi^{n-2} \cdot (\phi + 1) \qquad \textit{factoring}$$
$$= \frac{2}{3+\sqrt{5}} \cdot \phi^{n-2} \cdot \phi^2 \qquad \textit{we chose } \phi \textit{ so that } \phi + 1 = \phi^2$$
$$= \frac{2}{3+\sqrt{5}} \cdot \phi^{n}. \qquad \textit{algebra}$$

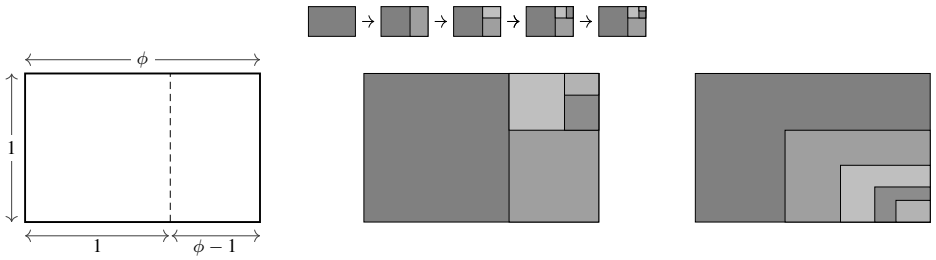
Therefore the claim follows by induction. □

Problem-solving tip: Sometimes starting in the middle of a proof helps! You still need to go back and connect the dots, but imagining that you've gotten somewhere may help you figure out how to get there.

Taking it further: The value $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803\cdots$ is called *the golden ratio.* It has a number of interesting characteristics, including both remarkable mathematical and aesthetic properties. For example, a rectangle whose side lengths are in the ratio ϕ-to-1 can be divided into a square and a rectangle whose side lengths are in the ratio 1-to-ϕ. That's because, for these rectangles to have the same ratios, we need $\frac{\phi}{1} = \frac{1}{\phi-1}$—that is, we need $\phi(\phi - 1) = 1$, which means $\phi^2 - \phi = 1$. (See Figure 6.27.) The golden ratio, it has been argued, describes proportions in famous works of art ranging from the Acropolis to Leonardo da Vinci's drawings. The Fibonacci numbers also show up all over the place in nature—and also in computation. One computational application in which they're relevant is in the design and analysis of a data structure called an *AVL tree,* a form of binary search tree that guarantees that the tree supports all its operations efficiently. See p. 6-53.

A closed-form formula for the Fibonaccis

While Example 6.27 establishes a lower bound on the Fibonacci numbers—in asymptotic notation, it proves that $f_n = \Omega(\phi^n)$—we have not yet established an exact formula for the nth Fibonacci number. Here is an exact formula, along with a proof that it's correct:



(a) A rectangle in the ratio of ϕ to 1, with an inscribed square.

(b) Repeatedly inscribing a square in the "leftover" rectangle.

(c) The same rectangles, rotated and shifted to share a corner.

Figure 6.27 Some golden rectangles.

Example 6.28: A closed-form solution for the Fibonaccis.

> *Some more brainstorming:* The trick will be to make use of $\hat{\phi}$. The inductive case would go through perfectly, just as in Example 6.27, if we tried to prove $f_n = a\phi^n + b\hat{\phi}^n$, for constants a and b. But what about the base cases? For f_1, we would need $1 = a\phi + b\hat{\phi}$; for f_2, we would need $1 = a\phi^2 + b(\hat{\phi}^2) = a(1+\phi) + b(1+\hat{\phi})$. That's two linear equations with two unknowns, and some algebra will reveal that $a = \frac{1}{\sqrt{5}}$ and $b = \frac{-1}{\sqrt{5}}$ solves these equations.

Claim: $f_n = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$, where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = \frac{1-\sqrt{5}}{2}$.

Proof (by strong induction on n). There are two base cases:

$$\begin{aligned}
\text{For } n = 1, \text{ we have } \frac{\phi^1 - \hat{\phi}^1}{\sqrt{5}} &= \frac{\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}}{\sqrt{5}} && \textit{definition of } \phi \textit{ and } \hat{\phi} \\
&= \frac{\frac{2\sqrt{5}}{2}}{\sqrt{5}} = 1 && \textit{algebra} \\
&= f_1. && \textit{definition of the Fibonaccis} \\
\text{For } n = 2, \text{ we have } \frac{\phi^2 - \hat{\phi}^2}{\sqrt{5}} &= \frac{1+\phi-(1+\hat{\phi})}{\sqrt{5}} && \phi^2 = 1+\phi \textit{ and } \hat{\phi}^2 = 1+\hat{\phi} \\
&= \frac{\phi - \hat{\phi}}{\sqrt{5}} = 1 && \textit{algebra and the previous case } (n=1) \\
&= f_2. && \textit{definition of the Fibonaccis}
\end{aligned}$$

For the inductive case ($n \geq 3$), we assume the inductive hypothesis: for any $k < n$, we have $f_k = \frac{\phi^k - \hat{\phi}^k}{\sqrt{5}}$. Then:

$$\begin{aligned}
f_n &= f_{n-1} + f_{n-2} && \textit{definition of the Fibonaccis} \\
&= \frac{\phi^{n-1} - \hat{\phi}^{n-1}}{\sqrt{5}} + \frac{\phi^{n-2} - \hat{\phi}^{n-2}}{\sqrt{5}} && \textit{inductive hypothesis, applied twice} \\
&= \frac{\phi^{n-2}(\phi+1) - \hat{\phi}^{n-2}(\hat{\phi}+1)}{\sqrt{5}} && \textit{factoring} \\
&= \frac{\phi^{n-2}\phi^2 - \hat{\phi}^{n-2}\hat{\phi}^2}{\sqrt{5}} && \phi + 1 = \phi^2 \textit{ and } \hat{\phi} + 1 = \hat{\phi}^2 \\
&= \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}. && \square
\end{aligned}$$

Computer Science Connections

AVL Trees

A *binary search tree* is a data structure that allows us to store a dynamic set of elements, supporting Insert, Delete, and Find operations. Briefly, a binary tree consists of a *root node* at the top; each node u can have zero, one, or two *children* directly attached beneath u. (See p. 11-73 for more about binary *search* trees, specifically.)

An *AVL tree* is a special type of binary search tree that ensures that the tree is "balanced" and therefore supports its operations very efficiently [3]. The point is to ensure that the tree is "shallow," because the cost of almost every operation on binary search trees is proportional to the height of the tree. (The *height* of a node in a tree is the number of levels of nodes beneath it; the height of the tree is the height of the root.) AVL trees were developed by and named after two Soviet computer scientists, Georgy Adelson-Velsky (1922–2014) and Evgenii Landis (1921–1997). (You'd expect it to be *three* people, but hyphenated names are confusing.)

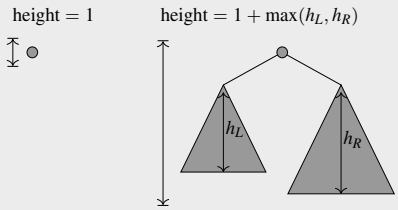


Figure 6.28 Binary trees and AVL trees. An AVL tree can be either any zero- or one-node tree (of height 0 and 1, respectively); or an AVL tree can be any tree whose root has one or two children in which (i) h_L and h_R differ by 0 or 1, and (ii) both subtrees are themselves AVL trees.

Specifically, an AVL tree is a binary search tree in which, for any node u, the height of u's left child and the height of u's right child differ by at most one. (See Figure 6.28.) Alternatively, we can define AVL trees recursively: an empty (zero-node) tree is an AVL tree of height 0; and a tree of height $h \geq 1$ is an AVL tree if both (i) the subtrees rooted at the two children of the root are both AVL trees; and (ii) the heights of the root's children are either both $h-1$, or one is $h-1$ and the other is $h-2$. A few examples of AVL trees are shown in Figure 6.29.

An upper bound

If you studied AVL trees before, you were probably told "AVL trees have logarithmic height." Here, we'll prove it. Consider an AVL tree T of height h. After a little contemplation, it should be clear that T will contain the maximum possible number of nodes (out of all AVL trees of height h) when both of the children of T's root node have height $h-1$, and furthermore that both subtrees of the root have as many nodes as an AVL tree of height $h-1$ can have. Let's think about this argument using a recurrence relation. Let $M(h)$ denote the maximum number of nodes that can appear in an AVL tree of height h. There can be only one node in a height 1 tree, so $M(1) = 1$. For $h \geq 2$, the above argument says that

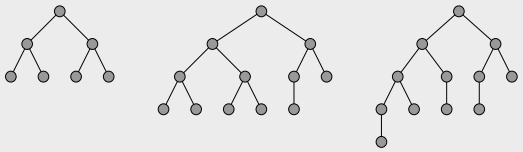


Figure 6.29 Three AVL trees. Take any node u in any of the three trees; you can check that the number of layers beneath u's left child and u's right child differ by at most one.

$$M(h) = \underbrace{M(h-1)}_{\text{the left subtree}} + \underbrace{M(h-1)}_{\text{the right subtree}} + \underbrace{1}_{\text{the root node}}. \qquad (*)$$

Claim: $M(h) = 2^h - 1$.

