Stein et.al

4 Induction, Recursion, and Recurrences

4.1 MATHEMATICAL INDUCTION

Smallest Counterexamples

In Section 3.3, we demonstrated one way of proving statements about infinite universes. We considered a "generic" member of the universe and derived the desired statement about that member. When our universe is the universe of integers, or when it is in a one-to-one correspondence with the integers, there is a second technique we can use.

Recall our proof of Euclid's division theorem (Theorem 2.12), which says that when n is a positive integer, for each nonnegative integer m, there exist unique nonnegative integers q and r such that $m = nq + r$ and $0 \leq r < n$. For the purpose of a proof by contradiction, we assumed that there is a nonnegative integer m for which no such q and r exist. We chose a smallest such m and observed that $m - n$ is a nonnegative integer less than m. Then we said:

> Therefore, there exist integers q' and r' such that $m - n = nq' + r'$ with $0 \leq r < n$. But then $m = n(q' + 1) + r'$. So, by taking $q = q' + 1$ and $r = r'$, we obtain $m = qn + r$ with $0 \leq r < n$. This contradicts the assumption that there are no integers q and r with $0 \leq r < n$ such that $m = qn + r$. Thus, by the principle of proof by contradiction, such integers q and r exist.

To analyze these sentences, let $p(m)$ denote the statement "There exist integers q' and r' such that $m - n = nq' + r'$ with $0 \leq r < n$." The first two sentences of the quotation provide a proof that $p(m - n) \Rightarrow p(m)$. This implication is the crux of the proof. Let us give an analysis of the proof that shows the pivotal role of this implication.

- We assumed that a counterexample with a smallest m existed.[1]
- Using the fact that $p(m')$ had to be true for every m' smaller than m, we chose $m' = m - n$ and observed that $p(m')$ had to be true.
- We used the implication $p(m-n) \Rightarrow p(m)$ to conclude the truth of $p(m)$.
- However, we had assumed that $p(m)$ was false, so this assumption is contradicted in the proof by contradiction.

Exercise 4.1-1 In Chapter 1, we learned Gauss's trick for showing that for all positive integers n,

$$1+2+3+4+\cdots+n = \frac{n(n+1)}{2}. \tag{4.1}$$

Use the technique of asserting that if there is a counterexample, then there is a smallest counterexample and deriving a contradiction to prove that the sum is $n(n+1)/2$. What implication did you have to prove in the process?

Exercise 4.1-2 For what values of $n \geq 0$ is $2^{n+1} \geq n^2 + 2$? Use the technique of asserting that if there is a counterexample, then there is a smallest counterexample and deriving a contradiction to prove you are right. What implication did you have to prove in the process?

Exercise 4.1-3 For what values of $n \geq 0$ do you think $2^{n+1} \geq n^2 + 3$? Is it possible to use the technique of asserting that if there is a counterexample, then there is a smallest counterexample and deriving a contradiction to prove you are right? If so, do so and describe the implication you had to prove in the process. If not, why not?

In Exercise 4.1-1, suppose the formula for the sum is false. Then there must be a smallest n such that the formula does not hold for the sum of the first n positive integers. Thus, for any positive integer i smaller than n,

$$1+2+\cdots+i = \frac{i(i+1)}{2}. \tag{4.2}$$

[1] The nonnegative integers are well-ordered.

Because $1 = 1 \cdot 2/2$, Equation 4.1 holds when $n = 1$. Therefore, the smallest counterexample is not $n = 1$. So, $n > 1$, and $n - 1$ is one of the positive integers i for which the formula holds. Substituting $n - 1$ for i in Equation 4.2 yields

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

Adding n to both sides gives

$$\begin{aligned} 1 + 2 + \cdots + n - 1 + n &= \frac{(n-1)n}{2} + n \\ &= \frac{n^2 - n + 2n}{2} \\ &= \frac{n(n+1)}{2}. \end{aligned}$$

Thus, n is not a counterexample after all. Therefore, there is no counterexample to the formula. Hence, the formula holds for all positive integers n. Note that the crucial step was proving that $p(n-1) \Rightarrow p(n)$, where $p(n)$ is the formula

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

In Exercise 4.1-2, let $p(n)$ be the statement $2^{n+1} \geq n^2 + 2$. Some experimenting with small values of n leads us to believe this statement is true for all nonnegative integers. Thus, we want to prove $p(n)$ is true for all nonnegative integers n. To do so, we assume that the statement, "$p(n)$ is true for all nonnegative integers n" is false. When a "for all" statement is false, there must be some n for which it is false. Therefore, there is some smallest nonnegative integer n so that $2^{n+1} \not\geq n^2 + 2$. Assume now that n is this value, which means that $2^{i+1} \geq i^2 + 2$ for all nonnegative integers i with $i < n$. Because we know from our experimentation that $n \neq 0$, we know $n - 1$ is a nonnegative integer less than n. Thus, using $n - 1$ in place of i, we get

$$2^{(n-1)+1} \geq (n-1)^2 + 2,$$

or

$$\begin{aligned} 2^n &\geq n^2 - 2n + 1 + 2 \\ &= n^2 - 2n + 3. \end{aligned} \tag{4.3}$$

From this, we want to draw a contradiction—a contradiction to $2^{n+1} \not\geq n^2 + 2$.

To get the contradiction, we want to convert the left side of Equation 4.3 to 2^{n+1}. For this purpose, we multiply both sides by 2. Because $2^{n+1} = 2 \cdot 2^n$, we may use Equation 4.3 to write

$$2^{n+1} \geq 2 \cdot (n^2 - 2n + 3), \text{ or}$$
$$2^{n+1} \geq 2n^2 - 4n + 6. \tag{4.4}$$

You may get this far and wonder, "What next?" Because we want to obtain a contradiction, we want to convert the right side of Inequality 4.4 into something like $n^2 + 2$. More precisely, we will convert the right side into $n^2 + 2$ plus an additional term. If we can show that the additional term is nonnegative, the proof will be complete. Thus, we write

$$\begin{aligned} 2^{n+1} &\geq 2n^2 - 4n + 6 \\ &= (n^2 + 2) + (n^2 - 4n + 4) \\ &= n^2 + 2 + (n - 2)^2 \\ &\geq n^2 + 2, \end{aligned} \tag{4.5}$$

where the last inequality holds because $(n - 2)^2 \geq 0$. This contradicts our assumption that $2^{n+1} \not\geq n^2 + 2$, so there must not have been a smallest counterexample. Thus, there must be no counterexample. Therefore, $2^n \geq n^2 + 2$ for all nonnegative integers n.

What implication did we prove? Let $p(n)$ stand for $2^{n+1} \geq n^2 + 2$. In Equations 4.3 and 4.5, we proved that $p(n - 1) \Rightarrow p(n)$. At one point in our proof, we had to note that we had considered the case with $n = 0$ already. Although we have given a proof by smallest counterexample, it is natural to ask whether it would make more sense to try to prove the statement $p(n - 1) \Rightarrow p(n)$ directly.

Once we have shown that

$$p(n - 1) \Rightarrow p(n),$$

we can apply it to obtain that $p(0)$ implies $p(1)$, $p(1)$ implies $p(2)$, $p(2)$ implies $p(3)$, and so on. In this way, we have $p(k)$ for every k. Isn't this a more direct proof? We will address this question shortly.

First, let's consider Exercise 4.1-3. Notice that $2^{n+1} \not> n^2 + 3$ for $n = 0$ and $n = 1$, but $2^{n+1} > n^2 + 3$ for any larger n we look at. Let us try to prove that $2^{n+1} > n^2 + 3$ for $n \geq 2$. We now let $p'(n)$ be the statement $2^{n+1} > n^2 + 3$. We can easily prove $p'(2)$ as follows: $8 = 2^3 \geq 2^2 + 3 = 7$. Now, suppose that among the integers larger than 2, there is a counterexample m to $p'(n)$.

That is, suppose there is an m such that $m > 2$ and $p'(m)$ is false. Then there is a smallest such m, and $p'(k)$ is true for k between 2 and $m - 1$. If you look back at your proof that $p(n-1) \Rightarrow p(n)$, you will see that when $n \geq 2$, essentially the same proof applies to p' as well. That is, with very similar computations, we can show that $p'(n-1) \Rightarrow p'(n)$, so long as $n \geq 2$. Thus, because $p'(m-1)$ is true, our implication tells us that $p'(m)$ is also true. This is a contradiction to our assumption that $p'(m)$ is false. Therefore, $p'(m)$ is true.

Again, we could conclude from $p'(2)$ and $p'(2) \Rightarrow p'(3)$ that $p'(3)$ is true (and similarly for $p'(4)$ and so on). This approach seems to give a more direct proof than proof by smallest counterexample. The implication we had to prove was $p'(n-1) \Rightarrow p'(n)$.

The Principle of Mathematical Induction

It may seem clear that repeatedly using the implication $p(n-1) \Rightarrow p(n)$ will prove $p(n)$ for all n (or all $n \geq 2$). This observation is the central idea of the principle of mathematical induction, which we are about to introduce. In a theoretical discussion of the integers, the principle of mathematical induction (or the equivalent **well-ordering principle**—every set of nonnegative integers has a smallest element, which allows us to use the "smallest counterexample" technique) is one of the first principles we assume. The principle of mathematical induction is usually described in two forms. The one we have talked about so far, called the "weak form," applies to statements about integers n.

Principle 4.1 (The Weak Principle of Mathematical Induction)

If the statement $p(b)$ is true and the statement $p(n-1) \Rightarrow p(n)$ is true for all $n > b$, then $p(n)$ is true for all integers $n \geq b$.

Suppose, for example, we wish to give a direct inductive proof that $2^{n+1} > n^2 + 3$ for $n \geq 2$. We would proceed as follows. (The material in square brackets is not part of the proof; it is a running commentary on what is going on in the proof.)

> We will prove by induction that $2^{n+1} > n^2 + 3$ for $n \geq 2$. First, $2^{2+1} = 2^3 = 8$, while $2^2 + 3 = 7$. [We just proved $p(2)$. We will now proceed to prove $p(n-1) \Rightarrow p(n)$.] Suppose now that $n > 2$ and $2^n > (n-1)^2 + 3$. [We just made the hypothesis of $p(n-1)$ in order to use Rule 8 of our rules of inference.]

Now we multiply both sides of this inequality by 2, giving

$$2^{n+1} > 2(n^2 - 2n + 1) + 6.$$

But

$$\begin{aligned} 2(n^2 - 2n + 1) + 6 &= n^2 + 3 + n^2 - 4n + 4 + 1 \\ &= n^2 + 3 + (n-2)^2 + 1. \end{aligned}$$

Therefore, $2^{n+1} > n^2 + 3 + (n-2)^2 + 1$.

Because $(n-2)^2 + 1$ is positive, this proves $2^{n+1} > n^2 + 3$. [We just showed that from the hypothesis of $p(n-1)$, we can derive $p(n)$. Now we can apply Rule 8 to assert that $p(n-1) \Rightarrow p(n)$.] Therefore, $2^n > (n-1)^2 + 3 \Rightarrow 2^{n+1} > n^2 + 3$, and by the principle of mathematical induction, $2^{n+1} > n^2 + 3$ for $n \geq 2$.

In this proof, the sentence "First, $2^{2+1} = 2^3 = 8$, while $2^2 + 3 = 7$" is called the **base case**. It consists of directly proving that $p(b)$ is true, where, in this case, b is 2 and $p(n)$ is $2^{n+1} > n^2 + 3$. The sentence "Suppose now that $n > 2$ and $2^n > (n-1)^2 + 3$" is called the **inductive hypothesis**, which is the assumption that $p(n-1)$ is true. In inductive proofs, we always make such a hypothesis[2] to prove the implication $p(n-1) \Rightarrow p(n)$. The proof of the implication is called the **inductive step**. The final sentence of the proof is called the **inductive conclusion**.

Exercise 4.1-4 Use mathematical induction to show that

$$1 + 3 + 5 + \cdots + (2k-1) = k^2$$

for each positive integer k.

Exercise 4.1-5 For what values of n is $2^n > n^2$? Use mathematical induction to show that your answer is correct.

For Exercise 4.1-4, we note that the formula holds when $k = 1$. Assume inductively that the formula holds when $k = n - 1$, so that $1 + 3 + \cdots +$

[2] At times, it might be more convenient to assume that $p(n)$ is true and use this assumption to prove that $p(n+1)$ is true. This proves the implication $p(n) \Rightarrow p(n+1)$, which lets us reason in the same way.

$(2n-3) = (n-1)^2$. Adding $2n-1$ to both sides of this equation gives

$$\begin{aligned} 1+3+\cdots+(2n-3)+(2n-1) &= n^2-2n+1+2n-1 \\ &= n^2. \end{aligned} \tag{4.6}$$

Thus, the formula holds when $k = n$, and so, by the principle of mathematical induction, the formula holds for all positive integers k.

Notice that in our discussion of Exercise 4.1-4, nowhere did we mention a statement $p(n)$. In fact, $p(n)$ is the statement we get by substituting n for k in the formula. In Equation 4.6, we were proving $p(n-1) \Rightarrow p(n)$. Next, notice that we did not explicitly say we were going to give a proof by induction; instead, we indicated that we were making an inductive proof when we were making the inductive hypothesis by saying, "Assume inductively that ...." This convention makes the prose flow nicely but still tells the reader that he or she is reading a proof by induction. Notice also how the notation in the statement of the exercise helped us write the proof. If we state what we are trying to prove in terms of a variable other than n, such as k, then we can assume that our desired statement holds when this variable, k, is $n-1$ and then prove that the statement holds when $k = n$. Without this notational device, we have to either mention our statement $p(n)$ explicitly or avoid any discussion of substituting values into the formula we are trying to prove. Our proof that $2^{n+1} > n^2 + 3$ demonstrates this last approach to writing an inductive proof in plain English. This approach is usually the "slickest" way of writing an inductive proof (though it is often the hardest to master). We will use this approach first for the next exercise.

For Exercise 4.1-5, we note that $2 = 2^1 > 1^2 = 1$, but then the inequality fails for $n = 2, 3, 4$. However, $32 > 25$. Now we assume inductively that for $n > 5$, we have $2^{n-1} > (n-1)^2$. Multiplying by 2 gives us the following:

$$\begin{aligned} 2^n &> 2(n^2-2n+1) \\ &= n^2+n^2-4n+2 \\ &> n^2+n^2-n\cdot n \\ &= n^2, \end{aligned}$$

because $n > 5$ implies that $-4n > -n \cdot n$. (We also used the fact that $n^2+n^2-4n+2 > n^2+n^2-4n$.) Thus, by the principle of mathematical induction, $2^n > n^2$ for all $n \geq 5$.

Alternatively, we could write the following: Let $p(n)$ denote the inequality $2^n > n^2$. Then $p(5)$ is true, because $32 > 25$. Assume that $n > 5$ and that

$p(n-1)$ is true. This implies that $2^{n-1} > (n-1)^2$. Multiplying by 2 gives us the following:

$$\begin{aligned} 2^n &> 2(n^2 - 2n + 1) \\ &= n^2 + n^2 - 4n + 2 \\ &> n^2 + n^2 - n \cdot n \\ &= n^2, \end{aligned}$$

because $n > 5$ implies that $-4n > -n \cdot n$. Therefore, $p(n-1) \Rightarrow p(n)$. Thus, by the principle of mathematical induction, $2^n > n^2$ for all $n \geq 5$.

Notice how the "slick" method simply assumes that the reader knows we are doing a proof by induction from our "Assume inductively...." It also assumes the reader mentally supplies the appropriate $p(n)$ and observes that we have proved $p(n-1) \Rightarrow p(n)$ at the right moment.

Here is a slight variation of the technique of changing variables. To prove that $2^n > n^2$ when $n \geq 5$, we observe that the inequality holds when $n = 5$, because $32 > 25$. Assume inductively that the inequality holds when $n = k$, so that $2^k > k^2$. Now, when $k \geq 5$, multiplying both sides of this inequality by 2 yields the following sequence of inequalities (which are explained in the text that follows):

$$\begin{aligned} 2^{k+1} &> 2k^2 \\ &= k^2 + k^2 \\ &> k^2 + 5k \\ &> k^2 + 2k + 1 \\ &= (k+1)^2, \end{aligned}$$

because $k \geq 5$ implies that $k^2 \geq 5k$ and $5k = 2k + 3k > 2k + 1$. Thus, by the principle of mathematical induction, $2^n > n^2$ for all $n \geq 5$.

This last variation of the proof illustrates two ideas. First, there is no need to save the name n for the variable we use in applying mathematical induction. We used k as our inductive variable in this case. Second, as suggested in footnote 2, there is no need to restrict ourselves to proving the implication $p(n-1) \Rightarrow p(n)$. In this case, we proved the implication $p(k) \Rightarrow p(k+1)$. Clearly, these two implications are equivalent as n ranges over all integers larger than b and as k ranges over all integers larger than or equal to b.

Strong Induction

In our proof of Euclid's division theorem, we had a statement of the form $p(m)$, and, assuming that it was false, we chose a smallest m such that $p(m)$ was false for some n. This choice meant that we could assume that $p(m')$ is true for *all* nonnegative $m' < m$. We needed this assumption because we had to show that $p(m-n) \Rightarrow p(m)$ in order to get our contradiction. This situation differs from the examples we used to introduce mathematical induction, because in those examples we used an implication of the form $p(n-1) \Rightarrow p(n)$. The essence of our method in proving Euclid's division theorem is the following:

1. We have a statement $q(k)$ that we want to prove for all k larger than some integer.
2. We suppose it is false; so, there must be a smallest k for which $q(k)$ is false.
3. The previous step implies that we may assume $q(k')$ is true for *all* k' in the universe of q with $k' < k$.
4. We then use this assumption to derive a proof of $q(k)$, thus generating our contradiction.

Again, we can avoid the step of generating a contradiction in the following way. Suppose first we have a proof of $q(0)$. Suppose also we have a proof that

$$q(0) \wedge q(1) \wedge q(2) \wedge \cdots \wedge q(k-1) \Rightarrow q(k)$$

for all k larger than 0. Then, from $q(0)$, we can prove $q(1)$; from $q(0) \wedge q(1)$, we can prove $q(2)$; from $q(0) \wedge q(1) \wedge q(2)$, we can prove $q(3)$; and so on. This method gives us a proof of $q(n)$ for any n we desire, and is another form of the principle of mathematical induction. We use this approach when, as in Euclid's division theorem, we can get an implication of the form $q(k') \Rightarrow q(k)$ for *some* $k' < k$ or when we can get an implication of the form $q(0) \wedge q(1) \wedge q(2) \wedge \cdots \wedge q(k-1) \Rightarrow q(k)$. (As is the case in Euclid's division theorem, we often don't really know what k' is, so the first kind of situation is really just a special case of the second. It is for this reason that we do not treat the first of the two implications separately.) We have just described the method of proof known as the strong principle of mathematical induction.

Principle 4.2 (The Strong Principle of Mathematical Induction)
If the statement $p(b)$ is true and the statement $p(b) \wedge p(b+1) \wedge \cdots \wedge p(n-1) \Rightarrow p(n)$ is true for all $n > b$, then $p(n)$ is true for all integers $n \geq b$.

The terms weak and strong arise from what is assumed in the inductive hypothesis. Adding more restrictions strengthens an assertion, while removing restrictions weakens the assertion. For example, stating that Sandy is a teenager is a weaker assertion than stating that Sandy is 16 years old. In weak induction our inductive hypothesis is only $p(n-1)$. In strong induction it is not only $p(n-1)$, but also $p(b) \wedge p(b+1) \wedge \ldots \wedge p(n-2)$. This is a stronger assertion.

Exercise 4.1-6 Prove that every positive integer is either a power of a prime number or the product of powers of prime numbers.

In Exercise 4.1-6, we observe that 1 is a power of a prime number; for example, $1 = 2^0$. Suppose now that we know that every number less than n is a power of a prime number or a product of powers of prime numbers. Then, if n is not a prime number, it is a product of two smaller numbers, each of which is, by our supposition, a power of a prime number or a product of powers of prime numbers. But multiplying two powers of primes or products of powers of primes gives a product of powers of primes. Therefore, n is a power of a prime number or a product of powers of prime numbers. Thus, by the strong principle of mathematical induction, every positive integer is a power of a prime number or a product of powers of prime numbers.

Note that there was no explicit mention of an implication of the form

$$p(b) \wedge p(b+1) \wedge \cdots \wedge p(n-1) \Rightarrow p(n).$$

Note also that we did not explicitly identify the base case or the inductive hypothesis in our proof. These are common conventions with inductive proofs. Readers of inductive proofs are expected to recognize when the base case is being given and when an implication of the form $p(n-1) \Rightarrow p(n)$ or $p(b) \wedge p(b+1) \wedge \cdots \wedge p(n-1) \Rightarrow p(n)$ is being proved.

Mathematical induction is used frequently in discrete math and computer science. Many quantities that we are interested in measuring, such as running time or space used in memory, typically are restricted to positive

integers. Thus, mathematical induction is a natural way to prove facts about these quantities. We will use it frequently throughout this book. We typically will not distinguish between strong and weak induction; we just think of them both as induction. (Problems 13 and 14 ask you to derive each version of the principle from the other.)

Induction in General

We now summarize what we have said so far. A typical proof by mathematical induction showing that a statement $p(n)$ is true for all integers $n \geq b$ consists of three steps.

1. We show that $p(b)$ is true. This step is called establishing a base case.
2. We either show that
$$p(n-1) \Rightarrow p(n)$$
for all $n > b$ or show that
$$p(b) \wedge p(b+1) \wedge \cdots \wedge p(n-1) \Rightarrow p(n)$$
for all $n > b$. For this purpose, we make either the inductive hypothesis $p(n-1)$ or the inductive hypothesis $p(b) \wedge p(b+1) \wedge \cdots \wedge p(n-1)$. Then we derive $p(n)$ to complete the proof of the implication we desire—either $p(n-1) \Rightarrow p(n)$ or $p(b) \wedge p(b+1) \wedge \cdots \wedge p(n-1) \Rightarrow p(n)$.
3. We conclude on the basis of the principle of mathematical induction that $p(n)$ is true for all integers n greater than or equal to b.

The second step is the core of an inductive proof, and is usually where we need the most insight into what we are trying to prove. Looking back on the examples of induction in this chapter, you may notice that in Example 4.1-5, we did not show that $p(n-1) \Rightarrow p(n)$; instead, we showed that $p(n) \Rightarrow p(n+1)$. Logically, in the context of an inductive proof, these statements are equivalent (simply substitute m for $n-1$). For convenience, we now restate an alternate to condition 2:

2′ We show that for all $n \geq b$, either
$$p(n) \Rightarrow p(n+1)$$
or
$$p(b) \wedge p(b+1) \wedge \cdots \wedge p(n) \Rightarrow p(n+1).$$

For this purpose, we make either the inductive hypothesis $p(n)$ or the inductive hypothesis $p(b) \wedge p(b+1) \wedge \cdots \wedge p(n)$. Then we derive $p(n+1)$ to complete the proof of the implication we desire—either $p(n) \Rightarrow p(n+1)$ or $p(b) \wedge p(b+1) \wedge \cdots \wedge p(n) \Rightarrow p(n+1)$.

It is important to realize that induction arises in some circumstances that do not fit the typical description we just gave. First, instead of a single base case, we may need multiple base cases. Second, instead of needing to show just one implication that demonstrates that $p(n)$ is true given that $p(n')$ is true for some set of $n' < n$, we may need to show a set of such implications.

For example, consider the problem of proving the following statement:

$$\sum_{i=0}^{n} \left\lfloor \frac{i}{2} \right\rfloor = \begin{cases} \frac{n^2}{4} & \text{if } n \text{ is even}, \\ \frac{n^2-1}{4} & \text{if } n \text{ is odd}. \end{cases} \tag{4.7}$$

To prove this, we must show that $p(0)$ is true, $p(1)$ is true, $p(n-2) \Rightarrow p(n)$ if n is odd, and $p(n-2) \Rightarrow p(n)$ if n is even. Putting all these together, we see that our formulas hold for all $n \geq 0$. We can view this as either two proofs by induction, one for even and one for odd numbers, or one proof in which we have two base cases and two methods of deriving results from previous ones. The second view is more useful because it expands our idea of what induction means and makes it easier to find inductive proofs. In this proof of Equation 4.7, we have two base cases and two inductive implications. We could also find situations where we have just one implication to prove but several base cases to check (we will see one such situation shortly) or just one base case but several different implications to prove.

Logically speaking, we could rework the proof of Equation 4.7 above so that it fits the pattern of strong induction. For example, when we prove a second base case, then we have just proved that the first base case implies it, because a true statement implies a true statement. However, in the mathematics literature and especially in the computer science literature, inductive proofs are written with multiple base cases and multiple implications with no effort to reduce them to one of the standard forms of mathematical induction. As long as it is possible to cover all the cases under consideration with such a proof, it can be rewritten as a standard inductive proof. Because readers of such proofs are expected to know this convention, and because reworking such a proof as a standard inductive proof adds unnecessary verbiage, the proofs are almost never rewritten into the "standard form."

A Recursive View of Induction

Those familiar with recursive programs might notice similarities between induction and recursion.[3] Both talk about base cases. Both may appear at first glance to be circular. In recursion, a function calls itself. When we prove the implication in the inductive step of an inductive proof, we prove a property for an instance of size n by assuming the property is true for other instances. In both cases, the same thing prevents circularity:

- The instances solved by recursion when the function calls itself are always smaller than the current instance and the recursion eventually gets down to base cases that are dealt with directly.
- The instances assumed in the inductive step of an inductive proof are always smaller than the current instance and the induction eventually gets down to base cases that are dealt with directly.

Students who have written a number of recursive programs come to understand that recursion works. As long as all the recursive calls are all to smaller-sized instances, the recursion is not circular. As long as the base cases are handled correctly and larger instances are correctly solved by building on solutions to smaller instances, the recursion will terminate and compute the correct answer.

In this section we use this understanding of recursion to present an alternate view of induction. We do this for two reasons. First, many people find induction difficult and counterintuitive, and seeing it explained from a number of different views can help them to understand it. Second, thinking of induction in terms of recursion is a very useful way of developing inductive proofs.

An inductive proof can be seen as a description of a recursive program that will print a complete, horribly detailed proof for any chosen instance of size n as long as n is bigger than some value. Because recursion works, we can call this program to print a proof for any n. Because a program exists that can generate a complete proof for any n, the property must be true for all n.

For our example we again show that for any positive integer n,

$$s(n) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

The recursive program works by calling itself to prove that $s(n-1) = (n-1)n/2$, and then using this proven lemma to verify the formula for n.

[3] In this section, when we speak of recursion, we mean recursion in a computer program. We simply say recursion to avoid being repetitive.

The base case is used when the recursion no longer works. We are proving the formula for positive integers, so $s(n)$ is defined when n is a positive integer. If we were to recursively try to prove the formula is correct for $s(1)$, it would try to do this in terms of $s(0)$. But 0 is not a positive integer, so $s(0)$ is not defined. Therefore we will have to prove that the formula is correct for $s(1)$ directly, without recursion.

The following program accomplishes this.

```
ProveSum(n)

// Assume that n is a positive integer.
// This is a recursive program that inputs n and prints a detailed proof
// showing that s(n) = n*(n+1)/2.
(1)  if (n == 1)
(2)       print "We note that"
(3)       print "  s(1) = 1 = 1*2/2, so the formula is correct for n = 1."
(4)  else
(5)       print "To prove that s(", n, ") = ", n , "*", n+1,
            "/2, we first prove that"
(6)       print "  s(", n-1, ") = ", n-1, "*", n, "/2."
(7)       proveSum(n-1)
(8)       print "Having proved s(", n-1,  ") = ", n-1, "*", n, "/2 = ",
            (n-1)*n/2," we add ", n
(9)       print "  to the first and last values, getting ",
            "s(", n, ") = ", ((n-1)*n/2 + n), "."
(10)      print "  This equals ", n, "*", n+1,
            "/2, so the formula is correct for n = " , n, "."
```

The print statements are messy but the code is fairly straighforward. It tests to see if we are in the base case ($n = 1$) and if so prints the proof for $s(1)$. Otherwise it calls itself recursively to print a proof for $s(n-1)$ and uses that result to prove the formula for $s(n)$. The output of the call ProveSum(4) is:

```
To prove that s(4) = 4*5/2, we first prove that
  s(3) = 3*4/2.
To prove that s(3) = 3*4/2, we first prove that
  s(2) = 2*3/2.
To prove that s(2) = 2*3/2, we first prove that
  s(1) = 1*2/2.
We note that
  s(1) = 1 = 1*2/2, so the formula is correct for n = 1.
Having proved s(1) = 1*2/2 = 1 we add 2
  to the first and last values, getting s(2) = 3.
  This equals 2*3/2, so the formula is correct for n = 2.
Having proved s(2) = 2*3/2 = 3 we add 3
  to the first and last values, getting s(3) = 6.
  This equals 3*4/2, so the formula is correct for n = 3.
```

```
Having proved s(3) = 3*4/2 = 6 we add 4
  to the first and last values, getting s(4) = 10.
  This equals 4*5/2, so the formula is correct for n = 4.
```

We do not expect you to write such a program when you are asked to do a proof by induction. However, thinking recursively is often the easiest way to discover an inductive proof. Given an instance for which you are trying to prove a property, start by figuring out how to break it down into one or more smaller instances of the same form. Because the instances are smaller, you can assume that the property is true for them; after all, if you had to, you could generate a proof for that case by writing a recursive program, as we just did. Then you show how the fact that the property holds for these smaller instances implies that the property holds for the original instance. Finally, you decide at what point the recursive decomposition stops giving problems of the same form. These problems, which cannot be recursively decomposed into smaller problems, are the base cases. You must check directly that the property holds for the base cases.

Note that this procedure is the reverse of the way that a proof is actually written. The recursive decomposition is developed first. The decomposition then determines whether the induction is strong or weak and what base cases are needed. Proofs could be written in this form, but it is traditional to prove the base cases first and then show that the smaller cases imply the larger one.

When proving the validity of *formulas* via induction, it sometimes helps, as we did implicitly in Exercise 4.1-4, to think of how to "grow" a smaller case into a larger one. However, it is usually more profitable to think of decomposing a larger case into smaller ones than to think of building a smaller case to create a larger one. (Building a smaller case to a larger one is just one way of seeing how to decompose the larger case into smaller ones.) As we will see in Sections 6.1 and 6.2 (especially in Exercises 6.2-6 and 6.2-7), there are times when this way of thinking is clearly the best way to get a valid proof. Such examples occur throughout computer science. Therefore, it is good to get in the habit of doing induction by starting with a larger instance and recursively decomposing it to get smaller instances.

There are two other advantages to this top-down approach. First, if we decompose the problem recursively we know that all possible larger instances can be decomposed in this way. "Building up" small cases into larger ones requires an additional step, namely showing that all larger cases can be created by using the construction given. It is often the case that a "building up" process constructs a proper subset of the possible cases. A "building up" proof must show that all larger cases can be covered.

Second, with a top-down approach, there is no question about what our base case or base cases should be. The base cases are the ones where the

recursive decomposition no longer works. This answers a question students often ask, namely, "How do I choose my base case or cases?"

To demonstrate this idea, we reconsider the proof that every positive integer is a prime or the product of powers of primes. The recursive decomposition is to factor a number into two smaller factors, which is always possible unless the number is a prime or is 1. Thus, our base cases are $1 = 2^0$ and all the primes. (As you may recall, our base case in our first solution of Exercise 4.1-6 was simply the case where the number is 1.) In all these cases, our number is either a prime or a power of a prime. For any other number n, we assume that the property holds for all $k < n$. Because the number is not a prime or 1, we can factor it into two smaller numbers, and, by our inductive hypothesis, each is either prime or the product of powers of primes. Multiplying two products of powers of primes gives another product of powers of primes. Thus, our number is the product of powers of primes. By the strong principle of mathematical induction, every positive integer is a prime or a product of powers of primes.

It may seem strange to talk about an infinite number of base cases (all the primes), but these would be the base cases of a recursive program to factor a number into a product of powers of primes. If you reread the original solution to Exercise 4.1-6, then you will see that primes are a special case handled without using the inductive hypothesis. They are base cases in the recursive sense, and whether we choose to call them base cases in the inductive sense or to view them as inductive cases that don't require the inductive hypothesis to prove them is a matter of taste.[4] Inductive proofs are often cleaner if we define a base case to be any case that does not use the inductive hypothesis in its proof.

Structural Induction

So far we have treated induction as a proof method that works on integers. However, there are other options. In the last section we noted that in computer science we often want to prove things about structures. Examples of

[4]Recall that in the truth table for $p \Rightarrow q$, in each row in which q is true, the statement $p \Rightarrow q$ is true as well. Thus, one thing that would prove $p \Rightarrow q$ to be true is a proof of q that does not make any assumption about p. This is what we were doing when we wrote, "Suppose now we know that every number less than n is a power of a prime number or a product of powers of prime numbers. Then, if n is not a prime number, it is a product of two smaller numbers." We were treating the cases where n is a prime number as special cases in which our conclusion could be shown to be true without using the hypothesis.

such structures include sets, lists, trees, and graphs, and induction is a very common method of proof in these cases. The recursive decomposition of a problem on a structure usually requires solving the problem on one or more proper substructures of the same form as the original structure. The inductive hypothesis is assumed to hold for these substructures, and the inductive hypothesis is proved for the original structure. This approach of assuming that the inductive hypothesis is true for proper substructures is called structural induction.