Proof. The proof is straightforward by induction. For the base case ($h = 1$), we have $M(h) = 1$ by definition, and $2^1 - 1 = 2 - 1 = 1$. For the inductive case, we have $M(h) = 2M(h) + 1 = 2 \cdot 2(2^{h-1} - 1) + 1$ by $(*)$ and the inductive hypothesis. Simplifying yields $M(h) = 2^h - 2 + 1 = 2^h - 1$. □

(Another way to see that the largest number of nodes in any binary tree of height h is $2^h - 1$ is by looking at the tree by row: there's 1 root, with 2 children, and 4 "grandchildren", and 8 "great-grandchildren, and so forth.) Figure 6.30a shows the fullest-possible AVL trees for a few small heights.

(continued)

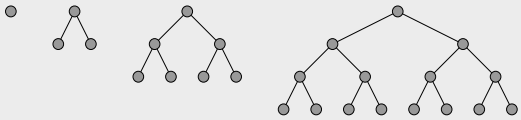
COMPUTER SCIENCE CONNECTIONS

AVL TREES, CONTINUED

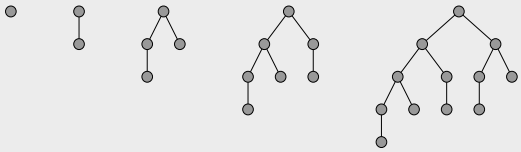
A lower bound

To analyze the worst-case height of an AVL tree, though, we need to look at the other direction: what is the *fewest* nodes that can appear in an AVL of height h? (We can transform this analysis into one that finds the largest possible height of an AVL tree with n nodes.)

Define $N(h)$ as the minimum number of nodes in an AVL tree of height h. As before, any height 1 tree has one node, so $N(1) = 1$. It's also immediate that $N(2) = 2$. (The emptiest-possible AVL trees of a few small heights are shown in Figure 6.30b. The smallest AVL tree of height 1 and height 2 are shown there, with 1 and 2 nodes, respectively.)



(a) The fullest-possible AVL trees of height $h \in \{1, 2, 3, 4\}$, respectively containing $1 = 2^1 - 1$, $3 = 2^2 - 1$, $7 = 2^3 - 1$, and $15 = 2^4 - 1$ nodes.



(b) The emptiest-possible AVL trees of height $h \in \{1, 2, 3, 4, 5\}$, which contain 1, 2, 4, 7, and 12 nodes.

Figure 6.30 The fullest and emptiest possible AVL trees for a few small heights.

For a larger height h, it's not too hard to persuade yourself that the minimum number of nodes in an AVL tree of height h is achieved when the root has one child of height $h-1$ and one child of height $h-2$—and furthermore when these two subtrees contain as few nodes as legally possible. That is,

$$N(h) = \underbrace{N(h-1)}_{\text{the deeper subtree}} + \underbrace{N(h-2)}_{\text{the shallower subtree}} + \underbrace{1}_{\text{the root node}}. \qquad (\dagger)$$

Observe that $N(h) = 1 + N(h-1) + N(h-2) \geq 1 + 2 \cdot N(h-2)$ because $N(h-1) \geq N(h-2)$. Therefore we can conclude that $N(h) \geq 2^{h/2} - 1$.

We can do better, though, with a bit more work. Define $P(h) = 1 + N(h)$. Adding one to both sides of (†), in this new notation, we have that $P(h) = P(h-1) + P(h-2)$. (This recurrence should look familiar: it's the same recurrence as for the Fibonacci numbers!) Because $P(1) = 1 + N(1) = 2 = f_3$ and $P(2) = 1 + N(2) = 3 = f_4$, we can prove inductively that $P(h) = f_{h+2}$.

Claim: $N(h) \geq \phi^h - 1$.

Proof. Using the definition of P, the proof in Example 6.27, and the fact that $\frac{1}{\phi^2} = \frac{2}{3+\sqrt{5}}$, we have

$$N(h) = P(h) - 1 = f_{h+2} - 1 \geq \tfrac{2}{3+\sqrt{5}} \cdot \phi^{h+2} - 1 = \phi^h - 1. \qquad \square$$

Putting it all together

The analysis above will let us prove the following theorem, where $\phi = \frac{1+\sqrt{5}}{2}$:

Theorem 6.20: The height of an AVL tree is logarithmic.
The height h of any n-node AVL tree satisfies $\log_\phi(n+1) \geq h \geq \log_2(n+1)$.

(We know from the first claim that $2^h - 1 = M(h) \geq n$. Thus $2^h \geq n+1$, and—taking logs of both sides—we have $h \geq \log_2(n+1)$. Similarly, we know from the second claim that $\phi^h - 1 = N(h) \leq n$. Thus $\phi^h \leq n+1$, and—taking $\log_\phi$ of both sides—we have $h \leq \log_\phi(n+1)$.)

By changing log bases, we have that $\log_\phi(x) = \frac{\log_2(x)}{\log_2(\phi)} \approx \frac{\log_2(x)}{0.69424\cdots} \approx 1.4404 \cdot \log_2(x)$. Thus this theorem says that an n-node AVL tree has height between $\log_2(n+1)$ and $1.44\log_2(n+1)$. (In fact, there are AVL trees whose height is as large as $1.44\log_2(n+1)$, so this analysis is tight.)

EXERCISES

A quadtree *is a data structure typically used to store a collection of n points in* $\mathbb{R}^2$*. The basic idea is to start with a bounding box that includes all n points, and then subdivide, into four equal-sized subregions, any region that contains more than a designated number k of points. (For simplicity, we will subdivide any region with more than* $k = 1$ *point.) The* height *of a quadtree is the number of levels of the deepest subdivision of the tree. See Figure 6.31.*

6.80 Let $R(h)$ denote the *largest number of regions* that a quadtree of height h can contain. Write a recurrence relation for $R(h)$.

6.81 Let $S(h)$ denote the *smallest number of regions* that a quadtree of height h can contain. Write a recurrence relation for $S(h)$.

6.82 Let $T(n)$ denote the *smallest number of regions* that a quadtree with n points can contain. Write a recurrence relation for $T(n)$. You may use the following fact without proof: the most efficient division of n points in a quadtree occurs when each subregion contains precisely $\frac{n}{4}$ points.

Consider the recursive algorithms shown in Figure 6.32, which all solve the same problem. The following exercises ask you to write down a recurrence relation to express each running time, and then use your recurrence to prove by induction that the algorithm requires $O(n)$ *time. (Assume that selecting a subarray takes* $\Theta(1)$ *time.)*

6.83 Give a recurrence for **B**.

6.84 Give a recurrence for **C**.

6.85 Give a recurrence for **D**.

6.86 Prove that **B** is $O(n)$.

6.87 Prove that **C** is $O(n)$. (For ease, you may assume n is a power of 2.)

6.88 Prove that **D** is $O(n)$.

6.89 What problem do **B**, **C**, and **D** solve?

Consider the following ternary search *algorithm, a variation on binary search. Suppose you have a sorted array* $A[1 \ldots n]$ *and you're searching for a particular value x in it. If* $n \leq 2$*, just check whether x is one of the one or two entries in A. Otherwise, compare x to* $A[n/3]$ *and* $A[2n/3]$*, and do the following:*

- *if* $x = A[\lfloor n/3 \rfloor]$ *or* $x = A[\lfloor 2n/3 \rfloor]$*, return true.*
- *if* $x < A[\lfloor n/3 \rfloor]$*, recursively search* $A[1 \ldots \lfloor n/3 \rfloor - 1]$.
- *if* $A[\lfloor n/3 \rfloor] < x < A[\lfloor 2n/3 \rfloor]$*, recursively search* $A[\lfloor n/3 \rfloor + 1 \ldots \lfloor 2n/3 \rfloor - 1]$.
- *if* $x > A[\lfloor 2n/3 \rfloor]$*, recursively search* $A[\lfloor 2n/3 \rfloor + 1 \ldots n]$.

6.90 Analyze the asymptotic worst-case running time of ternary search. Prove your answer correct using induction. For convenience, you may assume that n is a power of three.

6.91 Does ternary search perform better or worse than binary search? Here you should count the *exact* number of comparisons that each algorithm performs—don't give an asymptotic answer.

6.92 Consider a simplified (and thus slightly erroneous) version of the recurrence for Binary Search: $T(n) = T(\frac{n}{2}) + c$ and $T(1) = c$. (This recurrence ignores the off-by-one complications.) Prove by induction that $T(n) = c(1 + \log n)$ if n is a power of two.