It is possible to turn a proof using structural induction into a normal induction on integers by defining the "size" of a structure in such a way that the size of any proper substructure is smaller than the size of the original structure. We then use induction (strong or weak, as appropriate) on the size of the structure. However, this introduces the extra step of finding an appropriate definition of size, and this can make the proof less clear. It is often easier to simply assume that the inductive hypothesis is true on all smaller structures, where "smaller than" means "is a proper substructure of."

As an example, we will consider a theorem about triangulated polygons. To triangulate a polygon one keeps adding diagonals connecting pairs of vertices until no more diagonals can be added. These diagonals must lie entirely interior to the polygon and are not allowed to intersect. They break the interior of the polygon into a number of triangles, because any larger polygon can be split by adding a diagonal. (This fact is perhaps not obvious, but we won't get sidetracked by proving it here.) An example of a triangulated polygon appears in Figure 4.1. We say that a vertex of the polygon is incident to a diagonal if it is an endpoint of the diagonal. We define an ear of a triangulated polygon as a vertex that is not incident to any diagonal. We also say that two vertices are adjacent if they are connected by an edge in the polygon.

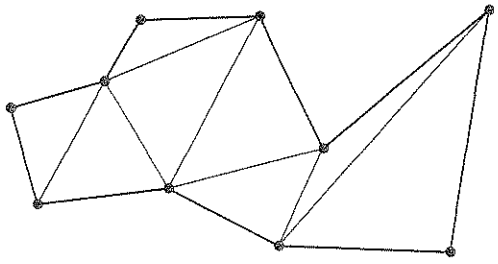


Figure 4.1: A triangulated polygon

We want to prove the Ear Lemma, which states that a triangle has three ears, and a larger triangulated polygon has at least two ears that were not adjacent in the original polygon. We will prove this via structural induction.

We first note that if the polygon is a triangle, it has three ears. This is the base case.

We must now recursively decompose the triangulated polygon into one or more smaller triangulated polygons. One way to do this would be to remove an ear and the two edges adjacent to it. However, this approach has a problem: how do we know that such an ear exists? Therefore we choose a different decomposition.

If the triangulated polygon is larger than a triangle it has at least one diagonal. We split the triangulated polygon into two smaller triangulated polygons along some diagonal (see Figure 4.2). For each subproblem the diagonal becomes an edge in the smaller polygon. These triangulated polygons are smaller than the original one, so by our inductive hypothesis each is either a triangle with three ears or a larger polygon with two nonadjacent ears. We consider what happens to these ears when we rejoin the two polygons into the larger polygon by joining them along the diagonal. If a polygon is a triangle the new diagonal will eliminate two of the ears, leaving one ear in the triangle. If it is a larger polygon the diagonal can be incident to at most one of the two nonadjacent ears, because endpoints of the diagonal are adjacent in the subproblem. Thus there must be at least one remaining ear in this subproblem. At least one ear remaining in each subproblem after joining means that there must be at least two ears in the original triangulated polygon. They cannot be adjacent because they are separated by the endpoints of the diagonal. Thus by the principal of mathematical induction we have proved the Ear Lemma.

Important Concepts, Formulas, and Theorems

1. *Weak principle of mathematical induction.* The *weak principle of mathematical induction* states that if the statement $p(b)$ is true and the statement $p(n-1) \Rightarrow p(n)$ is true for all $n > b$, then $p(n)$ is true for all integers $n \geq b$.
2. *Strong principle of mathematical induction.* The *strong principle of mathematical induction* states that if the statement $p(b)$ is true and the statement $p(b) \wedge p(b+1) \wedge \cdots \wedge p(n-1) \Rightarrow p(n)$ is true for all $n > b$, then $p(n)$ is true for all integers $n \geq b$.
3. *Base case.* Every proof by mathematical induction, strong or weak, begins with a *base case,* which establishes the result being proved for at least one value of the variable on which we are inducting. This

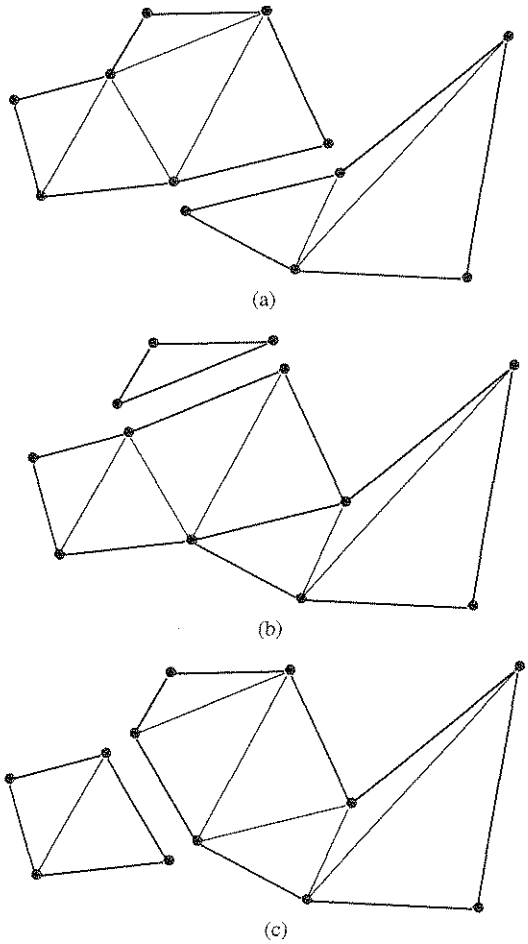


Figure 4.2: Three ways to decompose a triangulated polygon along a diagonal

base case should prove the result for the smallest value of the variable for which we are asserting the result. In a proof with multiple base cases, the base cases should cover all values of the variable that are not covered by the inductive step of the proof.

4. *Inductive hypothesis.* Every proof by induction includes an inductive hypothesis in which we assume that the result $p(n)$ we are trying to prove is true when $n = k - 1$ or when $n < k$ (or in which we assume an equivalent statement).
5. *Inductive step.* Every proof by induction includes an inductive step in which we prove the implication that $p(k-1) \Rightarrow p(k)$ or the implication that $p(b) \wedge p(b+1) \wedge \cdots \wedge p(k-1) \Rightarrow p(k)$, or some equivalent implication.

6. *Inductive conclusion.* A proof by mathematical induction should include, at least implicitly, a concluding statement of the form "Thus, by the principle of mathematical induction ...," which asserts that by the principle of mathematical induction, the result $p(n)$ that we are trying to prove is true for all values of n, including and beyond the base case(s).
7. *Structural induction.* In computer science we often prove things about structures (e.g., lists, graphs, and trees). While it is possible to do normal induction on integers on the size of the structures, it is usually simpler to do the induction directly on the structures. Assume that the inductive hypothesis is true for all proper substructures of a given structure, and use this assumption to prove it for the given structure. Structures that have no proper substructures are the base cases and must be proved directly.

Problems

All problems with blue boxes have an answer or hint available at the end of the book.

1. This problem explores ways to prove that

$$\frac{2}{3}+\frac{2}{9}+\cdots+\frac{2}{3^n}=1-\left(\frac{1}{3}\right)^n$$

for all positive integers n.

a. First, we explore how to prove the formula by contradiction. In other words, assume that there is some integer n that makes the formula false. In this case, there must be some smallest n that makes the formula false.
   i. Can this smallest n be 1?
   ii. What do you know about

$$\frac{2}{3}+\frac{2}{9}+\cdots+\frac{2}{3^i}$$

   when i is a positive integer smaller than this smallest n?
   iii. Is $n-1$ a positive integer for this smallest n?
   iv. What do you know about

$$\frac{2}{3}+\frac{2}{9}+\cdots+\frac{2}{3^{n-1}}$$

   for this smallest n?

v. Write the answer to part iv as an equation, add $2/3^n$ to both sides, and simplify the right side.
vi. What does the equation that results from part v say about your assumption that the formula is false?
vii. What can you conclude about the truth of the formula?
viii. If $p(k)$ is the statement

$$\frac{2}{3}+\frac{2}{9}+\cdots+\frac{2}{3^k}=1-\left(\frac{1}{3}\right)^k,$$

what implication did you prove in the process of deriving your contradiction?

b. i. What is the base case in a proof by mathematical induction that

$$\frac{2}{3}+\frac{2}{9}+\cdots+\frac{2}{3^n}=1-\left(\frac{1}{3}\right)^n$$

for all positive integers n?
ii. What would you assume as an inductive hypothesis?
iii. What would you prove in the inductive step of a proof of this formula by induction?
iv. Prove it.
v. What does the principle of mathematical induction allow you to conclude?
vi. If $p(k)$ is the statement

$$\frac{2}{3}+\frac{2}{9}+\cdots+\frac{2}{3^k}=1-\left(\frac{1}{3}\right)^k,$$

what implication did you prove in the process of doing your proof by induction?

2. Use contradiction to prove
$1\cdot 2+2\cdot 3+\cdots+n(n+1)=n(n+1)(n+2)/3$.

3. Use induction to prove that
$1\cdot 2+2\cdot 3+\cdots+n(n+1)=n(n+1)(n+2)/3$.

4. Prove that $1^3+2^3+3^3+\cdots+n^3=n^2(n+1)^2/4$.

5. Use strong induction to write a careful proof of Euclid's division theorem.

6. Prove that $\sum_{i=j}^{n}\binom{i}{j}=\binom{n+1}{j+1}$. In addition to an inductive proof, there is a nice "story" proof of this formula. It is well worth trying to figure out both proofs.

7. Prove that every number greater than 7 is a sum of a nonnegative integer multiple of 3 and a nonnegative integer multiple of 5.

8. We can define the nonnegative powers of a number a by the rules $a^0 = 1$ and $a^{n+1} = a^n \cdot a$. Explain why this defines a^n for all nonnegative integers n. From this definition, prove the rule of exponents $a^{m+n} = a^m a^n$ for nonnegative integers m and n.

9. Our arguments in favor of the sum principle were quite intuitive. In fact, the sum principle for n sets follows from the sum principle for two sets. Use induction to prove the sum principle for a union of n sets from the sum principle for a union of two sets.

10. We have proved that every positive integer is a power of a prime number or a product of powers of prime numbers. Show that this factorization is unique in the following sense: If you have two factorizations of a positive integer, both factorizations use exactly the same primes, and each prime occurs to the same power in both factorizations. For this purpose, it is helpful to know that if a prime divides a product of integers, then it divides one of the integers in the product. (Another way to say this is that if a prime is a factor of a product of integers, then it is a factor of one of the integers in the product.)

11. Find the error in the following "proof" that all positive integers n are equal: Let $p(n)$ be the statement that all numbers in an n-element set of positive integers are equal. Then $p(1)$ is true. Now assume $p(n-1)$ is true, and let N be the set of the first n integers. Let N' be the set of the first $n-1$ integers, and let N'' be the set of the last $n-1$ integers. By $p(n-1)$, all members of N' are equal, and all members of N'' are equal. Thus, the first $n-1$ elements of N are equal and the last $n-1$ elements of N are equal, and so all elements of N are equal. Therefore, all positive integers are equal.

12. Prove by induction that the number of subsets of an n-element set is 2^n.

13. Prove that the strong principle of mathematical induction implies the weak principle of mathematical induction.

14. Prove that the weak principle of mathematical induction implies the strong principle of mathematical induction.

15. Prove Statement 4.7.

16. An alternate version of the Ear Lemma states that a triangulated polygon is either a triangle with three ears or has at least two ears.

(This version does not specify that the ears are nonadjacent.) What happens if we try proving this by induction, using the same decomposition that we used in proving the Ear Lemma?

17. There is a relationship between the number of vertices in a polygon and the number of triangles in any triangulation of that polygon. State this relationship and prove it by induction.

4.2 RECURSION, RECURRENCES, AND INDUCTION

Recursion

Exercise 4.2-1 Describe how you have used recursion when writing programs. Include as many uses as you can.

Exercise 4.2-2 A standard problem for computer science students who are learning about recursion is the Tower of Hanoi problem. In this problem, we have three pegs numbered 1, 2, and 3. One peg has a stack of n disks, each smaller in diameter than the one below it, as in Figure 4.3. An allowable move consists of removing a disk from one peg and sliding it onto another peg so that it is not above another disk of smaller size. We are to determine how many allowable moves are needed to move the disks from one peg to another. Describe the strategy you have used or would use in a recursive program to solve this problem.



Figure 4.3: The Tower of Hanoi

For the Tower of Hanoi problem, to solve the problem with no disks, do nothing. To solve the problem of moving all n disks to Peg 3, do the following:

1. Recursively solve the problem of moving the top $n - 1$ disks from Peg 1 to Peg 2.
2. Move Disk n to Peg 3.
3. Recursively solve the problem of moving the $n - 1$ disks on Peg 2 to Peg 3.

Thus, if $M(n)$ is the number of moves needed to move n disks from Peg i to Peg j, we have

$$M(n) = 2M(n-1) + 1.$$

This equation is an example of a **recurrence equation** or **recurrence**. A recurrence equation for a function defined on the set of integers greater than or equal to some number b is one that tells us how to compute the nth value of a function from the $(n-1)$st value or how to compute the nth value from some or all the first $n-1$ values. To specify completely a function on the basis of a recurrence, we have to give enough information about the function to get started. This information is called the **initial condition** (or the initial conditions) (which we also call the *base case*) for the recurrence. In this case, we have said that $M(0) = 0$. Using this, we get from the recurrence that $M(1) = 1$, $M(2) = 3$, $M(3) = 7$, $M(4) = 15$, and $M(5) = 31$. We are led to guess that $M(n) = 2^n - 1$.

Formally, we write our recurrence and initial condition together as

$$M(n) = \begin{cases} 0 & \text{if } n = 0\,, \\ 2M(n-1) + 1 & \text{otherwise}\,. \end{cases} \tag{4.8}$$

Now we give an inductive proof that our guess that $M(n) = 2^n - 1$ is correct. The base case is trivial, because we have defined $M(0) = 0$, and $0 = 2^0 - 1$. For the inductive step, we assume that $n > 0$ and $M(n-1) = 2^{n-1} - 1$. From the recurrence, $M(n) = 2M(n-1) + 1$. But, by the inductive hypothesis, $M(n-1) = 2^{n-1} - 1$; so, we get that

$$\begin{aligned} M(n) &= 2M(n-1) + 1 \\ &= 2(2^{n-1} - 1) + 1 \\ &= 2^n - 1. \end{aligned}$$

Thus, by the principle of mathematical induction, $M(n) = 2^n - 1$ for all nonnegative integers n.

The ease with which we solved this recurrence and proved our solution correct is no accident. Recursion, recurrences, and induction are all intimately related. The relationship between recursion and recurrences is reasonably transparent—recurrences give a natural way of analyzing recursive algorithms. Both recursion and recurrences specify the solution to an instance of a problem in terms of solutions to one or more smaller instances. Induction also falls naturally into this paradigm in that we are deriving a statement

$p(n)$ from statements $p(n')$ for $n' < n$. In fact, we saw at the end of Section 4.1 that proof by induction can be thought of as proof by recursion. Thus, we really have three variations on the same theme.

We also observe, more concretely, that the mathematical correctness of solutions to recurrences is naturally proved via induction. Also, the correctness of a recurrence that describes the number of steps needed to solve a recursive problem is also naturally proved by induction. The recurrence or recursive structure of the problem makes setting up the inductive proof straightforward.

Examples of First-Order Linear Recurrences

Exercise 4.2-3 The empty set ($\emptyset$) is a set with no elements. How many subsets does it have? How many subsets does the one-element set $\{1\}$ have? How many subsets does the two-element set $\{1, 2\}$ have? How many of these subsets contain 2? How many subsets does $\{1, 2, 3\}$ have? How many contain 3? Give a recurrence for the number $S(n)$ of subsets of an n-element set, and prove that your recurrence is correct.

Exercise 4.2-4 When paying off a loan with initial amount A and monthly payment M at an interest rate of p percent, the total amount $T(n)$ of the loan after n months is computed by adding $p/12$ percent to the amount due after $n - 1$ months and then subtracting the monthly payment M. Convert this description into a recurrence for the amount owed after n months.

Exercise 4.2-5 Given the recurrence

$$T(n) = rT(n - 1) + a,$$

where r and a are constants, find a recurrence that expresses $T(n)$ in terms of $T(n - 2)$ instead of $T(n - 1)$. Now find a recurrence that expresses $T(n)$ in terms of $T(n - 3)$ instead of $T(n - 2)$ or $T(n - 1)$. Now find a recurrence that expresses $T(n)$ in terms of $T(n - 4)$ rather than $T(n - 1)$, $T(n - 2)$, or $T(n - 3)$. Based on your work so far, find a general formula for the solution to the recurrence

$$T(n) = rT(n - 1) + a,$$

with $T(0) = b$ and where r and a are constants.