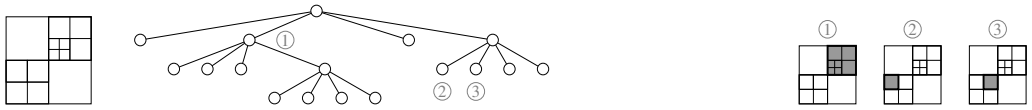


Figure 6.31 The decomposition of the plane to build a quadtree. (A region's children are its subregions, clockwise from the upper left.) The shaded regions at right correspond to the numbered nodes of the quadtree. This quadtree contains 17 regions, and its height is 4.

```
B(A[1...n]):
1 if n = 0 then
2     return 0
3 else if A[1] < 0 then
4     return 1 + B(A[2...n])
5 else
6     return B(A[2...n])
```

```
C(A[1...n]):
1 if n = 0 or (n = 1 and A[1] ≥ 0) then
2     return 0
3 else if n = 1 and A[1] < 0 then
4     return 1
5 else
6     count := 0
7     count := count + C(A[1...⌊n/2⌋])
8     count := count + C(A[⌊n/2⌋ + 1...n])
9     return count
```

```
D(A[1...n]):
1 if n = 0 or (n = 1 and A[1] ≥ 0) then
2     return 0
3 else if n = 1 and A[1] < 0 then
4     return 1
5 else
6     count := 0
7     count := count + D(A[1...⌊n/4⌋])
8     count := count + D(A[⌊n/4⌋ + 1...⌊3n/4⌋])
9     count := count + D( A[⌊3n/4⌋ + 1...n] )
10    return count
```

Figure 6.32 Three recursive algorithms.

6.93 Consider the "Median of Three" pivoting rule for **quickSort**. (See Example 5.14 and Exercises 6.74–6.77.) As Exercise 6.77 established, this algorithm can still be slow; the recurrence relation for the worst-case version of the algorithm is $T(1) = T(2) = 1$ and $T(n) = T(n-2) + cn$. Prove that $T(n) = \Theta(n^2)$.

6.94 Generalize your argument from Exercise 6.93 to show that the recurrence

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

has solution $T(n) = \Theta(n^2)$ *for any integer* $k \ge 1$.

Recall that the Fibonacci numbers are defined by the recurrence $f_1 = f_2 = 1$ *and* $f_n = f_{n-1} + f_{n-2}$. *The next several exercises refer to this recurrence and the algorithms for computing the Fibonacci numbers in Figure 6.33.*

6.95 First, a warmup unrelated to the algorithms in Figure 6.33: prove by induction that $f_n \le 2^n$.

6.96 Prove that **fibNaive**$(n-k)$ appears a total of f_{k+1} times in the call tree for **fibNaive**(n).

6.97 Write down and solve a recurrence for the running time of **helper** (and therefore **fibMedium**).

6.98 Write down and solve a recurrence for the running time of **exp** (and therefore **fibClever**).

6.99 *(programming required.)* The reference to "repeated squaring" in **fibMatrix** is precisely the same as the idea of **exp**. Implement **fibMatrix** using this idea in a programming language of your choice. (See Exercise 5.62.)

```
fibNaive(n):
1 if n = 0 or n = 1 then
2     return 1
3 else
4     return fibNaive(n−1)+fibNaive(n−2)
```

```
fibMatrix(n):
1 Compute (using repeated squaring)
      [x]   [1 1]^n   [1]
      [y] := [1 0]   · [1] .
  // See Exercises 5.62 and 6.99.
2 return x
```

```
fibMedium(n):
1 ⟨f_n, f_{n−1}⟩ := helper(n)
2 return f_n

helper(n):
1 if n = 0 then
2     return ⟨1, undefined⟩
3 else if n = 1 then
4     return ⟨1, 1⟩
5 else
6     ⟨f_{n−1}, f_{n−2}⟩ := helper(n − 1)
7     return ⟨f_{n−1} + f_{n−2}, f_{n−1}⟩
```

```
fibClever(n):
1 return (exp(φ,n) − exp(φ̂,n)) / √5

exp(b, n):
1 if n = 0 then
2     return 1
3 else
4     s := exp(b, ⌊n/2⌋)
5     if n is odd then
6         return b · s · s
7     else
8         return s · s
```

Figure 6.33 Four algorithms for the Fibonaccis. The values ϕ and $\hat{\phi}$ satisfy $f_n = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$; see Example 6.28.

The Merge Sort algorithm sorts an input array by splitting it in half, recursively sorting the two halves, and then merging the two resulting sorted arrays into a single sorted array. (See Figure 6.23 for **mergeSort***, and see Figure 6.34 for a reminder of the algorithm that does this merging.)*

6.100 Give a recurrence relation $T(n)$ describing the running time of **merge** on two input arrays with a total of n elements, and prove that $T(n) = \Theta(n)$.

6.101 Consider the recurrence for the running time of **mergeSort** (see Figure 6.23):

$$T(1) = c \qquad \text{and} \qquad T(n) = T(\lceil \tfrac{n}{2} \rceil) + T(\lfloor \tfrac{n}{2} \rfloor) + cn.$$

Prove that $T(n) \leq T(n')$ if $n \leq n'$—that is, T is monotonic.

6.102 Here is a recurrence relation for the number of *comparisons* done by **mergeSort** on an input array of size n:

$$C(1) = 0 \qquad \text{and} \qquad C(n) = 2C(\tfrac{n}{2}) + n - 1.$$

Explain the recurrence relation, and then prove that $C(n) = n \log n - n + 1$ by induction. (For ease, we'll assume that n is a power of two.)

The next few exercises refer to the algorithms **f** *and* **g** *in Figure 6.35, both which solve the same problem.*

6.103 Give and solve (using induction) a recurrence relation for the running time of **f**.

6.104 Give a recurrence relation for **g**, and use it to prove that $\mathbf{g}(n)$ runs in $O(\log^2 n)$ time.

6.105 Describe the set of input values n that cause the worst-case behavior for $\mathbf{g}(n)$.

6.106 What problem do **f** and **g** solve? Prove your answer.

(A true story, inspired by Michael Eisen's 2011 blog post "Amazon's $23,698,655.93 book about flies" [44].) Two copies of an out-of-print book were listed online by Seller A and Seller B. Their prices were over $1,000,000 each—and the next day, both prices were over $2,000,000, and they kept going up. By watching the prices over several days, it became clear that the two sellers were using algorithms to set their prices in response to each other. Let a_n and b_n be the prices offered on day n by Seller A and Seller B, respectively. The prices were set by two (badly conceived) algorithms such that $a_n = \alpha \cdot b_{n-1}$ and $b_n = \beta \cdot a_n$ where $\alpha = 0.9983$ and $\beta = 1.27059$.

6.107 Suppose that $b_0 = 1$. Find closed-form formulas for a_n and b_n. Prove your answer.

6.108 State a necessary and sufficient condition on α, β, and b_0 such that $a_n = \Theta(1)$ and $b_n = \Theta(1)$.

```
merge(X[1...n], Y[1...m]):
1 if n = 0 then
2     return Y
3 else if m = 0 then
4     return X
5 else if X[1] < Y[1] then
6     return X[1] followed by merge(X[2...n], Y)
7 else
8     return Y[1] followed by merge(X, Y[2...m])
```

Figure 6.34 The "merging" of two sorted arrays.

```
f(n):
1 if n ≤ 1 then
2     return n
3 else
4     return f(n − 2)
```

```
g(n):
1 if n ≤ 1 then
2     return n
3 else
4     x := 1
5     while n ≥ 2x:
6         x := 2 · x
7     return g(n − x)
```

Figure 6.35 Two recursive algorithms.

6.5 An Extension: Recurrence Relations of the Form $T(n) = aT\left(\frac{n}{b}\right) + cn^k$

> It is wise to do that, for life is but short and time passes quickly. If one is competent in one thing and understands one thing well, one gains at the same time insight into and knowledge of many other things into the bargain.
>
> Vincent van Gogh (1853–1890)
> letter to Theo van Gogh (3 April 1878)

In this section, we'll develop a formulaic method to solve a common type of recurrence relation that comes up often: in analyzing recursive algorithms, we will frequently encounter recurrences that look like $T(n) = aT\left(\frac{n}{b}\right) + c \cdot n^k$, for four constants $a \geq 1$, $b > 1$, $c > 0$, and $k \geq 0$. Here, we'll develop a unified solution to this type of recurrence relation. Why do these recurrences come up frequently? They arise in any recursive algorithm that has the following structure: if the input is small, then we compute the solution directly; otherwise, to solve an instance of size n:

- we make a different recursive calls on inputs of size $\frac{n}{b}$; and
- to construct the smaller instances and then to reconstruct the solution to the given instance from the recursive solutions, we spend $\Theta(n^k)$ time.

These algorithms are usually called *divide-and-conquer algorithms:* they "divide" their input into a pieces, and then recursively "conquer" those subproblems. (To be precise, the recurrence often has ceilings and floors as part of its recursive calls, but for now assume that n is exact power of b, so that the floors and ceilings don't matter.) Here are two examples of recursive algorithms with recurrences of this form:

Example 6.29: Binary Search.
In Binary Search, we spend $c = \Theta(1)$ time to compare the sought element to the middle of the range; we then make one recursive call to search for the element in the appropriate half of the array. If n is an exact power of two, then the recurrence is

$$T(n) = T(\tfrac{n}{2}) + c.$$

(So $a = 1$, $b = 2$, and $k = 0$, because $c = c \cdot 1 = c \cdot n^0$.)

Example 6.30: Merge Sort.
In Merge Sort, we spend $\Theta(1)$ time to divide the array in half. We make two recursive calls on the left and right subarrays, and then spend $\Theta(n)$ time to merge the resulting sorted subarrays into a single sorted array. If n is an exact power of two, then the recurrence is

$$T(n) = 2T(\tfrac{n}{2}) + c \cdot n.$$

(So $a = 2$, $b = 2$, and $k = 1$.)

6.5.1 Solving Recurrences of the Form $T(n) = aT\left(\frac{n}{b}\right) + cn^k$: Some Intuition

We are going to develop a general technique that allows us to solve any recurrence relation of the form $T(n) = aT(\frac{n}{b}) + c \cdot n^k$. The technique is based on examining the recursion tree for this recurrence (see Figure 6.36), and a theorem (Theorem 6.21) that describes the total amount of work represented by this tree. Here's the intuition. Let's think about the ith level of the recursion tree (again, see Figure 6.36)—in other words, the work done by the recursive calls that are i levels beneath the root of the recursion tree. Here are a few useful facts:

There are a^i different calls at level i. There is $1 = a^0$ call at the zeroth (root) level, then $a = a^1$ calls at first level, then a^2 calls at the second level, and so forth.

Each of the the calls at the ith level operates on an input of size $\frac{n}{b^i}$. The input size is $\frac{n}{1} = n$ at the zeroth level, then $\frac{n}{b}$ at the first level, then $\frac{n}{b^2}$ at the second, and so forth.

Thus the total amount of work in the ith level of the tree is $a^i \cdot c \cdot (\frac{n}{b^i})^k$. (That's just the number of calls at the ith level multiplied by the work per call.) Or, simplifying, the total work at the ith level is $cn^k \cdot (\frac{a}{b^k})^i$.

Thus the total amount of work contained within the entire tree is

$$\sum_i cn^k \cdot \left(\frac{a}{b^k}\right)^i = cn^k \cdot \sum_i \left(\frac{a}{b^k}\right)^i. \qquad (*)$$

(We'll worry about the bounds on the summation later.)

Note that $(*)$ expresses the total work in the recursion tree as a geometric sum $\sum_i r^i$, in which the ratio between terms is given by $r = \frac{a}{b^k}$. (See Section 5.2.2.) As with any geometric sum, the critical question

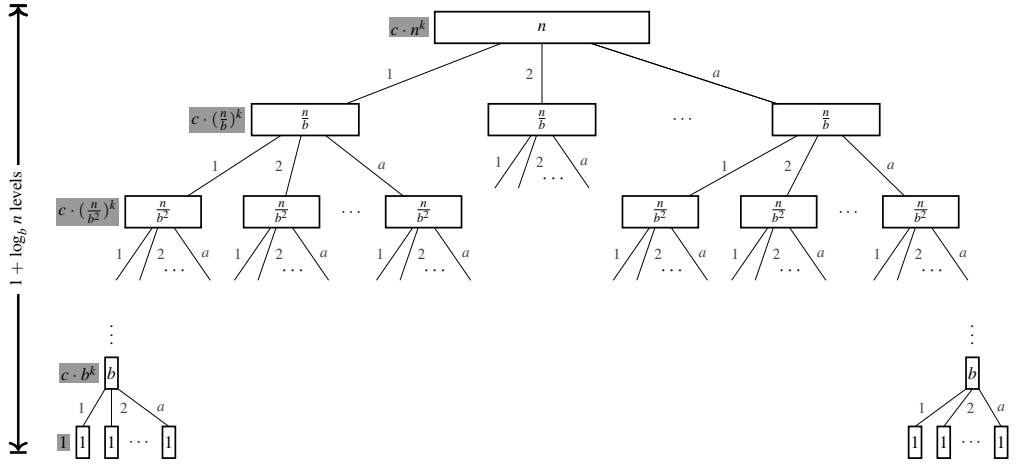


Figure 6.36 The recursion tree for a recurrence relation $T(n) = aT(\frac{n}{b}) + c \cdot n^k$. Assume that n is an exact power of b.

is how the ratio compares to 1: if $r < 1$, then the terms of the sum are getting smaller and smaller as i increases; if $r > 1$, then the terms of the sum are getting bigger and bigger as i increases. (And if $r = 1$, then each term is simply equal to 1.)

Our unifying theorem will have three cases, each of which corresponds to one of these three natural cases for the summation in $(*)$: its terms *increase exponentially* with i, its terms *decrease exponentially* with i, or its terms are *constant* with respect to i. In these cases, respectively, most of the work is done at the leaves of the tree; most of the work is done at the root of the tree; or the work is spread evenly across the levels of the tree.

A trio of examples

Before we prove the general theorem, we'll solve a few recurrences that illustrate these three cases, and then we'll prove the result in general. The three example recurrences are

$$T(n) = 2T(\tfrac{n}{2}) + 1 \qquad \text{and} \qquad T(n) = 2T(\tfrac{n}{2}) + n \qquad \text{and} \qquad T(n) = 2T(\tfrac{n}{2}) + n^2,$$

all with $T(1) = 1$. Figure 6.37 simultaneously sketches the recursion trees for these recurrences.

In each of these recurrences, we divide the input by two at every level of the recursion. Thus, the total depth of the recursion tree is $\log_2 n$. (Assume that n is an exact power of two.) In the recursion tree for any one of these recurrences, consider the ith level of the tree beneath the root. (The root of the recursion tree has depth 0.) We have divided n by 2 a total of i times, and thus the input size at that level is $\frac{n}{2^i}$. Furthermore, there are 2^i different calls at the ith level of the tree.

Solving the three recurrences

To solve each recurrence, we will sum the total amount of work generated at each level of the tree. The three recursion trees for these three recurrences are shown in Figure 6.38.

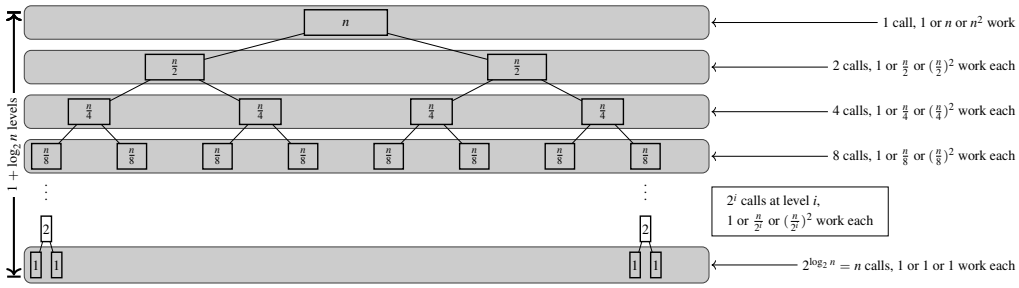
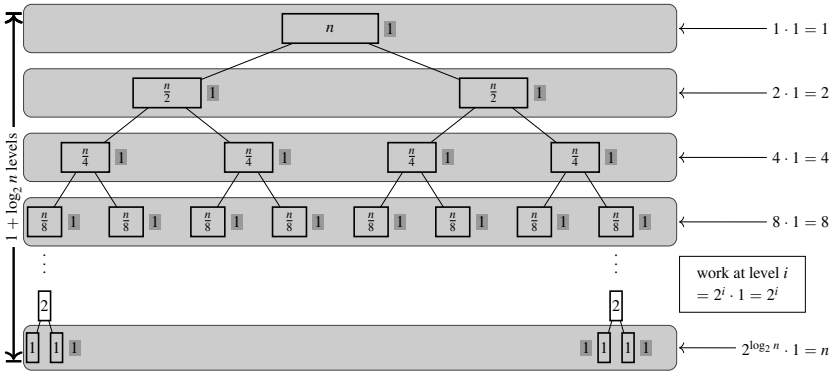
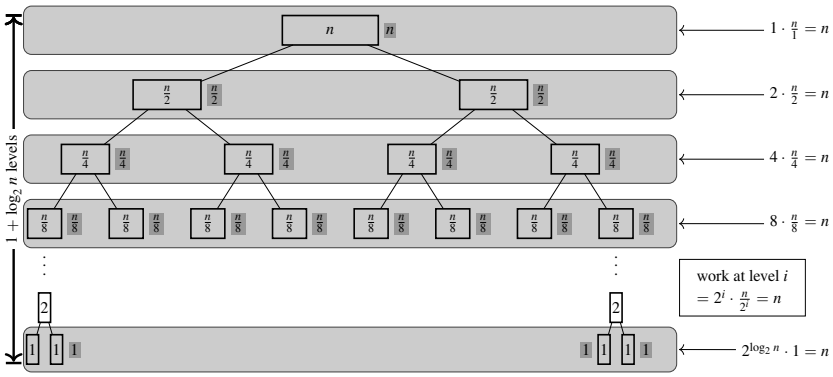