If we construct small examples for Exercise 4.2-3, we see that $\emptyset$ has only one subset, $\{1\}$ has two subsets, $\{1, 2\}$ has four subsets, and $\{1, 2, 3\}$ has eight subsets. These small examples give us a good guess as to what the general formula is, but to prove it, we will need to think recursively. Consider the subsets of $\{1, 2, 3\}$:

$$\begin{array}{cccc} \emptyset & \{1\} & \{2\} & \{1, 2\} \\ \{3\} & \{1, 3\} & \{2, 3\} & \{1, 2, 3\} \end{array}$$

The first four subsets do not contain 3, but the second four do. Furthermore, the first four subsets are exactly the subsets of $\{1, 2\}$, while the second four are the four subsets of $\{1, 2\}$ with 3 added into each one. So, we get a subset of $\{1, 2, 3\}$ either by taking a subset of $\{1, 2\}$ or by adjoining 3 to a subset of $\{1, 2\}$. This suggests that the recurrence for the number of subsets of an n-element set (which we may assume is $\{1, 2, \ldots, n\}$) is

$$S(n) = \begin{cases} 2S(n-1) & \text{if } n \geq 1, \\ 1 & \text{if } n = 0. \end{cases} \tag{4.9}$$

To prove that this recurrence is correct, we note that the subsets of $\{1, 2, \ldots, n\}$ can be partitioned according to whether they contain element n. The subsets of $\{1, 2, \ldots, n\}$ containing element n can be constructed by adjoining the element n to the subsets not containing element n. So, the number of subsets containing element n is the same as the number of subsets not containing element n. The number of subsets not containing element n is simply the number of subsets of an $(n-1)$-element set. Therefore, each block of our partition has size equal to the number of subsets of an $(n-1)$-element set. Thus, by the sum principle, the number of subsets of $\{1, 2, \ldots, n\}$ is twice the number of subsets of $\{1, 2, \ldots, n-1\}$. This proves that $S(n) = 2S(n-1)$ if $n > 0$. We already observed that $\emptyset$ has only one subset (itself), so we have proved the correctness of Recurrence 4.9.

For Exercise 4.2-4, we can algebraically describe what the problem said in words by

$$T(n) = \left(1 + \frac{0.01p}{12}\right) \cdot T(n-1) - M,$$

with $T(0) = A$. Note that we add $0.01p/12$ times the principal to the amount due each month, because $p/12$ percent of a number is $0.01p/12$ times the number.

Iterating a Recurrence

Turning to Exercise 4.2-5, we can substitute the right side of the equation $T(n-1) = rT(n-2) + a$ for $T(n-1)$ in our recurrence and then substitute the similar equations for $T(n-2)$ and $T(n-3)$:

$$\begin{aligned} T(n) &= r\big(rT(n-2) + a\big) + a \\ &= r^2T(n-2) + ra + a \\ &= r^2\big(rT(n-3) + a\big) + ra + a \\ &= r^3T(n-3) + r^2a + ra + a \\ &= r^3\big(rT(n-4) + a\big) + r^2a + ra + a \\ &= r^4T(n-4) + r^3a + r^2a + ra + a. \end{aligned}$$

From this, we can guess that

$$\begin{aligned} T(n) &= r^nT(0) + a\sum_{i=0}^{n-1} r^i \\ &= r^nb + a\sum_{i=0}^{n-1} r^i. \end{aligned} \tag{4.10}$$

The method we used to guess the solution is called **iterating the recurrence** because we repeatedly use the recurrence with smaller and smaller values in place of n. We could instead have written

$$\begin{aligned} T(0) &= b \\ T(1) &= rT(0) + a \\ &= rb + a \\ T(2) &= rT(1) + a \\ &= r(rb + a) + a \\ &= r^2b + ra + a \\ T(3) &= rT(2) + a \\ &= r^3b + r^2a + ra + a, \end{aligned}$$

which leads us to the same guess. Why, then, have we introduced two methods? Having different approaches to solving a problem often yields insights we would not get with just one approach. For example, when we

study recursion trees, we will see how to visualize the process of iterating certain kinds of recurrences to simplify the algebra involved in solving them.

Geometric Series

You may recognize the sum $\sum_{i=0}^{n-1} r^i$ in Equation 4.10. It is called a **finite geometric series with common ratio r**. The sum $\sum_{i=0}^{n-1} ar^i$ is called a **finite geometric series with common ratio r and initial value a.**

To get a closed formula for a finite geometric series when $r \neq 1$, we multiply by r and subtract:

$$\begin{aligned} S &= 1 + r + r^2 + \cdots + r^{n-1} \\ rS &= r + r^2 + \cdots + r^n \\ S - rS &= 1 + 0 + \cdots + 0 - r^n \\ (1-r)S &= 1 - r^n \\ S &= \frac{1-r^n}{1-r}. \end{aligned}$$

Combining the first and last equation above, and using sum notation, we obtain:

$$\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r}. \tag{4.11}$$

This formula lets us rewrite the formula for $T(n)$ in a very nice form.

Theorem 4.1 If $T(n) = rT(n-1) + a$, $T(0) = b$, and $r \neq 1$, then

$$T(n) = r^n b + a\frac{1-r^n}{1-r} \tag{4.12}$$

for all nonnegative integers n.

Proof We prove our formula by induction. Notice that the formula gives

$$T(0) = r^0 b + a\frac{1-r^0}{1-r},$$

which is b. So, the formula is true when $n = 0$. Now assume that $n > 0$ and

$$T(n-1) = r^{n-1} b + a\frac{1-r^{n-1}}{1-r}.$$

Then we have

$$\begin{aligned} T(n) &= rT(n-1) + a \\ &= r\left(r^{n-1}b + a\frac{1-r^{n-1}}{1-r}\right) + a \\ &= r^n b + \frac{ar - ar^n}{1-r} + a \\ &= r^n b + \frac{ar - ar^n + a - ar}{1-r} \\ &= r^n b + a\frac{1-r^n}{1-r}. \end{aligned}$$

Therefore, by the principle of mathematical induction, our formula holds for all integers $n \geq 0$.

We can use Theorem 4.1 as an alternate way to prove Equation 4.11. One possible value for r in Theorem 4.1 is 0. With $r = 0$, our recurrence gives us $T(0) = b$, so we expect Equation 4.12 to give b as well. It is standard for many reasons to define 0^0 to be 1, which is exactly what we need to make Equation 4.12 correct in this special case.

Corollary 4.2 The formula for the sum of a geometric series with $r \neq 1$ is

$$\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r}. \tag{4.13}$$

Proof Define $T(n) = \sum_{i=0}^{n-1} r^i$ for $n > 0$ and $T(0) = 0$. Then $T(n) = rT(n-1) + 1$. Applying Theorem 4.1, with $b = 0$ and $a = 1$, gives us

$$T(n) = \frac{1-r^n}{1-r}.$$

Often, when we see a geometric series, we will only be concerned with expressing the sum in big O notation. In this case, we can show that the sum of a geometric series is at most the largest term times a constant factor, where the constant factor depends on r but not on n. For example, if $|r| < 1$, then the largest term in the sum is 1 and the numerator of $(1 - r^n)/(1 - r)$

is less than 1; so, the quotient is no more than the constant $1/(1-r)$. Thus, the sum of the series is no more than the constant $1/(1-r)$ times 1. In other words, the sum of the series is $O(1)$.

Lemma 4.3

Let r be a quantity whose value is independent of n and not equal to 1. Let $t(n)$ be the largest term of the geometric series

$$\sum_{i=0}^{n-1} r^i.$$

Then the value of the geometric series is $O(t(n))$.

Proof It is straightforward to see that we may limit ourselves to proving the lemma for $r > 0$. We consider two cases, depending on whether $r > 1$ or $r < 1$. If $r > 1$, then

$$\begin{aligned}\sum_{i=0}^{n-1} r^i &= \frac{1-r^n}{1-r} \\ &= \frac{r^n - 1}{r-1} \\ &\leq \frac{r^n}{r-1} \\ &= r^{n-1}\frac{r}{r-1} \\ &= O(r^{n-1}).\end{aligned}$$

On the other hand, if $r < 1$, then the largest term is $r^0 = 1$, and the sum has value

$$\frac{1-r^n}{1-r} < \frac{1}{1-r}.$$

Thus, the sum is $O(1)$, and because $t(n) = 1$, the sum is $O(t(n))$.

In fact, when r is nonnegative, an even stronger statement is true. Recall that we said that for two functions f and g from the real numbers to the real numbers, $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$.

Theorem 4.4

Let r be a nonnegative quantity whose value is independent of n and not equal to 1. Let $t(n)$ be the largest term of the geometric series

$$\sum_{i=0}^{n-1} r^i.$$

Then the value of the geometric series is $\Theta(t(n))$.

Proof By Lemma 4.3, we need only show that

$$t(n) = O\left(\frac{r^n - 1}{r - 1}\right).$$

Because all r^i are nonnegative, the sum $\sum_{i=0}^{n-1} r^i$ is at least as large as any of its summands. But $t(n)$ is one of these summands, so

$$t(n) = O\left(\frac{r^n - 1}{r - 1}\right).$$

Note from the proofs that $t(n)$ and the constant in the big O and big Θ upper bounds depend on r. We will use this theorem in subsequent sections.

First-Order Linear Recurrences

A recurrence of the form $T(n) = f(n)T(n-1) + g(n)$ is called a **first-order linear recurrence**. When $f(n)$ is a constant, such as r, the general solution is almost as easy to write as in Theorem 4.1. Iterating the recurrence gives us

$$\begin{aligned}
T(n) &= rT(n-1) + g(n) \\
&= r\big(rT(n-2) + g(n-1)\big) + g(n) \\
&= r^2T(n-2) + rg(n-1) + g(n) \\
&= r^2\big(rT(n-3) + g(n-2)\big) + rg(n-1) + g(n) \\
&= r^3T(n-3) + r^2g(n-2) + rg(n-1) + g(n) \\
&= r^3\big(rT(n-4) + g(n-3)\big) + r^2g(n-2) + rg(n-1) + g(n)
\end{aligned}$$

$$= r^4 T(n-4) + r^3 g(n-3) + r^2 g(n-2) + r g(n-1) + g(n)$$

$$\vdots$$

$$= r^n T(0) + \sum_{i=0}^{n-1} r^i g(n-i).$$

This calculation suggests our next theorem.

Theorem 4.5

For any positive constants a and r and any function g defined on the nonnegative integers, the solution to the first-order linear recurrence

$$T(n) = \begin{cases} rT(n-1) + g(n) & \text{if } n > 0, \\ a & \text{if } n = 0, \end{cases}$$

is

$$T(n) = r^n a + \sum_{i=1}^{n} r^{n-i} g(i). \tag{4.14}$$

Proof Let's prove this by induction.

Because the sum $\sum_{i=1}^{n} r^{n-i} g(i)$ in Equation 4.14 has no terms when $n = 0$, the formula gives $T(0) = a$ and, so, is valid[5] when $n = 0$. We now assume that n is positive and $T(n-1) = r^{n-1}a + \sum_{i=1}^{n-1} r^{(n-1)-i} g(i)$. Using the definition of the recurrence and the inductive hypothesis, we get that

$$\begin{aligned} T(n) &= rT(n-1) + g(n) \\ &= r\left(r^{n-1}a + \sum_{i=1}^{n-1} r^{(n-1)-i} g(i)\right) + g(n) \\ &= r^n a + \sum_{i=1}^{n-1} r^{(n-1)+1-i} g(i) + g(n) \\ &= r^n a + \sum_{i=1}^{n-1} r^{n-i} g(i) + g(n) \\ &= r^n a + \sum_{i=1}^{n} r^{n-i} g(i) \ . \end{aligned}$$

[5]Part of the definition of *summation notation* is that we assign 0 to a summation that has no terms because the value of the summation index below the summation sign is larger than the value of the summation index above the sign.

Therefore, by the principle of mathematical induction, the solution to

$$T(n) = \begin{cases} rT(n-1) + g(n) & \text{if } n > 0, \\ a & \text{if } n = 0, \end{cases}$$

is given by Equation 4.14 for all nonnegative integers n.

The formula in Theorem 4.5 is a little less easy to use than that in Theorem 4.1 because it gives us a sum to compute. Fortunately, for a number of commonly occurring functions g, the sum $\sum_{i=1}^{n} r^{n-i} g(i)$ is not too hard to compute.

Exercise 4.2-6 Solve the recurrence $T(n) = 4T(n-1) + 2^n$, with $T(0) = 6$.

Exercise 4.2-7 Solve the recurrence $T(n) = 3T(n-1) + n$, with $T(0) = 10$.

For Exercise 4.2-6, we can use Equation 4.14 to write

$$\begin{aligned} T(n) &= 6 \cdot 4^n + \sum_{i=1}^{n} 4^{n-i} \cdot 2^i \\ &= 6 \cdot 4^n + 4^n \sum_{i=1}^{n} 4^{-i} \cdot 2^i \\ &= 6 \cdot 4^n + 4^n \sum_{i=1}^{n} \left(\frac{1}{2}\right)^i \\ &= 6 \cdot 4^n + 4^n \cdot \frac{1}{2} \cdot \sum_{i=0}^{n-1} \left(\frac{1}{2}\right)^i \\ &= 6 \cdot 4^n + \left(1 - \left(\frac{1}{2}\right)^n\right) \cdot 4^n \\ &= 7 \cdot 4^n - 2^n. \end{aligned}$$

For Exercise 4.2-7, we begin in the same way and quickly face a bit of a surprise. Using Equation 4.14, we write

$$\begin{aligned} T(n) &= 10 \cdot 3^n + \sum_{i=1}^{n} 3^{n-i} \cdot i \\ &= 10 \cdot 3^n + 3^n \sum_{i=1}^{n} i 3^{-i} \end{aligned}$$

$$= 10 \cdot 3^n + 3^n \sum_{i=1}^{n} i \left(\frac{1}{3}\right)^i. \tag{4.15}$$

Now we are faced with a sum that you may not recognize, a sum that has the form

$$\sum_{i=1}^{n} i x^i = x \sum_{i=1}^{n} i x^{i-1},$$

with $x = 1/3$. However, by writing it in this form, we can use calculus to recognize it as x times a derivative. In particular, using the fact that $0x^0 = 0$, we can write

$$\sum_{i=1}^{n} i x^i = x \sum_{i=0}^{n} i x^{i-1} = x \frac{d}{dx} \sum_{i=0}^{n} x^i = x \frac{d}{dx} \left(\frac{1 - x^{n+1}}{1 - x}\right).$$

Using the formula from calculus for the derivative of a quotient, we may write

$$\begin{aligned} x \frac{d}{dx} \left(\frac{1 - x^{n+1}}{1 - x}\right) &= x \frac{(1 - x)\big(-(n+1)x^n\big) - (1 - x^{n+1})(-1)}{(1 - x)^2} \\ &= \frac{nx^{n+2} - (n+1)x^{n+1} + x}{(1 - x)^2}. \end{aligned}$$

Connecting our first and last equations, we get

$$\sum_{i=1}^{n} i x^i = \frac{nx^{n+2} - (n+1)x^{n+1} + x}{(1 - x)^2}. \tag{4.16}$$

Substituting $x = 1/3$ and simplifying gives us

$$\sum_{i=1}^{n} i \left(\frac{1}{3}\right)^i = -\frac{3}{2}(n+1)\left(\frac{1}{3}\right)^{n+1} - \frac{3}{4}\left(\frac{1}{3}\right)^{n+1} + \frac{3}{4}.$$

Substituting this into Equation 4.15 gives us

$$\begin{aligned} T(n) &= 10 \cdot 3^n + 3^n \left(-\frac{3}{2}(n+1)\left(\frac{1}{3}\right)^{n+1} - \frac{3}{4}\left(\frac{1}{3}\right)^{n+1} + \frac{3}{4}\right) \\ &= 10 \cdot 3^n - \frac{n+1}{2} - \frac{1}{4} + \frac{3^{n+1}}{4} \\ &= \frac{43}{4} 3^n - \frac{n+1}{2} - \frac{1}{4}. \end{aligned}$$

The sum that arises in this exercise occurs so often that we give its formula as a theorem. Because the formula is so complicated, we prefer deriving it when we need it rather than memorizing it.[6]

Theorem 4.6 For any real number $x \neq 1$,

$$\sum_{i=1}^{n} ix^i = \frac{nx^{n+2} - (n+1)x^{n+1} + x}{(1-x)^2}.$$

Proof The proof for this theorem was given before the statement of the theorem.

Important Concepts, Formulas, and Theorems

1. *Recurrence equation or recurrence.* A *recurrence equation* for a function defined on the set of integers greater than or equal to some number b is one that tells us how to compute the nth value of a function from the $(n-1)$st value or how to compute the nth value from some or all the first $n-1$ values.
2. *Initial condition.* To specify completely a function on the basis of a recurrence, we have to give enough information about the function to get started. This information is called the *initial condition* (or the initial conditions) for the recurrence.
3. *First-order linear recurrence.* A recurrence $T(n) = f(n)T(n-1) + g(n)$ is called a *first-order linear recurrence.*
4. *Constant coefficient recurrence.* A recurrence in which $T(n)$ is expressed in terms of a sum of constant multiples of $T(k)$ for certain values $k < n$ (and perhaps another function of n) is called a *constant coefficient recurrence.*
5. *Solution to a first-order constant coefficient linear recurrence.* If $T(n) = rT(n-1) + a$, $T(0) = b$, and $r \neq 1$, then

$$T(n) = r^n b + a\frac{1-r^n}{1-r}$$

for all nonnegative integers n.