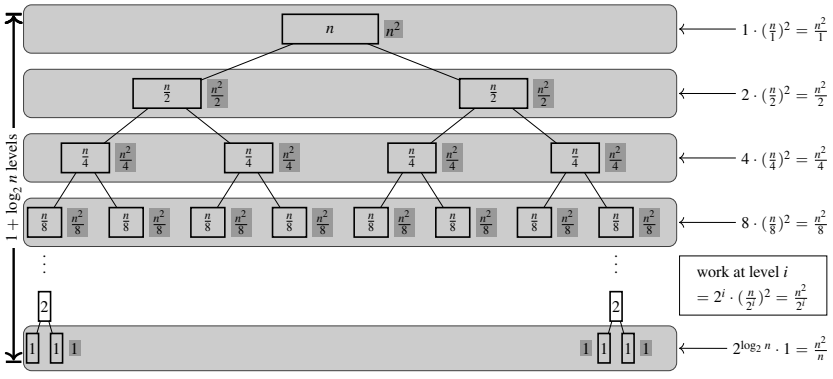
Figure 6.37 The recursion trees for three recurrences: $T(n) = 2T(\frac{n}{2}) + f(n)$, for $f(n) \in \{1, n, n^2\}$. Each row of the tree is annotated with both the number of calls at that level, plus the additional work done by each call at that level.



(a) The recursion tree for $T(n) = 2T(\frac{n}{2}) + 1$, with the "row-wise" sums of work. The work at each level is twice the work at the level above it; thus the work is increasing exponentially at each level of the tree.



(b) The recursion tree for $T(n) = 2T(\frac{n}{2}) + n$. The work at each level is exactly n; thus the work is constant across the levels of the tree.



(c) The recursion tree for $T(n) = 2T(\frac{n}{2}) + n^2$. The work at each level is half of the work at the level above it; thus the work is decreasing exponentially at each level of the tree.

Figure 6.38 Three different recursion trees.

Example 6.31: Solving $T(n) = 2T(\frac{n}{2}) + 1$.

Figure 6.38a shows the recursion tree for this recurrence. There are 2^i different calls at the ith level, each of which is on an input of size $\frac{n}{2^i}$—and we do 1 unit of work for each of these 2^i calls. Thus the total amount of work at level i is 2^i. The total amount of work in the entire tree is therefore

$$T(n) = \sum_{i=0}^{\log_2 n} 2^i = \frac{2^{1+\log_2 n} - 1}{2 - 1} = 2 \cdot 2^{\log_2 n} = 2n$$

by Theorem 5.5 (on geometric series). And, indeed, $T(n) = \Theta(n)$.

Example 6.32: Solving $T(n) = 2T(\frac{n}{2}) + n$.

Figure 6.38b shows the recursion tree. There are 2^i calls at the ith level of the recursion tree, on inputs of size $\frac{n}{2^i}$. We do $\frac{n}{2^i}$ units of work at each call, so the total work at the ith level is $2^i \cdot (\frac{n}{2^i}) = n$. Note that the amount of work at level i is independent of the level i. The total amount of work in the tree is therefore

$$T(n) = \sum_{i=0}^{\log_2 n} n = n \cdot \sum_{i=0}^{\log_2 n} 1 = n(1 + \log_2 n) = \Theta(n \log n).$$

(the work at level i is exactly n, for every i)

Example 6.33: Solving $T(n) = 2T(\frac{n}{2}) + n^2$.

Figure 6.38c shows the recursion tree. There are 2^i calls at the ith level of the tree, and we do $(\frac{n}{2^i})^2$ work at each call at this level. Thus the work represented by the ith row of the recursion tree is $(\frac{n}{2^i})^2 \cdot 2^i = \frac{n^2}{2^i}$. The total amount of work in the tree is therefore

$$T(n) = \sum_{i=0}^{\log_2 n} (\tfrac{1}{2})^i n^2 = n^2 \cdot \sum_{i=0}^{\log_2 n} (\tfrac{1}{2})^i = n^2 \cdot (2 - \tfrac{1}{n}),$$

($1 + \frac{1}{2} + \cdots + \frac{1}{2^{\log_2 n}} = 2 - \frac{1}{2^{\log_2 n}} = 2 - \frac{1}{n}$)

by Theorem 5.5 (on geometric series). Because $2 - \frac{1}{n}$ is certainly at most 2, and also certainly at least 1, we know that $n^2 \leq T(n) \leq 2n^2$, which allows us to conclude that $T(n) = \Theta(n^2)$.

6.5.2 The Formal Statement and Some Examples

Examples 6.31–6.33 were designed to build the necessary intuition about the three different cases of the recursion tree: work increases exponentially across levels of the tree; work stays constant across levels; or work decreases exponentially across levels. Precisely the same intuition will yield the proof of the unifying theorem for the collection of recurrences we're considering. Here is the formal statement of the theorem, which generalizes the idea of these examples to all recurrences of the form $T(n) = aT(\frac{n}{b}) + cn^k$:

Theorem 6.21: A formulaic solution to recurrences of the form $T(n) = aT(\frac{n}{b}) + cn^k$.
Consider the recurrence

$$T(1) = c \qquad \text{and} \qquad T(n) = a \cdot T(\tfrac{n}{b}) + c \cdot n^k$$

for constants $a \geq 1$, $b > 1$, $c > 0$, and $k \geq 0$. Then:

Case (i) ["the leaves dominate"]: if $b^k < a$, then $T(n) = \Theta(n^{\log_b(a)})$.
Case (ii) ["all levels are equal"]: if $b^k = a$, then $T(n) = \Theta(n^k \cdot \log n)$.
Case (iii) ["the root dominates"]: if $b^k > a$, then $T(n) = \Theta(n^k)$.

(As we discussed previously, we are abusing notation by using c to denote two different constants in this theorem statement. Again, as you'll prove in Exercise 6.126, the recurrence $T(1) = d$ with a constant $d > 0$ possibly different than c has precisely the same asymptotic solution.)

The proof of the theorem is in Section 6.5.3. Here, we'll just use the theorem with a few examples, using Theorem 6.21 to reproduce the recursion-tree analysis of Examples 6.31–6.33:

Example 6.34: Solving $T(n) = 2T(\frac{n}{2}) + 1$ using Theorem 6.21.
Consider the recurrence $T(n) = 2T(\frac{n}{2}) + 1$ with $T(1) = 1$. We have $a = 2$, $b = 2$, $c = 1$, and $k = 0$; because $b^k = 2^0 = 1 < 2 = a$, case (i) of Theorem 6.21 says that $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$.

Example 6.35: Solving $T(n) = 2T(\frac{n}{2}) + n$ using Theorem 6.21.
Consider the recurrence $T(n) = 2T(\frac{n}{2}) + n$ with $T(1) = 1$. We have $a = 2$, $b = 2$, $c = 1$, and $k = 1$; because $b^k = 2^1 = 2 = a$, case (ii) of Theorem 6.21 says that $T(n) = \Theta(n^1 \log n) = \Theta(n \log n)$.

Example 6.36: Solving $T(n) = 2T(\frac{n}{2}) + n^2$ using Theorem 6.21.
Consider the recurrence $T(n) = 2T(\frac{n}{2}) + n^2$ with $T(1) = 1$. We have $a = 2$, $b = 2$, $c = 1$, and $k = 2$; because $b^k = 2^2 = 4 > 2 = a$, case (iii) of Theorem 6.21 says that $T(n) = \Theta(n^2)$.

Taking it further: Although we've mostly presented "algorithmic design" and "algorithmic analysis" as two separate phases, in fact there's interplay between these pieces. See p. 6-68 for a discussion of a particular computational problem—matrix multiplication—and algorithms for it, including a straightforward but slow algorithm and another that (with inspiration from Theorem 6.21) improves upon that slow algorithm.

6.5.3 A Proof of Theorem 6.21

(This section goes into a bit more detail of a more technical proof than most of those in the book. If you prefer to use the theorem without delving into the proof, you can skip this section. But if you're interested, the proof is here.)

Taking it further: While Theorem 6.21 holds even when the input n is not an exact power of b—we just have to fix the recurrence by adding floors or ceilings so that it still makes sense—we will prove the result for exact powers of b only. A full proof that includes for the case when n is not an exact power of b can be found in [33], or you can read more about the original formulation and motivation for this solution technique in the 1980 paper by Jon Bentley, Dorothea Haken, and James Saxe that presented the method [13]. The unified technique that we're considering in this section is frequently called the "Master Method" in papers and textbooks, but I've avoided that terminology here—in part because I think it's a little misleading (there *are* many recurrence relations that have the form addressed by Theorem 6.21, but there are also many other recurrence relations that do not, so it's not quite as universal as a "master key"), and in part to avoid the word "master." There's a recent (and sadly belated) push in CS to think about and avoid some racially tinged terminology that has been commonly used—things like "blacklist/whitelist" in networking and "master/slave" in computer architecture—and, while the use of "master" in "Master Method" is something much more benign, it remains valuable and kind to think carefully about the impact of one's choice of language. For more, see the Inclusive Naming Initiative's homepage `inclusivenaming.org`.

Remember that we're considering the recurrence $T(n) = a \cdot T(\frac{n}{b}) + c \cdot n^k$ with $T(1) = c$. We will show that the total amount work contained in the recursion tree is

$$T(n) = cn^k \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i. \qquad (*)$$

As before, the formula $(*)$ should make intuitive the fact that $a = b^k$ (that is, $\frac{a}{b^k} = 1$) is the critical value for the theorem. The value of $\frac{a}{b^k}$ corresponds to whether the work at each level of the tree is increasing ($\frac{a}{b^k} > 1$), steady ($\frac{a}{b^k} = 1$), or decreasing ($\frac{a}{b^k} < 1$). The summation in $(*)$ is a geometric sum, and as we saw in Chapter 5 geometric sums behave fundamentally differently based on whether their ratio is less than, equal to, or greater than one.

Proof of Theorem 6.21 (for n an exact power of b). For all three cases, we'll use $(*)$ as the starting point, so we need to prove that statement first. A formal proof by induction is left to you as Exercise 6.127, but here is the intuition from the recursion tree (Figure 6.36). There are a^i nodes at the ith level of the tree (counting the root as level zero). Each node at the ith level corresponds to an input of size $\frac{n}{b^i}$ and therefore contributes $c \cdot (\frac{n}{b^i})^k$ work. The tree continues until the inputs are of size 1—that is, until $\frac{n}{b^i} = 1$, or when $i = \log_b n$. Thus the work at level i is $c \cdot (\frac{n}{b^i})^k \cdot a^i$, and the total work is the level-by-level sum, from level $i = 0$ (the root) down to level $i = \log_b n$ (the leaves). This summation is exactly the right-hand side of $(*)$.

We'll now examine this summation in each of the three cases, depending on how the value of $\frac{a}{b^k}$ compares to 1—and we'll handle the cases in order of ease, rather than numerically:

Case (ii): $a = b^k$. Then $(*)$ says that

$$T(n) = cn^k \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i = cn^k \sum_{i=0}^{\log_b n} 1 = cn^k(1 + \log_b n).$$

(in case (ii), we have $a = b^k$, which means $(\frac{a}{b^k})^i = 1^i = 1$ for all i)

Thus the total work is $\Theta(n^k \log n)$.

Case (iii): $a < b^k$. Then $(*)$ is a geometric sum whose ratio is strictly less than 1. Corollary 5.6 states that any geometric sum whose ratio is strictly between 0 and 1 is $\Theta(1)$. (Namely, the summation $\sum_{i=0}^{\log_b n} (\frac{a}{b^k})^i$

is lower-bounded by 1 and upper-bounded by $\frac{1}{1-a/b^k}$, both of which are positive constants when $a < b^k$.) Therefore:

$$T(n) = cn^k \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i = cn^k \cdot \Theta(1).$$

(by Corollary 5.6, because $\frac{a}{b^k} < 1$ in case (iii))

Therefore the total work is $\Theta(n^k)$.

Case (i): $a > b^k$. Then $(*)$ is a geometric sum whose ratio is strictly larger than one. But we can make this summation look more like Case (iii), by using a little algebraic manipulation. Notice that, for any $r \neq 0$, we can rewrite $\sum_{i=0}^{m} r^i$ as follows:

$$\sum_{i=0}^{m} r^i = \sum_{i=0}^{m} r^m \cdot r^{i-m} = r^m \cdot \sum_{i=0}^{m} r^{i-m} = r^m \cdot \sum_{i=0}^{m} \left(\frac{1}{r}\right)^{m-i} = r^m \cdot \sum_{j=0}^{m} \left(\frac{1}{r}\right)^j. \qquad (\dagger)$$

(reindexing the summation, by setting $j = m - i$)

Applying this manipulation to $(*)$, we have

$$\begin{aligned}
T(n) &= cn^k \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i && \text{by } (*) \\
&= cn^k \cdot \left(\frac{a}{b^k}\right)^{\log_b n} \cdot \sum_{j=0}^{\log_b n} \left(\frac{b^k}{a}\right)^j && \text{by } (\dagger) \\
&= cn^k \cdot \left(\frac{a}{b^k}\right)^{\log_b n} \cdot \Theta(1) && \text{by Corollary 5.6 (similar to Case (iii)), because } \tfrac{b^k}{a} < 1. \\
&= cn^k \cdot \frac{a^{\log_b n}}{n^k} \cdot \Theta(1) && (b^k)^{\log_b n} = b^{k \log_b n} = b^{\log_b n^k} = n^k \\
&= c \cdot a^{\log_b n} \cdot \Theta(1).
\end{aligned}$$

And $a^{\log_b n} = n^{\log_b a}$, which we can verify by log manipulations:

$$a^{\log_b n} = b^{\log_b [a^{\log_b n}]} = b^{[\log_b n] \cdot [\log_b a]} = b^{[\log_b a] \cdot [\log_b n]} = b^{\log_b [n^{\log_b a}]} = n^{\log_b a}.$$

Therefore the total work in this case is $\Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$. □

Computer Science Connections

Divide-and-Conquer Algorithms and Matrix Multiplication

Matrix multiplication (see Definition 2.45) is a fundamental operation with wide-ranging applications throughout CS: in computer graphics, in data mining, and in social-network analysis, just to name a few. Often the matrices in question are quite large—perhaps a matrix of hyperlinks among thousands or millions of web pages, for example. Thus asymptotic improvements to matrix multiplication algorithms have potential practical importance, too.

For simplicity, consider multiplying two square (n-by-n) matrices. The most obvious algorithm for matrix multiplication just follows the definition: separately, for each of the n^2 entries in the output matrix, perform the $\Theta(n)$ multiplications and additions to compute the entry. (See Figure 6.39.) But, in the spirit of this section, what might we be able to do with a recursive algorithm? There is indeed a nice way to think about matrix multiplication recursively. To multiply two n-by-n matrices M and N, divide M and N each into four quarters. The matrix product MN can be expressed by appropriately summing the products of the quarters of M and the quarters of N. This fact suggests a recursive, divide-and-conquer algorithm for multiplying matrices, with the recurrence $T(n) = 8T(\frac{n}{2}) + n^2$. (It takes $c \cdot n^2$ time to combine the result of the recursive calls.) See Figure 6.40.

matmult$(M \in \mathbb{R}^{n\times n}, N \in \mathbb{R}^{n\times n})$:

Input: Two matrices $M \in \mathbb{R}^{n\times n}$ and $N \in \mathbb{R}^{n\times n}$

Output: A matrix $P \in \mathbb{R}^{n\times n}$, where $P_{i,j} := \sum_{k=1}^{n} M_{i,k}N_{k,j}$

```
1 for i = 1, 2, ... n:
2     for j = 1, 2, ..., n:
3         for k = 1, 2, ..., n:
4             P_{i,j} := P_{i,j} + M_{i,k}N_{k,j}     // Assume P_{i,j} := 0 to start.
5 return P
```

Figure 6.39 The iterative algorithm for matrix multiplication for n-by-n matrices.

The recursive approach seems clever. But, using Theorem 6.21—we have $a = 8$, $b = 2$, and $k = 2$; thus we are in case (i)—we can conclude that $T(n) = \Theta(n^{\log_2(8)}) = \Theta(n^3)$, so the recursive algorithm is actually not an improvement over Figure 6.39 at all! But, in a major algorithmic breakthrough, in 1969 Volker Strassen found a way to use *seven* recursive calls instead of *eight.* (See Figure 6.41.) This change makes the recurrence $T(n) = 7T(\frac{n}{2}) + n^2$, reducing the 8 to a 7. Now Theorem 6.21—$a = 7$, $b = 2$, and $k = 2$; still case (i)—says that $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.8073\cdots})$, which is a nice improvement. (For example, $1000^{\log_2 7}$ is only about 25% of 1000^3—four times faster!)

$$\begin{bmatrix} R & S \\ T & V \end{bmatrix} \begin{bmatrix} W & X \\ Y & Z \end{bmatrix} = \begin{bmatrix} RW+SY & RX+SZ \\ TW+UY & TX+UZ \end{bmatrix}$$

Figure 6.40 Multiplying two n-by-n matrices by computing the product of eight pairs of $\frac{n}{2}$-by-$\frac{n}{2}$ matrices. We compute $RW, SY, \ldots, UZ$ recursively, and then add the results as indicated to compute the final answer.