[6]The derivation consists of recognizing the left side of the formula as x times a derivative of a geometric series, using the quotient rule for this derivative, and substituting.

6. *Finite geometric series.* A *finite geometric series with common ratio* r is a sum of the form $\sum_{i=0}^{n-1} r^i$. The formula for the sum of a geometric series with $r \neq 1$ is

$$\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r}.$$

7. *Big Θ bounds on the sum of a geometric series.* Let r be a nonnegative quantity whose value is independent of n and not equal to 1. Let $t(n)$ be the largest term of the geometric series

$$\sum_{i=0}^{n-1} r^i.$$

Then the value of the geometric series is $\Theta\big(t(n)\big)$.

8. *Solution to a first-order linear recurrence.* For any positive constants a and r and any function g defined on the nonnegative integers, the solution to the first-order linear recurrence

$$T(n) = \begin{cases} rT(n-1) + g(n) & \text{if } n > 0, \\ a & \text{if } n = 0, \end{cases}$$

is

$$T(n) = r^n a + \sum_{i=1}^{n} r^{n-i} g(i).$$

9. *Iterating a recurrence.* We are *iterating a recurrence* when we guess its solution by
   a. using the equation that expresses $T(n)$ in terms of $T(k)$ for k smaller than n to reexpress $T(n)$ in terms of $T(k)$ for k smaller than $n-1$,
   b. reexpressing $T(n)$ in terms of $T(k)$ for k smaller than $n-2$, and
   c. repeating this procedure until we can guess the formula for the sum.

10. *An important sum.* For any real number $x \neq 1$,

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

The derivation of this formula consists of recognizing the left side of the formula as x times a derivative of a geometric series, using the quotient rule for this derivative, and substituting.

Problems

All problems with blue boxes have an answer or hint available at the end of the book.

1. Prove Equation 4.13 directly by induction. (Recall that $r \neq 1$.)
2. Prove Equation 4.16 directly by induction. (Assume $x \neq 1$.)
3. Solve the recurrence $M(n) = 2M(n-1) + 2$, with a base case of $M(1) = 1$. How does it differ from the solution to Recurrence 4.8?
4. Solve the recurrence $M(n) = 3M(n-1) + 1$, with a base case of $M(1) = 1$. How does it differ from the solution to Recurrence 4.8?
5. Solve the recurrence $M(n) = M(n-1) + 2$, with a base case of $M(1) = 1$. How does it differ from the solution to Recurrence 4.8?
6. There are m functions from a one-element set to the set $\{1, 2, \ldots, m\}$. How many functions are there from a two-element set to $\{1, 2, \ldots, m\}$? From a three-element set? Give a recurrence for the number $T(n)$ of functions from an n-element set to $\{1, 2, \ldots, m\}$. Solve the recurrence.
7. Solve the recurrence derived in Exercise 4.2-4.
8. At the end of each year, a state fish hatchery puts 2000 fish into a lake. The number of fish in the lake at the beginning of the year doubles by the end of the year due to reproduction. Give a recurrence for the number of fish in the lake after n years, and solve the recurrence.
9. Consider the recurrence $T(n) = 3T(n-1) + 1$, with the initial condition $T(0) = 2$. You could write the solution from Theorem 4.1. Instead of using the theorem, try to guess the solution from the first four values of $T(n)$ and then try to guess the solution by iterating the recurrence four times.
10. What sort of big Θ bound can you give on the value of a geometric series $1 + r + r^2 + \cdots + r^n$, with common ratio $r = 1$?
11. Solve the recurrence $T(n) = 2T(n-1) + n2^n$, with the initial condition $T(0) = 1$.
12. Solve the recurrence $T(n) = 2T(n-1) + n^3 2^n$, with the initial condition $T(0) = 2$.

13. Solve the recurrence $T(n) = 2T(n-1) + 3^n$, with $T(0) = 1$.

14. Solve the recurrence $T(n) = rT(n-1) + r^n$, with $T(0) = 1$.

15. Solve the recurrence $T(n) = rT(n-1) + r^{2n}$, with $T(0) = 1$. (Assume that $r \neq 1$.)

16. Solve the recurrence $T(n) = rT(n-1) + s^n$, with $T(0) = 1$. (Assume that $r \neq s$.)

17. Solve the recurrence $T(n) = rT(n-1) + n$, with $T(0) = 1$. (Assume that $r \neq 1$.)

18. The Fibonacci numbers are defined by the recurrence

$$T(n) = \begin{cases} T(n-1) + T(n-2) & \text{if } n > 0, \\ 1 & \text{if } n = 0 \text{ or } n = 1. \end{cases}$$

    a. Write the first 10 Fibonacci numbers, starting with $T(0)$.

    b. Show that $\left((1+\sqrt{5})/2\right)^n$ and $\left((1-\sqrt{5})/2\right)^n$ are solutions to the equation $F(n) = F(n-1) + F(n-2)$.

    c. Why is

$$c_1\left(\frac{1+\sqrt{5}}{2}\right)^n + c_2\left(\frac{1-\sqrt{5}}{2}\right)^n$$

    a solution to the equation $F(n) = F(n-1) + F(n-2)$ for any real numbers c_1 and c_2?

    d. Find constants c_1 and c_2 such that the Fibonacci numbers are given by

$$F(n) = c_1\left(\frac{1+\sqrt{5}}{2}\right)^n + c_2\left(\frac{1-\sqrt{5}}{2}\right)^n.$$

19. Solve the sum in Theorem 4.6 by using the "multiply by x and subtract" approach used to derive Equation 4.11.

4.3 GROWTH RATES OF SOLUTIONS TO RECURRENCES

Divide and Conquer Algorithms

One of the most basic and powerful algorithmic techniques is **divide and conquer**. Consider, for example, the binary search algorithm, which we will describe in the context of guessing a number between 1 and 100. Suppose

someone picks a number between 1 and 100 and allows you to ask questions of the form "Is the number greater than k?" or "Is the number equal to k?" where k is an integer you choose. Your goal is to ask as few questions as possible to get a "yes" to a question of the form "Is the number equal to k?" Why should your first question be, "Is the number greater than 50?" After asking if the number is bigger than 50, you have learned either that the number is between 1 and 50 or that the number is between 51 and 100. In either case, you have reduced your problem to one in which the range is only half as big. Thus, you have *divided* the problem into a problem that is only half as big, and you can now (recursively) *conquer* this remaining problem. (If you ask any other question, the size of one of the possible ranges of values you could end up with would be more than half the size of the original problem.) If you continue in this fashion, always cutting the problem size in half, you will reduce the problem size to 1 fairly quickly, and then you will know what the number is. Of course, if we started with a number in the range from 1 to 128, it would be easier to cut the problem size exactly in half each time, but the question doesn't sound quite so plausible then. Thus, to analyze the problem, we will assume someone asks you to figure out a number between 0 and n, where n is a power of 2.

Exercise 4.3-1 Let $T(n)$ be the number of questions in a binary search on the range of numbers between 1 and n. Assuming that n is a power of 2, give a recurrence for $T(n)$.

For Exercise 4.3-1, we get

$$T(n) = \begin{cases} T(n/2) + 1 & \text{if } n \geq 2 \\ 1 & \text{if } n = 1. \end{cases} \tag{4.17}$$

That is, the number of questions needed to carry out binary search on n items is equal to one step (the first question) plus the time to perform binary search on the remaining $n/2$ items. Note that the base case is $T(1) = 1$ because we have to ask a question of the form "Is the number k?" when we have reduced the range of possible values to 1.

What we are really interested in is how much time it takes to use binary search in a computer program that looks for an item in an ordered list. While the number of questions gives us a feel for the amount of time, processing each question may take several steps in our computer program. The exact amount of time these steps take might depend on some factors over which we have little control, such as where portions of the list are stored. Also, we may have to deal with lists with lengths that are not a

power of 2. Thus, a more realistic description of the maximum time needed would be

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + C_1 & \text{if } n \geq 2, \\ C_2 & \text{if } n = 1, \end{cases} \tag{4.18}$$

where C_1 and C_2 are constants.

Note that $\lceil x \rceil$ stands for the smallest integer larger than or equal to x, whereas $\lfloor x \rfloor$ stands for the largest integer less than or equal to x. It turns out that the solution to Recurrences 4.17 and 4.18 are roughly the same, in a sense that should become clear later. For now, let's not worry about floors and ceilings and the distinction between things that take one unit of time and things that take no more than some constant amount of time.

Instead, let's turn to **merge sort**, another example of a divide-and-conquer algorithm. In this algorithm, we wish to sort a list of n items. Assume that the data are stored in Positions 1 through n of an array A and that n is a power of 2. If the list has only one element, we don't need to do anything to sort it. Otherwise, to sort the list, we divide A into the portions from 1 to $n/2$ and from $n/2 + 1$ to n. We recursively sort the first half, we recursively sort the second half, and then we merge the two sorted "half lists" into one sorted list. (We saw examples of one way to merge two lists in the beginning of Section 3.1.) Merge sort can be described in pseudocode as follows:

```
MergeSort(A,low,high)

// This algorithm sorts the portion of list A from
// location low to location high.
if (low == high)
     return
else
     mid = ⌊(low + high)/2⌋
     MergeSort(A,low,mid)
     MergeSort(A,mid+1,high)
     Merge the sorted lists from the previous two steps
     return
```

More details on merge sort can be found in almost any algorithms textbook. The base case `(low == high)` takes one step. The other case executes one step, makes two recursive calls on problems of size $n/2$, and then executes the merge instruction, which can be done in n steps.

Thus, we obtain the following recurrence for the running time of merge sort:

$$T(n) = \begin{cases} 2T(n/2) + n & \text{if } n > 1, \\ 1 & \text{if } n = 1. \end{cases} \tag{4.19}$$

Recurrences such as this one can be understood via the idea of a recursion tree, which we introduce next. This concept allows us to analyze recurrences that arise in divide-and-conquer algorithms, as well as those that arise in other recursive situations, such as the Tower of Hanoi.

Recursion Trees

A recursion tree for a recurrence is a visual and conceptual representation of the process of iterating the recurrence. We use several examples to introduce the idea of a recursion tree. To understand recursion trees, it is helpful to have an "algorithmic" interpretation of a recurrence. For example, ignoring for a moment the base case, we can interpret the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + n \tag{4.20}$$

as, "To solve a problem of size n, we must solve two problems of size $n/2$ and do n units of additional work." Similarly, we can interpret

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

as, "To solve a problem of size n, we must solve one problem of size $n/4$ and do n^2 units of additional work." We can also interpret the recurrence

$$T(n) = 3T(n-1) + n$$

as, "To solve a problem of size n, we must solve three subproblems of size $n-1$ and do n additional units of work."

In Figure 4.4, we draw the beginning of the recursion tree diagram for Recurrence 4.20. For now, assume n is a power of 2. We draw the diagram in levels, each level representing a level of recursion. Equivalently, each level of the diagram represents a level of iteration of the recurrence. A level of a recursion tree diagram has five parts: two on the left, one in the middle, and two on the right. On the left, we keep track of the problem size and the number of problems; in the middle, we draw the tree; and on the right, we keep track of the work done per problem and the total

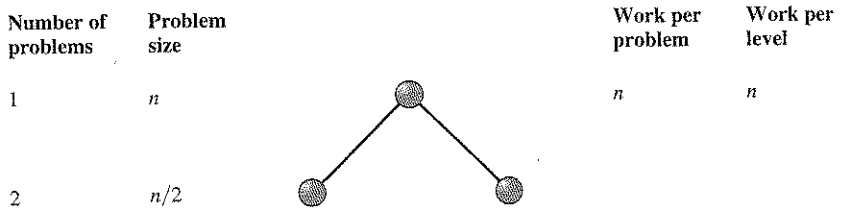


Figure 4.4: The initial stage of drawing a recursion tree diagram

amount of work done on the current level. So, to begin the recursion tree diagram for Recurrence 4.20, we show, in Level 0 on the left, that we have one problem of size n. Then, by drawing a root vertex with two edges leaving it, we show in the middle that we are splitting our problem into two problems. We note on the right that we do n units of work in addition to whatever is done on the two new problems we created. Because there is only one problem on this level, the total work done on this level is n units of work. In the next level, we draw two vertices in the middle, representing the two problems into which we split our main problem, and we show on the left that we have two problems of size $n/2$.

Notice how the recurrence is reflected in Levels 0 and 1 of the recursion tree. The top vertex of the tree represents $T(n)$. On the next level, we have two problems of size $n/2$, representing the recursive term $2T(n/2)$ of our recurrence. After we solve these two problems, we return to Level 0 of the tree and do n additional units of work for the nonrecursive term of the recurrence.

Now we continue to draw the tree in the same manner. Filling in the rest of Level 1 (which is the second level because the first is Level 0) and adding a few more levels, we get Figure 4.5.

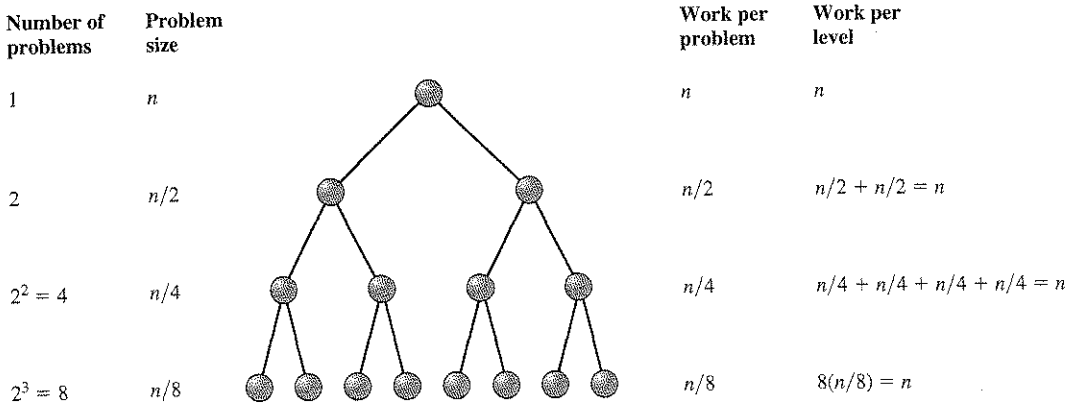


Figure 4.5: Four levels of a recursion tree diagram

Let us summarize what the diagram tells us so far. At Level 0 (the top level), n units of work are done. We see that at each succeeding level, we halve the problem size and double the number of subproblems. We also see that at Level 1, each of the two subproblems requires $n/2$ units of additional work; thus, a total of n units of additional work are done. Similarly, Level 2 has four subproblems of size $n/4$; thus, $4(n/4) = n$ units of additional work are done. Notice that to compute the total work done on a level, we add the amount of work done on each subproblem. When the problems all have the same size, as they do here, this is equivalent to multiplying the number of subproblems by the amount of additional work per subproblem.

To see how iteration of the recurrence is reflected in the diagram, we iterate the recurrence once to obtain

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n \\ &= 4T\left(\frac{n}{4}\right) + n + n \\ &= 4T\left(\frac{n}{4}\right) + 2n. \end{aligned}$$

If we examine Levels 0, 1, and 2 in Figure 4.5, we see that at Level 2 we have four vertices, which represent four problems, each of size $n/4$. This corresponds to the recursive term that we obtained after iterating the recurrence. However, after we solve these problems, we return to Level 1, where we do $n/2$ additional units of work twice, and to Level 0, where we do another n additional units of work. In this way, each time we add a level to the tree, we are showing the result of one more iteration of the recurrence.

We now have enough information to describe the recursion tree diagram in general. To do this, we need to determine four things for each level:

- the number of subproblems
- the size of each subproblem
- the amount of work done per subproblem
- the total work done at that level

Once we know, for each level, the total work done at that level, we can sum over all levels to obtain the total overall work. For this purpose, we also need to figure out how many levels there are in the recursion tree.

We see that for this problem, at Level i, we have 2^i subproblems of size $n/2^i$. Furthermore, because a problem of size 2^i requires 2^i units of additional work, there are $(2^i)\left(n/(2^i)\right) = n$ units of work done per level. To figure

out how many levels there are in the tree, we notice that at each level, the problem size is cut in half, and the tree stops when the problem size is 1. Therefore, there are $\log_2 n + 1$ levels of the tree, because we start with the top level and cut the problem size in half $\log_2 n$ times.[7] We can thus visualize the whole tree in Figure 4.6.