With this fundamental idea, one can investigate other Strassen-like algorithms, making fewer recursive calls and combining them cleverly. In 1978, Victor Pan gave a further running-time improvement using this style of algorithm—though more complicatedly!—using a total of 143,640 recursive calls on inputs of size $\frac{n}{70}$ (!), plus $\Theta(n^2)$ additional work. Using Theorem 6.21, that algorithm yields a running time of

$$\begin{bmatrix} R & S \\ T & V \end{bmatrix} \begin{bmatrix} W & X \\ Y & Z \end{bmatrix} = \begin{bmatrix} A+D \\ -E+G & C+E \\ B+D & A-B \\ +C+F \end{bmatrix}$$

where $A := (R+V)(W+Z)$, $B := (T+V)W$, $C := R(X-Z)$, $D := V(Y-W)$, $E := (R+S)Z$, $F := (T-R)(W+X)$, $G := (S-V)(Y+Z)$

Figure 6.41 Strassen's Algorithm. We compute $A, B, \ldots, G$ recursively, and then add/subtract the results as indicated to compute the final answer. The additions and subtractions take $\Theta(n^2)$ time.

$\Theta(n^{\log_{70} 143{,}640}) = \Theta(n^{2.7951\cdots})$. Algorithms continued to improve for several years, culminating in 1990 with an $\Theta(n^{2.3754\cdots})$-time algorithm due to Don Coppersmith and Shmuel Winograd. Their algorithm was the best known for two decades, but recently some new researchers with some new insights have come along, and the exponent is now down to 2.373. (For more about matrix multiplication and the recent algorithmic improvements, see [131], a survey paper by Virginia Vassilevska Williams, one of the researchers responsible for the reinvigorated progress in improving this exponent.) Many people think that the exponent can be improved all the way to 2—but no one has yet found an algorithm that's that fast!

EXERCISES

The following recurrence relations follow the form of Theorem 6.21. Solve each. Assume $T(1) = 1$.

6.109 $T(n) = 4T(\frac{n}{3}) + n^2$

6.110 $T(n) = 3T(\frac{n}{4}) + n^2$

6.111 $T(n) = 2T(\frac{n}{3}) + n^4$

6.112 $T(n) = 3T(\frac{n}{3}) + n$

6.113 $T(n) = 16T(\frac{n}{4}) + n^2$

6.114 $T(n) = 2T(\frac{n}{4}) + 1$

6.115 $T(n) = 4T(\frac{n}{2}) + 1$

6.116 $T(n) = 3T(\frac{n}{3}) + 1$

6.117 $T(n) = 2T(\frac{n}{2}) + n^2$

6.118 $T(n) = 2T(\frac{n}{2}) + n$

6.119 $T(n) = 2T(\frac{n}{4}) + n^2$

6.120 $T(n) = 2T(\frac{n}{4}) + n$

6.121 $T(n) = 4T(\frac{n}{2}) + n^2$

6.122 $T(n) = 4T(\frac{n}{2}) + n$

6.123 $T(n) = 4T(\frac{n}{4}) + n^2$

6.124 $T(n) = 4T(\frac{n}{4}) + n$

6.125 Solve the quadtree recurrence $T(1) = 1$ and $T(n) = 1 + 4T(\frac{n}{4})$ using Theorem 6.21. (See Exercise 6.82.)

6.126 Prove that the recurrences $T(n) = aT(\frac{n}{b}) + c \cdot n^k$ with $T(1) = d$ and $S(n) = aS(\frac{n}{b}) + n^k$ with $S(1) = 1$ have the same asymptotic solution, for any constants $a \geq 1$, $b > 1$, $c > 0$, $d > 0$, and $k \geq 0$. (This equivalence justifies the ways that we have paid little attention to the constant c in recurrences in this section.)

6.127 Consider the recurrence $T(n) = aT(\frac{n}{b}) + n^k$ with $T(1) = 1$. Using induction, prove the equivalence $(*)$ from the proof of Theorem 6.21: prove that $T(n) = n^k \cdot \sum_{i=0}^{\log_b n} (\frac{a}{b^k})^i$ for any n that's an exact power of b.

6.128 Consider the recurrence $T(n) = aT(\frac{n}{b})$ with $T(1) = 1$, for constants $a \geq 1$ and $b > 1$. (This recurrence does not match the form of Theorem 6.21, because there is no n^k term added in the recursive case.) Give a closed-form solution for this recurrence, and prove your answer correct.

6.129 Theorem 6.21 does not apply for the recurrence $T(n) = 2T(\frac{n}{2}) + n \log n$, but the same idea—considering the summation of all the work in the recursion tree—will still work. Prove that $T(n) = \Theta(n \log^2 n)$ by analyzing the summation analogous to $(*)$.

Each of the following exercises gives a brief *description of an algorithm for an interesting problem in computer science. (Sometimes the recurrence relation is explicitly written; sometimes it's up to you to write down the recurrence.) For each, state the recurrence (if it's missing) and give a Θ-bound on the running time. If Theorem 6.21 applies, you may use it. If not, give a proof by induction.*

6.130 The *Towers of Hanoi* is the following classic puzzle. There are three posts (the "towers"); post A starts with n concentric discs stacked in order of their radius (smallest radius at the top, largest radius at the bottom). We must move all the discs to post B, never placing a disc of larger radius on top of a disc of smaller radius. The easiest way to solve this puzzle is with recursion. (See Figure 6.42.) The total number of moves made satisfies $T(n) = 2T(n-1) + 1$ and $T(1) = 1$. Prove that $T(n) = 2^n - 1$.

6.131 Suppose we are given a sorted array $A[1 \ldots n]$, and we wish to determine where in A the element x belongs—that is, the index i such that $A[i-1] < x \leq A[i]$. (Binary Search solves this problem.) Here's a sketch of an algorithm **rootSearch** to solve this problem:

- if n is small (say, less than 100), find the index by brute force. Otherwise:
- define $mileposts := A[\sqrt{n}], A[2\sqrt{n}], A[3\sqrt{n}], \ldots, A[n]$ to be a list of every $(\sqrt{n})$th element of A.
- recursively, find $post :=$ **rootSearch**$(mileposts, x)$.
- return **rootSearch**$(A[(post-1)\sqrt{n}, \ldots, post\sqrt{n}], x)$.

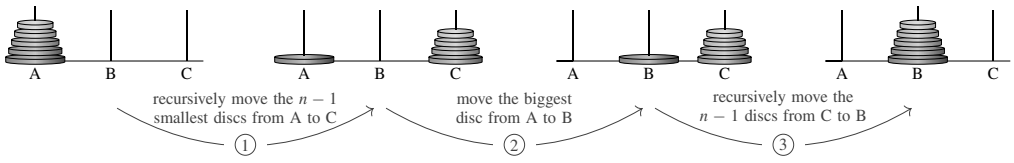


Figure 6.42 Solving the Towers of Hanoi by moving the n discs from post A to B.

(Note that **rootSearch** makes *two* recursive calls: one to identify *which* $\sqrt{n}$-sized subarray is the right one to search in, and one to search in it.) Find a recurrence relation for the running time of this algorithm, and solve it.

6.132 A *van Emde Boas tree* is a recursive data structure (with somewhat similar inspiration to Exercise 6.131) that allows us to insert, delete, and look up *keys* drawn from a set $U = \{1, 2, \ldots, u\}$ quickly. (It solves the same problem that binary search trees solve, but our running time will be in terms of the size of the universe U rather than in terms of the number of keys stored.) A van Emde Boas tree achieves a running time given by $T(u) = T(\sqrt{u}) + 1$ and $T(1) = 1$. Solve this recurrence. *(Hint: define $R(k) = T(2^k)$. Solving $R(k)$ is easier!)*

6.6 Chapter at a Glance

Asymptotics

Asymptotic analysis considers the rate of growth of functions, ignoring multiplicative constant factors and concentrating on the long-run behavior of the function on large inputs. Consider two functions $f : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ and $g : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$. Then $f(n) = O(g(n))$ ("f grows no faster than g") if there exist $c > 0$ and $n_0 \geq 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Some useful properties of $O(\cdot)$:

- $f(n) = O(g(n) + h(n))$ if and only if $f(n) = O(\max(g(n), h(n)))$.
- if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.
- if $f(n) = O(h_1(n))$ and $g(n) = O(h_2(n))$, then $f(n) + g(n)$ is $O(h_1(n) + h_2(n))$.
- similarly, if $f(n) = O(h_1(n))$ and $g(n) = O(h_2(n))$, then $f(n) \cdot g(n)$ is $O(h_1(n) \cdot h_2(n))$.
- a polynomial $p(n) = a_k n^k + \cdots + a_1 n + a_0$ satisfies $p(n) = O(n^k)$.
- $\log n = O(n^\epsilon)$ for any $\epsilon > 0$.
- for any base b and exponent k, we have $\log_b(n^k) = O(\log n)$.
- for constants $b, c \geq 1$, we have $b^n = O(c^n)$ if and only if $b \leq c$.