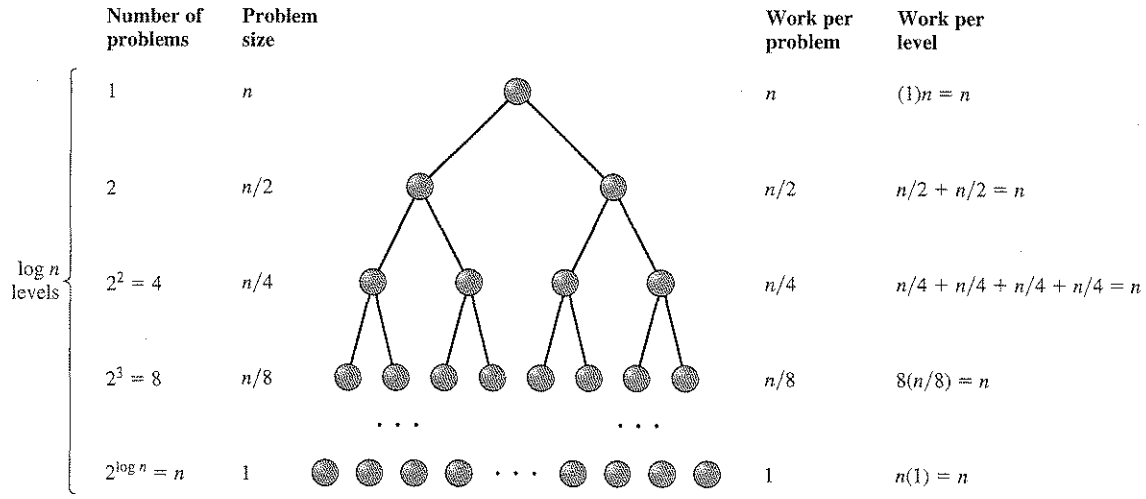


Figure 4.6: A finished recursion tree diagram

The computation of the work done at the bottom level is different from the other levels. In the other levels, the work is described by the recursive equation of the recurrence. At the bottom level, the work comes from the base case. Thus, we must compute the number of problems of size 1 (for this recurrence, the base case is $n = 1$) and then multiply this value by $T(1) = 1$. In the recursion tree in Figure 4.6, the number of nodes at the bottom level is $2^{\log_2 n} = n$. Because $T(1) = 1$, we do n units of work at the bottom level of the tree. But if we had chosen to say that $T(1)$ was some constant c other than 1, the work done at the bottom level would have been cn. We emphasize that the correct value of work per problem at the bottom level always comes from the base case.

The bottom level of the tree represents the final stage of iterating the recurrence. We have seen that at this level, we have n problems, each requiring work $T(1) = 1$, giving us total work n for the level. After we solve the problems represented by the bottom level, we have to do all the additional work from all the earlier levels. For this reason, we sum the work done at

[7] To simplify notation for the remainder of the book, if we omit the base of a logarithm, it should be assumed to be base 2.

all the levels of the tree to get the total work done. *Iteration of the recurrence shows that the solution to the recurrence is the sum of all the work done at all the levels of the recursion tree.*

The important thing is that we now know how much work is done at each level. Once we know this, we can sum the total amount of work done over all the levels, giving us the solution to our recurrence. In this case, there are $\log_2 n + 1$ levels; at each level, the amount of work we do is n units. Thus, we conclude that the total amount of work done to solve the problem described by Recurrence 4.20 is $n(\log_2 n + 1)$.

Because one unit of time will vary from computer to computer, and because some kinds of work might take longer than other kinds, we are usually interested in the big Θ behavior of $T(n)$. For example, we can consider a recurrence that is identical to Recurrence 4.19, except that $T(1) = a$ for some constant a. In this case, $T(n) = an + n \log n$, because an units of work are done at Level 1, and n additional units of work are done at each of the remaining $\log n$ levels. It is still true that $T(n) = \Theta(n \log n)$, because the different base case did not change the solution to the recurrence by more than a constant factor.[8] Although recursion trees can give the exact solutions (such as $T(n) = an + n \log n$) to recurrences, our interest in the big Θ behavior of solutions will usually lead us to use a recursion tree to determine the big Θ or, in complicated cases, the big O behavior of the actual solution to the recurrence. Problem 18 explores whether the value of $T(1)$ actually influences the big Θ behavior of the solution to a recurrence that arises from a divide-and-conquer algorithm.

Let's look at one more recurrence:

$$T(n) = \begin{cases} T(n/2) + n & \text{if } n > 1, \\ 1 & \text{if } n = 1. \end{cases} \tag{4.21}$$

Again, assume n is a power of 2. We can interpret this as follows: To solve a problem of size n, we must solve one problem of size $n/2$ and do n units of additional work. Figure 4.7 shows the recursion tree diagram for this problem. We see that the problem sizes are the same as in the previous tree. The remainder, however, is different. The number of subproblems does not double; rather, it remains at 1 on each level. Consequently, the amount of work halves at each level. Note that there are still $\log n + 1$ levels, because the number of levels is determined by how the problem size changes, not by how many subproblems there are. So, on Level i, we have one problem of size $n/2^i$, for total work of $n/2^i$ units.

[8] More precisely, $n \log n < an + n \log n < (a + 1)n \log n$ for any $a > 0$.

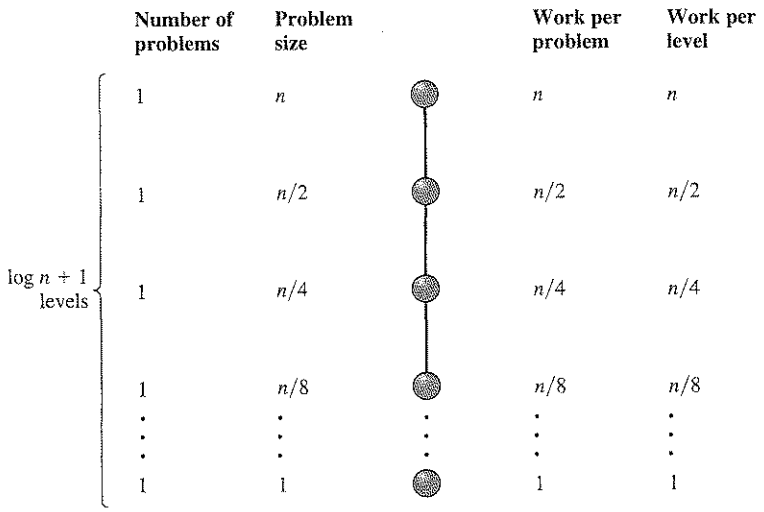


Figure 4.7: A recursion tree diagram for Recurrence 4.21

We now wish to compute how much work is done in solving a problem that gives this recurrence. Note that the additional work done is different on each level, so we have that the total amount of work is

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + 2 + 1 = n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \left(\frac{1}{2}\right)^{\log n}\right),$$

which is n times a geometric series. By Theorem 4.4, the value of a geometric series in which the largest term is 1 is $\Theta(1)$. This implies that the work done is described by $T(n) = \Theta(n)$.

We emphasize that there is exactly one solution to Recurrence 4.21; it is the one we get by using the recurrence to compute $T(2)$ from $T(1)$, then to compute $T(4)$ from $T(2)$, and so on. Here, we have shown that $T(n) = \Theta(n)$. In fact, for the kinds of recurrences we have been examining, once we know $T(1)$, we can compute $T(n)$ for any relevant n by repeatedly using the recurrence. Thus, there is no question that solutions do exist and can, in principle, be computed for any value of n. In most applications, we are not interested in the exact form of the solution; rather, we are interested in a big O upper bound or a big Θ bound on the solution.

Exercise 4.3-2 Use a recursion tree to find a big Θ bound for the solution to the recurrence

$$T(n) = \begin{cases} 3T(n/3) + n & \text{if } n \geq 3, \\ 1 & \text{if } n < 3. \end{cases}$$

Assume that n is a power of 3.

Exercise 4.3-3 Use a recursion tree to solve the recurrence

$$T(n) = \begin{cases} 4T(n/2) + n & \text{if } n \geq 2, \\ 1 & \text{if } n = 1. \end{cases}$$

Assume that n is a power of 2. Convert your solution to a big Θ statement about the behavior of the solution.

Exercise 4.3-4 Can you give a general big Θ bound for solutions to recurrences of the form $T(n) = aT(n/2) + n$ when n is a power of 2? You may have different answers for different values of a.

The recurrence in Exercise 4.3-2 is similar to the merge sort recurrence. One difference is that at each step, we divide into three problems of size $n/3$ rather than two problems of size $n/2$. Thus, we get the picture in Figure 4.8. Another difference is that the number of levels, instead of being $\log_2 n + 1$, is now $\log_3 n + 1$, so that the total work is still $\Theta(n \log n)$ units. (Note that $\log_b n = \Theta(\log_2 n)$ for any $b > 1$.)

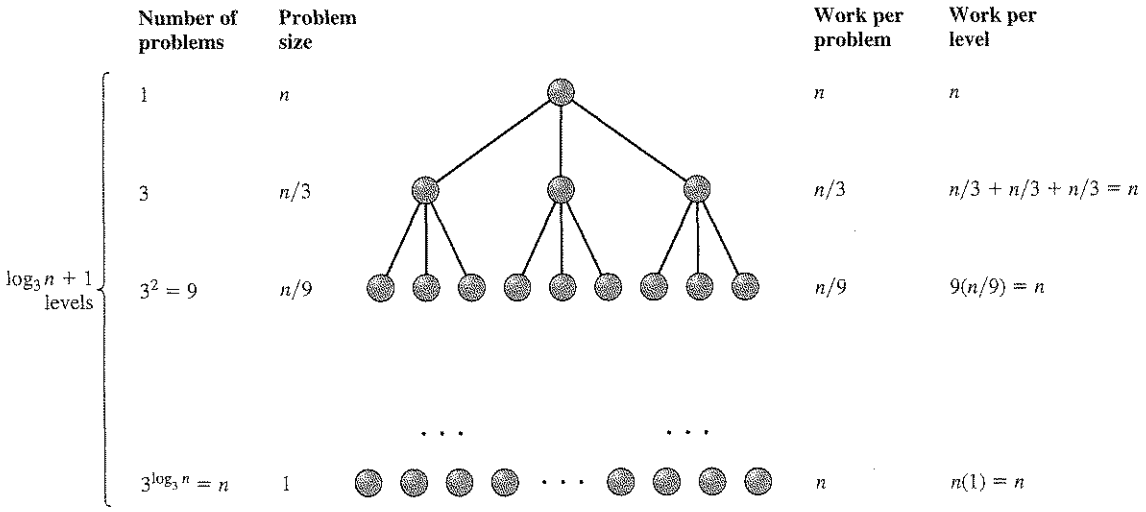


Figure 4.8: The recursion tree diagram for the recurrence in Exercise 4.3-2

Now let's look at the recursion tree for Exercise 4.3-3. A node of size n has four children of size $n/2$, and we get Figure 4.9. Just as in the merge sort tree, there are $\log_2 n + 1$ levels. However, as we pointed out, each node has four children. Thus, Level 0 has 1 node, Level 1 has 4 nodes, Level 2 has 16 nodes, and, in general, Level i has 4^i nodes. On Level i, each node

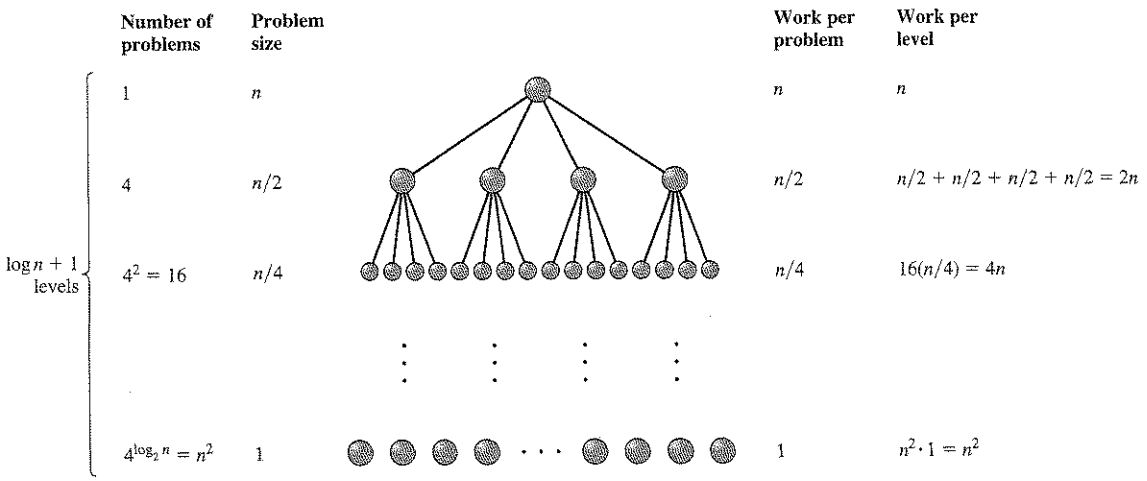


Figure 4.9: The Recursion tree for Exercise 4.3-3

corresponds to a problem of size $n/2^i$ and, hence, requires $n/2^i$ units of additional work. Thus, the total work on Level i is $4^i(n/2^i) = 2^i n$ units. This formula also applies on Level $\log_2 n$ (the bottom level), because there are $4^{\log n} = (2^2)^{\log n} = 2^{2\log n} = (2^{\log n})^2 = n^2 = 2^{\log n} n$ nodes, each requiring $T(1) = 1$ work. Summing over the levels, we get

$$\sum_{i=0}^{\log n} 2^i n = n \sum_{i=0}^{\log n} 2^i.$$

There are many ways to simplify this expression. For example, from our formula for the sum of a geometric series, we get

$$\begin{aligned} T(n) &= n \sum_{i=0}^{\log n} 2^i \\ &= n \frac{1 - 2^{(\log n)+1}}{1 - 2} \\ &= n \frac{1 - 2n}{-1} \\ &= 2n^2 - n \\ &= \Theta(n^2). \end{aligned}$$

More simply, by Theorem 4.4, we have that $T(n) = n \cdot \Theta(2^{\log n}) = \Theta(n^2)$.

Three Different Behaviors

Let's compare the recursion tree diagrams for the recurrences $T(n) = 2T(n/2) + n$, $T(n) = T(n/2) + n$, and $T(n) = 4T(n/2) + n$. Note that all three trees have depth $1 + \log_2 n$, as this is determined by the size of the subproblems relative to the parent problem, and that in each case, the size of each subproblem is half the size of the parent problem. The trees differ, however, in the amount of work done per level. For the first recurrence, the amount of work on each level is the same. In the second, the amount of work done on a level decreases as we go down the tree, with the most work being at the top level. In fact, it decreases geometrically; by Theorem 4.4, the total work done is bounded above and below by a constant multiplied by the work done at the root node. In the third recurrence, the number of nodes per level is growing at a faster rate than the problem size is decreasing, and the level with the largest amount of work is the bottom one. Again, we have a geometric series; and so, by Theorem 4.4, the total work is bounded above and below by a constant multiplied by the amount of work done at the last level.

If you understand these three cases and the differences among them, then you understand the great majority of the recursion trees that arise in algorithms.

So, to answer Exercise 4.3-4, which asks for a general big Θ bound for the solutions to recurrences of the form $T(n) = aT(n/2) + n$, we can conclude the following:

Lemma 4.7

Suppose that we have a recurrence of the form

$$T(n) = aT\left(\frac{n}{2}\right) + n,$$

where a is a positive integer and $T(1)$ is nonnegative. Then we have the following big Θ bounds on the solution:

1. If $a < 2$, then $T(n) = \Theta(n)$.
2. If $a = 2$, then $T(n) = \Theta(n \log n)$.
3. If $a > 2$, then $T(n) = \Theta(n^{\log_2 a})$.

Proof Cases 1 and 2 follow immediately from our earlier observations. We can verify Case 3 as follows: At Level i, we have a^i nodes, each corresponding to a problem of size $n/2^i$. Thus, at Level i, the total amount of work is $a^i(n/2^i) = n(a/2)^i$ units. Summing over the $\log_2 n$ levels, we obtain

$$a^{\log_2 n} T(1) + n \sum_{i=0}^{(\log n)-1} \left(\frac{a}{2}\right)^i.$$

The sum given by the summation sign is a geometric series. Therefore, because $a/2 \neq 1$, the sum will be big Θ of the largest term (see Theorem 4.4). Because $a > 2$, the largest term in this case is clearly the last one, namely, $n(a/2)^{(\log n)-1}$. Applying rules of exponents and logarithms, we get that n times the largest term is

$$\begin{aligned} n\left(\frac{a}{2}\right)^{(\log_2 n)-1} &= \frac{2}{a} \cdot \frac{n \cdot a^{\log n}}{2^{\log n}} \\ &= \frac{2}{a} \cdot \frac{n \cdot a^{\log n}}{n} \\ &= \frac{2}{a} \cdot a^{\log n} \\ &= \frac{2}{a}(2^{\log a})^{\log n} \\ &= \frac{2}{a}(2^{\log n})^{\log a} \\ &= \frac{2}{a} \cdot n^{\log a}. \end{aligned} \tag{4.22}$$

Thus, $T(1)a^{\log n} = T(1)n^{\log a}$. Because $2/a$ and $T(1)$ are both nonnegative, the total work done is $\Theta(n^{\log_2 a})$.

In fact, Lemma 4.7 holds for all positive real numbers a; we can iterate the recurrence to see this. Because a recursion tree diagram is a way to visualize iterating the recurrence when a is an integer, iteration is the natural thing to try when a is not an integer.

Notice that in the last two equalities of the computation made in Equation 4.22, we showed that $a^{\log n} = n^{\log a}$. This fact is useful, so we state it (in slightly more generality) as a corollary to the proof.

Corollary 4.8 For any base b, we have $a^{\log_b n} = n^{\log_b a}$.

Important Concepts, Formulas, and Theorems

1. *Divide-and-conquer algorithm.* A *divide-and-conquer algorithm* is one that solves a problem by dividing the problem into "subproblems" that are smaller than, but otherwise of the same type as, the original one; recursively solving these subproblems; and then

assembling the solution of these subproblems into a solution of the original one. Although not all problems can be solved by such a strategy, a great many problems of interest in computer science can be.

2. *Merge sort.* In *merge sort,* we sort a list of items that have some underlying order by dividing the list in half, sorting the first half (by recursively using merge sort), sorting the second half (by recursively using merge sort), and then merging the two sorted lists. For a list of length 1, merge sort returns the same list.
3. *Recursion tree diagram.* We draw a *recursion tree diagram* for a recurrence by levels, with each level representing a level of recursion. A level of a recursion tree diagram has five parts: two on the left, one in the middle, and two on the right. On the left, we keep track of the problem size and the number of problems; in the middle, we draw the tree; and on the right, we keep track of the work done per problem and the total amount of work done on the current level. The tree has a vertex representing the initial problem and one representing each subproblem to be solved. The work done per problem at each level, other than the bottom, is given by the "additional work" part of the recurrence. The work done at the bottom level is determined by the base case of the recurrence, as is the size of a problem at the bottom level. The solution to the recurrence is the sum of the total work done at each level of the recursion tree.
4. *The base level of a recursion tree.* The amount of work done on the lowest level in a recursion tree is the number of nodes times the value given by the initial condition; it is not determined by attempting to make a computation of "additional work" done at the lowest level.
5. *Bases for logarithms.* We use $\log n$ as an alternate notation for $\log_2 n$. A fundamental fact about logarithms is that $\log_b n = \Theta(\log_2 n)$ for any real number $b > 1$.
6. *An important fact about logarithms.* For any $b > 0$, we have $a^{\log_b n} = n^{\log_b a}$.
7. *Three behaviors of solutions.* The solution to a recurrence of the form $T(n) = aT(n/2) + n$ behaves in one of the following ways:
   a. If $a < 2$, then $T(n) = \Theta(n)$.
   b. If $a = 2$, then $T(n) = \Theta(n \log n)$.
   c. If $a > 2$, then $T(n) = \Theta(n^{\log_2 a})$.

Problems

All problems with blue boxes have an answer or hint available at the end of the book.

1. Draw a recursion tree diagram for

$$T(n) = \begin{cases} 4T(n/4) + n & \text{if } n \geq 2, \\ 1 & \text{if } n = 1. \end{cases}$$

Use it to find the exact solution to the recurrence. Assume n is a power of 4.

2. Draw a recursion tree diagram for

$$T(n) = \begin{cases} 2T(n/2) + 2n & \text{if } n \geq 2, \\ 2 & \text{if } n = 1. \end{cases}$$

Use it to find the exact solution to the recurrence. Assume n is a power of 2.

3. Draw a recursion tree diagram for

$$T(n) = \begin{cases} 9T(n/3) + n & \text{if } n > 1, \\ 1 & \text{if } n = 1. \end{cases}$$

Use it to find a big Θ bound on the solution to the recurrence. Assume n is a power of 3.

4. Draw a recursion tree diagram for

$$T(n) = \begin{cases} T(n/4) + n & \text{if } n \geq 2, \\ 1 & \text{if } n = 1. \end{cases}$$

Use it to find a big Θ bound to the solution to the recurrence. Assume n is a power of 4.

5. Draw a recursion tree diagram for

$$T(n) = \begin{cases} 2T(n/4) + n & \text{if } n \geq 2, \\ 1 & \text{if } n = 1. \end{cases}$$

Use it to find a big Θ bound on the solution to the recurrence. Assume n is a power of 4.

6. Draw a recursion tree diagram for

$$T(n) = \begin{cases} 4T(n/2) + n^2 & \text{if } n \geq 2, \\ 3 & \text{if } n = 1. \end{cases}$$

Use it to find the exact solution to the recurrence. Assume n is a power of 2.

7. Draw a recursion tree diagram for

$$T(n) = \begin{cases} 3T(n/3) + 1 & \text{if } n \geq 2, \\ 2 & \text{if } n = 1. \end{cases}$$

Use it to find the exact solution to the recurrence. Assume n is a power of 3.

8. Draw a recursion tree diagram for $T(n) = T(n/3) + 1$, with $T(1) = 3$. Use it to find an exact solution to the recurrence.

9. Draw recursion trees, and use them to find big Θ bounds on the solutions to the following recurrences. For each, assume that $T(1) = 1$ and that n is a power of the appropriate integer.
   a. $T(n) = 8T(n/2) + n$
   b. $T(n) = 8T(n/2) + n^3$
   c. $T(n) = 3T(n/2) + n$
   d. $T(n) = T(n/4) + 1$
   e. $T(n) = 3T(n/3) + n^2$

10. Draw recursion trees and find exact solutions to the following recurrences. For each, assume that $T(1) = 1$ and that n is a power of the appropriate integer.
   a. $T(n) = 8T(n/2) + n$
   b. $T(n) = 8T(n/2) + n^3$
   c. $T(n) = 3T(n/2) + n$
   d. $T(n) = T(n/4) + 1$
   e. $T(n) = 3T(n/3) + n^2$

11. Find the exact solution to Recurrence 4.21.

12. Show that $\log_b n = \Theta(\log_2 n)$ for any constant $b > 1$.

13. Prove Corollary 4.8 by showing that $a^{\log_b n} = n^{\log_b a}$ for any $b > 0$.

14. Recursion trees work, even if the problems do not break up geometrically or if the work per level is not n^c units. Draw recursion trees and find the best big O bounds you can for solutions to the following recurrences. For each, assume that $T(1) = 1$.
    a. $T(n) = T(n-1) + n$
    b. $T(n) = 2T(n-1) + n$
    c. $T(n) = T(\lfloor\sqrt{n}\rfloor) + 1$ (Assume n has the form $n = 2^{2^i}$.)
    d. $T(n) = 2T(n/2) + n \log n$ (Assume n is a power of 2.)

15. In each case in Problem 14, is the big O bound you found a big Θ bound?

16. If $S(n) = aS(n-1) + g(n)$ and $g(n) < c^n$ with $1 \leq c < a$, how fast does $S(n)$ grow (in big Θ terms)?

17. If $S(n) = aS(n-1) + g(n)$ and $g(n) = c^n$ with $0 < a < c$, how fast does $S(n)$ grow in big Θ terms?

18. Suppose you are given recurrences of the form $T(n) = aT(n/b) + g(n)$, with $T(1) = d > 0$ and $g(n) > 0$ for all n, and $S(n) = aS(n/b) + g(n)$, with $S(1) = 0$ (and the same a, b, and $g(n)$). Is there any difference in the big Θ behavior of the solutions to the two recurrences? What does this say about the influence of the initial condition on the big Θ behavior of such recurrences?

4.4 THE MASTER THEOREM

Master Theorem

In Section 4.3, we saw three different kinds of behavior for recurrences of the form

$$T(n) = \begin{cases} aT(n/2) + n & \text{if } n > 1, \\ d & \text{if } n = 1. \end{cases}$$

These behaviors depend on whether $a < 2$, $a = 2$, or $a > 2$. Remember that a is the number of subproblems into which our problem is divided. Dividing by 2 cuts our problem size in half each time. The n term says that after we complete our recursive work, we have n additional units of work to do for a problem of size n. There is no reason that the amount of additional work required by each subproblem needs to be the size of the subproblem. In many applications, it will be something else. In the master theorem that follows, we consider a more general case. Similarly, the sizes of the subproblems don't have to be half the size of the parent problem. We get the following theorem, our first version of the **master theorem**. (In an appendix we prove some stronger forms of this theorem.)

Theorem 4.9 **(Master Theorem, Preliminary Version)** Let a be an integer greater than or equal to 1, and let b be a real number greater than 1. Let c be a positive real number, and d, a nonnegative real number. Given a recurrence of the form

$$T(n) = \begin{cases} aT(n/b) + n^c & \text{if } n > 1, \\ d & \text{if } n = 1, \end{cases}$$

in which n is restricted to be a power of b, we get the following:

1. If $\log_b a < c$, then $T(n) = \Theta(n^c)$.
2. If $\log_b a = c$, then $T(n) = \Theta(n^c \log n)$.
3. If $\log_b a > c$, then $T(n) = \Theta(n^{\log_b a})$.

Proof We will prove the special case $d = 1$; the case for general d is not much more difficult and is dealt with in Problem 6.

Let's think about the recursion tree for this recurrence. There will be $1 + \log_b n$ levels. At each level, the number of subproblems will be multiplied by a; so, the number of subproblems at Level i will be a^i. Each subproblem at Level i is a problem of size n/b^i. A subproblem of size n/b^i requires $(n/b^i)^c$ additional work, and because there are a^i problems on Level i, the total number of units of work on Level i is

$$a^i \left(\frac{n}{b^i}\right)^c = n^c \left(\frac{a^i}{b^{ci}}\right) = n^c \left(\frac{a}{b^c}\right)^i. \tag{4.23}$$

At the bottom level, $n/b^i = 1$ and there are a^i subproblems, each requiring one unit of work, so Equation 4.23 gives the work for the bottom level as well. In Lemma 4.7, the different cases for $c = 1$ occurred when the work per level was decreasing, constant, or increasing. The same analysis applies here. From our formula for work on Level i, we see that the work per level is decreasing, constant, or increasing exactly when $(a/b^c)^i$ is decreasing, constant, or increasing, respectively. These three cases depend on whether (a/b^c) is less than 1, equal to 1, or greater than 1, respectively. Now observe that

$$\begin{aligned} & \left(\tfrac{a}{b^c}\right) = 1 \\ \Leftrightarrow \quad & a = b^c \\ \Leftrightarrow \quad & \log_b a = c \log_b b \\ \Leftrightarrow \quad & \log_b a = c. \end{aligned}$$

This equation shows us where the three cases in the statement of the theorem come from. Now we need to show the bound on $T(n)$ in the different cases. In the next few paragraphs, we will use the following facts (whose proofs are a straightforward application of the definition of logarithms and rules of exponents).

- For any x, y, and z, each greater than 1, we have $x^{\log_y z} = z^{\log_y x}$. (See Corollary 4.8, Problem 13 from Section 4.3, and Problem 7 at the end of this section.)
- For any $y > 0$ and any real number $x > 1$, we have $\log_x y = \Theta(\log_2 y)$. (See Problem 12 from Section 4.3.)

In general, we compute the total work done by summing the expression, given in Equation 4.23, for the work per level over all the levels. This gives

$$\sum_{i=0}^{\log_b n} n^c \left(\frac{a}{b^c}\right)^i = n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i .$$

In Case 1 (part 1 in the statement of the theorem), this is n^c times a geometric series with a common ratio less than 1. We now complete the proof in Case 1 and leave Cases 2 and 3 as exercises. Theorem 4.4 tells us that

$$n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i = \Theta(n^c).$$

This concludes the proof of Case 1.

Exercise 4.4-1 Prove Case 2 of the master theorem.

Exercise 4.4-2 Prove Case 3 of the master theorem.

In Case 2, we have that $a/b^c = 1$, and so

$$\begin{aligned} n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i &= n^c \sum_{i=0}^{\log_b n} 1^i \\ &= n^c(1 + \log_b n) \\ &= \Theta(n^c \log n). \end{aligned}$$

In Case 3, we have that $a/b^c > 1$. So, in the series

$$\sum_{i=0}^{\log_b n} n^c \left(\frac{a}{b^c}\right)^i = n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i,$$

the largest term is the last one. Then by Theorem 4.4, the sum is $\Theta\left(n^c (a/b^c)^{\log_b n}\right)$. But

$$\begin{aligned} n^c \left(\frac{a}{b^c}\right)^{\log_b n} &= n^c \cdot \frac{a^{\log_b n}}{(b^c)^{\log_b n}} \\ &= n^c \cdot \frac{n^{\log_b a}}{n^{\log_b b^c}} \\ &= n^c \cdot \frac{n^{\log_b a}}{n^c} \\ &= n^{\log_b a}. \end{aligned}$$

Thus, the solution is $\Theta(n^{\log_b a})$.

Note that we may assume that a is a real number with $a > 1$ and give a somewhat similar proof of the master theorem (replacing the recursion tree with an iteration of the recurrence), but we do not give the details here.

Solving More General Kinds of Recurrences

Exercise 4.4-3 What can we say about the big Θ behavior of the solution to

$$T(n) = \begin{cases} 2T(n/3) + 4n^{3/2} & \text{if } n > 1, \\ d & \text{if } n = 1, \end{cases}$$

where n can be any nonnegative power of 3?

Exercise 4.4-4 If $f(n) = n\sqrt{n+1}$, what can we say about the big Θ behavior of solutions to

$$S(n) = \begin{cases} 2S(n/3) + f(n) & \text{if } n > 1, \\ d & \text{if } n = 1, \end{cases}$$

where n can be any nonnegative power of 3?

For Exercise 4.4-3, the work done at each level of the tree, except for the bottom level, will be four times the work done by the recurrence

$$T'(n) = \begin{cases} 2T'(n/3) + n^{3/2} & \text{if } n > 1, \\ d & \text{if } n = 1. \end{cases}$$

Thus, the work done by T will be no more than four times the work done by T' but will be larger than the work done by T'. Therefore, $T(n) = \Theta\big(T'(n)\big)$. By the master theorem, because $\log_3 2 < 1 < 3/2$, we have that $T(n) = \Theta(n^{3/2})$.

For Exercise 4.4-4, because $n\sqrt{n+1} > n\sqrt{n} = n^{3/2}$, we have that $S(n)$ is at least as big as the solution to the recurrence

$$T'(n) = \begin{cases} 2T'(n/3) + n^{3/2} & \text{if } n > 1, \\ d & \text{if } n = 1, \end{cases}$$

where n can be any nonnegative power of 3. But the solution to the recurrence for S will be no more than the solution to the recurrence in Exercise 4.4-3 for T, because $n\sqrt{n+1} \leq 4n^{3/2}$ for $n \geq 0$. Because $T(n) = \Theta\big(T'(n)\big)$, we have that $S(n) = \Theta\big(T'(n)\big)$ as well. Thus $S(n) = \Theta(n^{3/2})$.

Extending the Master Theorem

As Exercises 4.4-3 and 4.4-4 suggest, there is a whole range of interesting recurrences that do not fit the preliminary version of the master theorem but are closely related to recurrences that do. These recurrences have the same kind of behavior predicted by our original version of the master theorem. However, the original version of the theorem does not apply to them, just as it does not apply to the recurrences of Exercises 4.4-3 and 4.4-4.

We now state a second version of the master theorem that covers these cases. A still stronger version of the theorem may be found in *Introduction to Algorithms* by Cormen et al. [13]; the version here captures much of the interesting behavior of recurrences that arise from the analysis of algorithms.

Theorem 4.10 **(Master Theorem)** Let a and b be positive real numbers, with $a \geq 1$ and $b > 1$. Let $T(n)$ be defined for integers n that are powers of b by

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{if } n > 1, \\ d & \text{if } n = 1. \end{cases}$$

Then we have the following:

1. If $f(n) = \Theta(n^c)$, where $\log_b a < c$, then $T(n) = \Theta(n^c) = \Theta\big(f(n)\big)$.
2. If $f(n) = \Theta(n^c)$, where $\log_b a = c$, then $T(n) = \Theta(n^c \log n) = \Theta\big(f(n) \log n\big)$.
3. If $f(n) = \Theta(n^c)$, where $\log_b a > c$, then $T(n) = \Theta(n^{\log_b a})$.

Proof We construct a recursion tree or iterate the recurrence. Because we have assumed that $f(n) = \Theta(n^c)$, there are constants c_1 and c_2, independent of the level, so that the work at each level is between $c_1 n^c (a/b^c)^i$ and $c_2 n^c (a/b^c)^i$. From this point on, the proof is largely a translation of the original proof.

Exercise 4.4-5 What does the master theorem tell us about the solutions to the recurrence

$$T(n) = \begin{cases} 3T(n/2) + n\sqrt{n+1} & \text{if } n > 1, \\ 1 & \text{if } n = 1\ ? \end{cases}$$

Our solution to Exercise 4.4-4 showed us that $x\sqrt{x+1} = \Theta(x^{3/2})$. Because $2^{3/2} = \sqrt{2^3} = \sqrt{8} < 3$, we have that $\log_2 3 > 3/2$. Then, by the third conclusion of the master theorem, $T(n) = \Theta(n^{\log_2 3})$.