There are several other forms of asymptotic notation, to capture other relationships between functions. A function *f grows no slower than g*, written $f(n) = \Omega(g(n))$, if there exist constants $d > 0$ and $n_0 \geq 0$ such that $\forall n \geq n_0 : f(n) \geq d \cdot g(n)$. Two functions f and g satisfy $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

A function *f grows at the same rate as g*, written $f(n) = \Theta(g(n))$, if $f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$; it *grows (strictly) slower than g*, written $f(n) = o(g(n))$, if $f(n) = O(g(n))$ but $f(n) \neq \Omega(g(n))$; and it *grows (strictly) faster than g*, written $f(n) = \omega(g(n))$, if $f(n) = \Omega(g(n))$ but $f(n) \neq O(g(n))$. Many of the properties of O have analogous properties for Ω, Θ, o, and ω. One possibly surprising point is that there are functions that are *incomparable*: there are functions f and g such that *neither* $f(n) = O(g(n))$ *nor* $f(n) = \Omega(g(n))$.

Asymptotic Analysis of Algorithms

Our main interest in asymptotics is in the *analysis of algorithms*, so that we can make statements about which of two algorithms that solve the same problem is faster. The *running time* of an algorithm is a count of the number of primitive steps that the algorithm takes to complete on a particular input. (Think of one machine instruction as a primitive step.)

We generally evaluate the efficiency of an algorithm $\mathcal{A}$ using *worst-case analysis*: as a function of n, how many primitive steps does $\mathcal{A}$ take *on the input of size n for which $\mathcal{A}$ is the slowest*. (A primary goal of algorithmic analysis is to provide a guarantee on the running time of an algorithm, so we will be pessimistic.) We can also analyze the *space* used by an algorithm, in the same way. Sometimes we will instead

consider *average-case running time* of an algorithm $\mathcal{A}$, which computes the running time of $\mathcal{A}$, averaged over all inputs of size n. Almost never will we consider an algorithm's running time on the input of size n for which $\mathcal{A}$ is the fastest (known as *best-case analysis*); this type of analysis is rarely used.

Recurrence Relations: Analyzing Recursive Algorithms

Typically, for nonrecursive algorithms, we compute the running time by inspecting the algorithm and writing down a summation corresponding to the operations done in each iteration of each loop, summed over the iterations, and then simplifying. For recursive algorithms, we typically record the work using a *recurrence relation* that expresses the (worst-case) running time on inputs of size n in terms of the (worst-case) running time on inputs of size less than n. (For small inputs, the running time is a constant—say, $T(1) = c$.) For example, ignoring floors and ceilings, $T(1) = c$ and $T(n) = 2T(\frac{n}{2}) + cn$ is the recurrence relation for Merge Sort. (Almost always, we can safely ignore floors and ceilings.)

A *solution* to a recurrence relation is a closed-form (nonrecursive) expression for $T(n)$. Recurrence relations can be solved by conjecturing a solution and proving that conjecture correct by induction.

A recurrence relation can be represented using a *recursion tree,* where each node is annotated with the work that is performed there, aside from the recursive calls, as in Figure 6.43. Recurrence relations can also be solved by summing up all of the work contained within the recursion tree.

An Extension: Recurrence Relations of the Form $T(n) = aT\left(\frac{n}{b}\right) + cn^k$

A particularly common type of recurrence relation is one of the form $T(n) = aT(\frac{n}{b}) + c \cdot n^k$, for constants $a \geq 1$, $b > 1$, $c > 0$, and $k \geq 0$. This type of recurrence arises in divide-and-conquer algorithms that solve an instance of size n by making a different recursive calls on inputs of size $\frac{n}{b}$, and reconstructing the

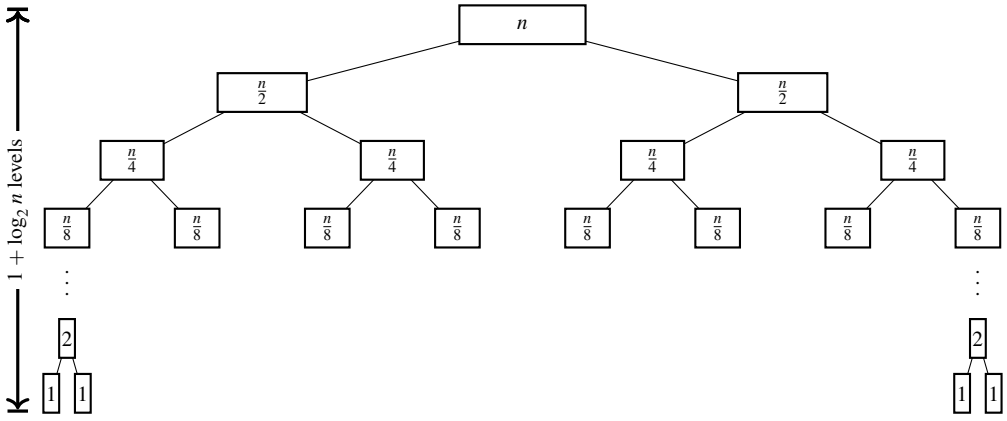


Figure 6.43 A example of a recursion tree, with each input's size marked in the rectangular node.

solution to the given instance in $\Theta(n^k)$ time. Theorem 6.21 states that the solution to any such recurrence relation is given by:

(i) if $b^k < a$, then $T(n) = \Theta(n^{\log_b(a)})$. *"The leaves dominate."*

(ii) if $b^k = a$, then $T(n) = \Theta(n^k \cdot \log n)$. *"All levels are equal."*

(iii) if $b^k > a$, then $T(n) = \Theta(n^k)$. *"The root dominates."*

The proof follows by building the recursion tree, and summing the work at each level of the tree; the cases correspond to whether the work increases exponentially, decreases exponentially, or stays constant across levels of the tree.

Key Terms and Results

Key Terms

Asymptotics

- asymptotic analysis
- O (big oh)
- Ω (big omega)
- Θ (big theta)
- ω (little omega)
- o (little oh)

Analysis of Algorithms

- running time
- worst-case analysis
- average-case analysis
- best-case analysis

Recurrence Relations

- recurrence relation
- recursion tree
- iterating a recurrence

Recurrences of the Form $T(n) = aT\left(\frac{n}{b}\right) + cn^k$

- "the leaves dominate"
- "all levels are equal"
- "the root dominates"

Key Results

Asymptotics

1 Some sample useful properties of $O(\cdot)$:

- $f(n) = O(g(n) + h(n))$ if and only if $f(n) = O(\max(g(n), h(n)))$.
- $O(\cdot)$ is transitive.
- any degree-k polynomial satisfies $p(n) = O(n^k)$.
- $\log n = O(n^\epsilon)$ for any $\epsilon > 0$.
- if $f(n) = O(g(n))$ then $\log f(n) = O(\log g(n))$.
- for any b and k, we have $\log_b(n^k) = O(\log n)$.
- for any $b, c \geq 1$, we have $b^n = O(c^n) \Leftrightarrow b \leq c$.

2 Two functions f and g satisfy $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

3 There are pairs of functions f and g such that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$.

Analysis of Algorithms

1 We generally evaluate the efficiency of an algorithm $\mathcal{A}$ using worst-case analysis: what is (asymptotically) the number of steps consumed by $\mathcal{A}$ as function of the input size *n on the input of size n for which $\mathcal{A}$ is the slowest*?

2 Typically we can analyze the running time of a nonrecursive algorithm by simple counting and manipulation of summations.

Recurrence Relations

1 The running time of a recursive algorithm can be expressed using a recurrence relation, which can be solved by figuring out a conjecture of a closed-form formula for the relation, and then verifying by induction.

Recurrences of the Form $T(n) = aT\left(\frac{n}{b}\right) + cn^k$

1 Recurrence relations of the form $T(n) = aT(\frac{n}{b}) + cn^k$ (and $T(1) = c$) can be solved using Theorem 6.21:

- Case (i): if $b^k < a$, then $T(n) = \Theta(n^{\log_b(a)})$.
- Case (ii): if $b^k = a$, then $T(n) = \Theta(n^k \cdot \log n)$.
- Case (iii): if $b^k > a$, then $T(n) = \Theta(n^k)$.