An appendix is devoted to careful analysis of divide-and-conquer recurrences in which n is not a power of b and $T(n/b)$ is replaced by $T(\lceil n/b \rceil)$. Although the details are somewhat technical, the end result is that the big Θ behavior of such recurrences is the same as the corresponding recurrences for functions defined on powers of b. In particular, the following theorem is a consequence of what we prove.

Theorem 4.11 Let a and b be positive real numbers, with $a \geq 1$ and $b \geq 2$. Let $T(n)$ satisfy the recurrence

$$T(n) = \begin{cases} aT(\lceil n/b \rceil) + f(n) & \text{if } n > 1, \\ d & \text{if } n = 1. \end{cases}$$

Then we have the following:

1. If $f(n) = \Theta(n^c)$, where $\log_b a < c$, then $T(n) = \Theta(n^c) = \Theta\big(f(n)\big)$.
2. If $f(n) = \Theta(n^c)$, where $\log_b a = c$, then $T(n) = \Theta(n^c \log n) = \Theta\big(f(n) \log n\big)$.
3. If $f(n) = \Theta(n^c)$, where $\log_b a > c$, then $T(n) = \Theta(n^{\log_b a})$.

(The condition that $b \geq 2$ can be changed to $b > 1$ with an appropriate change in the base case of the recurrence, but the base case will then depend on b. We do not prove this here.)

Important Concepts, Formulas, and Theorems

1. *Master theorem, preliminary version.* This simplified version of the *master theorem* states: Let a be an integer greater than or equal to 1 and b be a real number greater than 1. Let c be a positive real number and d a nonnegative real number. Given a recurrence of the form

$$T(n) = \begin{cases} aT(n/b) + n^c & \text{if } n > 1, \\ d & \text{if } n = 1, \end{cases}$$

for n a power of b, we have the following:

   a. If $\log_b a < c$, then $T(n) = \Theta(n^c)$.

   b. If $\log_b a = c$, then $T(n) = \Theta(n^c \log n)$.

   c. If $\log_b a > c$, then $T(n) = \Theta(n^{\log_b a})$.

2. *Properties of logarithms.* For any x, y, and z, each greater than 1, we have that $x^{\log_y z} = z^{\log_y x}$. Also, $\log_x y = \Theta(\log_2 y)$ if x is a constant.

3. *Master theorem, final version.* Let a and b be positive real numbers, with $a \geq 1$ and $b \geq 2$. Let $T(n)$ be defined for integers n that are

powers of b by

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{if } n > 1, \\ d & \text{if } n = 1. \end{cases}$$

Then we have the following:

a. If $f(n) = \Theta(n^c)$, where $\log_b a < c$, then $T(n) = \Theta(n^c) = \Theta\big(f(n)\big)$.

b. If $f(n) = \Theta(n^c)$, where $\log_b a = c$, then $T(n) = \Theta(n^c \log n) = \Theta\big(f(n) \log n\big)$.

c. If $f(n) = \Theta(n^c)$, where $\log_b a > c$, then $T(n) = \Theta(n^{\log_b a})$.

A similar result with a base case that depends on b holds when $1 < b < 2$.

4. *A more general master theorem.* Let a and b be positive real numbers with $a \geq 1$ and $b \geq 2$. Let $T(n)$ satisfy the recurrence

$$T(n) = \begin{cases} aT(\lceil n/b \rceil) + f(n) & \text{if } n > 1, \\ d & \text{if } n = 1. \end{cases}$$

Then we have the following:

a. If $f(n) = \Theta(n^c)$, where $\log_b a < c$, then $T(n) = \Theta(n^c) = \Theta\big(f(n)\big)$.

b. If $f(n) = \Theta(n^c)$, where $\log_b a = c$, then $T(n) = \Theta(n^c \log n) = \Theta\big(f(n) \log n\big)$.

c. If $f(n) = \Theta(n^c)$, where $\log_b a > c$, then $T(n) = \Theta(n^{\log_b a})$.

Problems

All problems with blue boxes have an answer or hint available at the end of the book.

1. Use the master theorem to give big Θ bounds on the solutions to the following recurrences. For each, assume that $T(1) = 1$ and that n is a power of the appropriate integer.

a. $T(n) = 8T(n/2) + n$

b. $T(n) = 8T(n/2) + n^3$

c. $T(n) = 3T(n/2) + n$

d. $T(n) = T(n/4) + 1$

e. $T(n) = 3T(n/3) + n^2$

2. Give a big Θ bound on the solution to the recurrence

$$T(n) = \begin{cases} 3T(\lceil n/2 \rceil) + \sqrt{n+3} & \text{if } n > 1, \\ d & \text{if } n = 1. \end{cases}$$

3. Give a big Θ bound on the solution to the recurrence

$$T(n) = \begin{cases} 3T(\lceil n/2 \rceil) + \sqrt{n^3+3} & \text{if } n > 1, \\ d & \text{if } n = 1. \end{cases}$$

4. Give a big Θ bound on the solution to the recurrence

$$T(n) = \begin{cases} 3T(\lceil n/2 \rceil) + \sqrt{n^4+3} & \text{if } n > 1, \\ d & \text{if } n = 1. \end{cases}$$

5. Give a big Θ bound on the solution to the recurrence

$$T(n) = \begin{cases} 2T(\lceil n/2 \rceil) + \sqrt{n^2+3} & \text{if } n > 1, \\ d & \text{if } n = 1. \end{cases}$$

6. Extend the proof of the preliminary version of the master theorem (Theorem 4.9) to the case $T(1) = d$.

7. Prove Corollary 4.8 by showing that for any x, y, and z, each greater than 1, $x^{\log_y z} = z^{\log_y x}$.

4.5 MORE GENERAL KINDS OF RECURRENCES

Recurrence Inequalities

The recurrences we have been working with arise from idealized descriptions of important processes in computer science. For example, in merge sort on a list of n items, we divide the list into two parts of equal size, sort each part, and then merge the two sorted parts. The time it takes to do this is the time it takes to divide the list into two parts, plus the time it takes to sort each part, plus the time it takes to merge the two sorted lists. We don't specify how we are dividing the list or how we are doing the merging. We assume the sorting of smaller lists is done by applying the same method to the smaller lists, unless they have size 1, in which case we do nothing. What we know is that any sensible way of dividing the

list into two parts takes no more than some constant multiple of n time units (and might take no more than constant time if we do it by leaving the list in place and manipulating pointers) and that any sensible algorithm for merging two lists will take no more than some (other) constant multiple of n time units. Thus, we know that if $T(n)$ is the time it takes to apply merge sort to n data items, then there is a constant c (the sum of the two constant multiples we mentioned) such that

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn. \tag{4.24}$$

Thus, rather than leading to recurrence equations, real-world problems often lead us to **recurrence inequalities**, which are inequalities that state that $T(n)$ is less than or equal to some expression involving values of $T(m)$ for $m < n$. (We could also include inequalities with a greater than or equal to sign, but they do not arise in the applications we are studying.) A **solution** to a recurrence inequality is a function T that satisfies the inequality. For simplicity, we will expand what we mean by the word "recurrence" to include either recurrence inequalities or recurrence equations.

In Recurrence 4.24, we are implicitly assuming that T is defined only on positive integer values, and because we said we divided the list into two equal parts each time, our analysis only makes sense if we assume that n is a power of 2.

Note that there are actually infinitely many solutions to Recurrence 4.24. (For example, for any $c' < c$, the unique solution to

$$T(n) = \begin{cases} 2T(n/2) + c'n & \text{if } n \geq 2, \\ k & \text{if } n = 1, \end{cases} \tag{4.25}$$

satisfies Recurrence 4.24 for any constant k.) The idea that Recurrence 4.24 has infinitely many solutions while Recurrence 4.25 has exactly one solution is analogous to the idea that $x - 3 \leq 0$ has infinitely many solutions whereas $x - 3 = 0$ has one solution. There are several ways to show that all the solutions to Recurrence 4.24 satisfy $T(n) = O(n \log n)$. In other words, no matter how we sensibly implement merge sort, we have a $O(n \log n)$ time bound on how long the merge sort process takes.

The Master Theorem for Inequalities

We commented that the unique solution to Recurrence 4.24 is also a solution to Recurrence 4.25. The largest solution to $x - 3 \leq 0$ is 3, which is the unique solution to $x - 3 = 0$. We have a similar phenomenon with recurrences.

Theorem 4.12

Let a and b be real numbers with $a > 0$ and $b > 1$, and let f be a function from nonnegative integer powers of b to the real numbers. Suppose that T is the unique solution to the recurrence

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{if } n \geq 1, \\ k & \text{if } n = 1, \end{cases}$$

defined on nonnegative integral powers n of b, and that S is a solution to

$$S(n) \leq \begin{cases} aS(n/b) + f(n) & \text{if } n > 1, \\ k & \text{if } n = 1. \end{cases}$$

Then $S(n) \leq T(n)$ for all $n \geq 1$.

Proof We are given that $S(1) \leq k = T(1)$. Suppose that for $j < m$ for both powers of b, we have $S(j) \leq T(j)$. Then

$$S(m) \leq S\left(\frac{m}{b}\right) + f(m) \leq T\left(\frac{m}{b}\right) + f(m) = T(m).$$

Thus, by the principle of mathematical induction, $S(n) \leq T(n)$ for all nonnegative integral powers n of b.

Corollary 4.13

(Master Theorem for Recurrence Inequalities) Let a and b be real numbers with $a \geq 1$ and $b > 1$, and let S be a function from nonnegative integer powers of b to the real numbers. If

$$S(n) \leq \begin{cases} aS(n/b) + f(n) & \text{if } n > 1, \\ k & \text{if } n = 1, \end{cases}$$

then the conclusions of the master theorem (Theorem 4.10) hold for S with Θ replaced by O.

Proof Define T by replacing $\leq$ with $=$ and S with T. Then T satisfies the conclusions of the master theorem, and, by Theorem 4.15, $S(n) \leq T(n)$.

This argument tells us immediately that all solutions to Recurrence 4.24 are $O(n \log n)$. Thus, in situations where the function $f(n)$ that tells us the additional work for a problem of size n in a divide-and-conquer algorithm

satisfies one of the three cases of the master theorem, we can analyze recurrence inequalities as easily as we analyze recurrence equations. However, not all realistic recurrences satisfy the hypotheses of the master theorem. For example, if $f(n) = n \log n$, none of the three conditions of the master theorem are satisfied. In this case, we can analyze recurrence inequalities via a recursion tree diagram. The process is virtually identical to our previous use of recursion trees; however, we must keep in mind that on each level we are really computing an upper bound on the work done on that level. We can also use a variant of the method that we used in solving Exercise 4.2-2—guessing an answer (in this case an upper bound) and verifying by induction. There are some technical aspects of induction that sometimes arise in inductive proofs in this context. Because it is possible to illustrate them more easily by using familiar recurrences, we shall do that.

A Wrinkle with Induction

Exercise 4.5-1 Carefully prove by induction that for any function T defined on the nonnegative integral powers of 2, if

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn$$

for some constant c, then $T(n) = O(n \log n)$.

We wish to show that $T(n) = O(n \log n)$. From the definition of big O, we can see that we wish to show $T(n) \leq kn \log n$ for some positive constant k (so long as n is larger than some value n_0).

We will now do something that may seem rather curious: We will consider the possibility that we have a value of k for which the inequality holds. Then, in analyzing the consequences of this possibility, we will discover that there are assumptions we need to make about k in order for such a k to exist. What we will really be doing is experimenting to see how to choose k to make an inductive proof work.

We are given that $T(n) \leq 2T(n/2) + cn$ for all positive integers n that are powers of 2. We want to prove there is another positive real number $k > 0$ and an $n_0 > 0$ such that $T(n) \leq kn \log n$ for $n > n_0$. We cannot expect to have the inequality $T(n) \leq kn \log n$ hold for $n = 1$, because $\log 1 = 0$. To have $T(2) \leq k \cdot 2 \log 2 = k \cdot 2$, we must choose $k \geq T(2)/2$. This is the first assumption we must make about k. Our inductive hypothesis is that if n is a power of 2 and m is a power of 2, with $2 \leq m < n$, then $T(m) \leq km \log m$. Now $n/2 < n$, and because n is a power of 2 greater than 2, we have that $n/2 \geq 2$. By the inductive hypothesis, $T(n/2) \leq k(n/2) \log n/2$. But then

$$\begin{aligned} T(n) \leq 2T\left(\frac{n}{2}\right) + cn &\leq 2k\frac{n}{2}\log\frac{n}{2} + cn \\ &= kn\log\frac{n}{2} + cn \\ &= kn\log n - kn\log 2 + cn \\ &= kn\log n - kn + cn. \end{aligned}$$

Recall that we are trying to show that $T(n) \leq kn\log n$; but that is not quite what the preceding inequality tells us. Rather, the inequality shows that we need to make another assumption about k—namely, $-kn + cn \leq 0$, or equivalently $k \geq c$. If both of our assumptions about k are satisfied, we will have $T(n) \leq kn\log n$, and we can conclude, by the principle of mathematical induction, that for all $n > 1$ (so our n_0 is 2), $T(n) \leq kn\log n$; thus, $T(n) = O(n\log n)$.

A full inductive proof that $T(n) = O(n\log n)$ is actually embedded in the preceding discussion. However, because it might not appear to everyone to be a proof, in the next paragraph we summarize our observations in a more traditional-looking proof. Be aware that some authors and teachers prefer to write their proofs in a style that shows why they make certain choices about k. You should learn how to read discussions like the one above as proofs.

We want to show that if $T(n) \leq T(n/2) + cn$, then $T(n) = O(n\log n)$. We are given a real number $c > 0$ such that $T(n) \leq 2T(n/2) + cn$ for all $n > 1$. Choose k to be larger than or equal to $T(2)/2$ and larger than or equal to c. Then

$$T(2) \leq k \cdot 2\log 2,$$

because $k \geq T(2)/2$ and $\log 2 = 1$. Now assume that $n > 2$ and that for m with $2 \leq m < n$, we have $T(m) \leq km\log m$. Because n is a power of 2, we have $n \geq 4$, so that $n/2$ is an m with $2 \leq m < n$. Thus, by the inductive hypothesis,

$$T\left(\frac{n}{2}\right) \leq k\frac{n}{2}\log\frac{n}{2}.$$

Then by the recurrence,

$$\begin{aligned} T(n) &\leq 2k\frac{n}{2}\log\frac{n}{2} + cn \\ &= kn(\log n - 1) + cn \\ &= kn\log n + cn - kn \\ &\leq kn\log n, \end{aligned}$$

because $k \geq c$. Thus, by the principle of mathematical induction, $T(n) \leq kn \log n$ for all $n > 2$, and therefore, $T(n) = O(n \log n)$.

There are three things to note about this proof. First, without the preceding discussion, the choice of k seems arbitrary. Second, without the preceding discussion, the implicit choice of 2 for the n_0 in the big O statement also seems arbitrary. Third, the constant k is chosen in terms of the previous constant c. Because c was given to us by the recurrence, we may use it in choosing the constant that we use to prove a big O statement about solutions to the recurrence. If you compare the formal proof we just gave with the informal discussion that preceded it, you will find that each step of the formal proof actually corresponds to something we said in the informal discussion. Because the informal discussion explained why we were making the choices we did, it is natural that some people prefer the informal explanation to the formal proof.

Further Wrinkles in Induction Proofs

Exercise 4.5-2 Suppose that c is a real number greater than 0. Show by induction that any solution $T(n)$ to the recurrence

$$T(n) \leq T\left(\frac{n}{3}\right) + cn,$$

with n restricted to integer powers of 3, has $T(n) = O(n)$.

Exercise 4.5-3 Suppose that c is a real number greater than 0. Show by induction that any solution $T(n)$ to the recurrence

$$T(n) \leq 4T\left(\frac{n}{2}\right) + cn,$$

with n restricted to integer powers of 2, has $T(n) = O(n^2)$.

In Exercise 4.5-2, we are given a constant c such that $T(n) \leq T(n/3) + cn$ if $n > 1$. Because we want to show that $T(n) = O(n)$, we want to find two more constants n_0 and k such that $T(n) \leq kn$ whenever $n > n_0$.

We will choose $n_0 = 1$ here. (This was not an arbitrary choice; it is based on observing that the condition $T(n) \leq kn$ is not impossible to satisfy when $n = 1$.) To have $T(n) \leq kn$ for $n = 1$, we must assume $k \geq T(1)$. Assuming

inductively that $T(m) \leq km$ when $1 \leq m < n$, we can write

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{3}\right) + cn \\ &\leq k\left(\frac{n}{3}\right) + cn \\ &= kn + \left(c - \frac{2k}{3}\right) n. \end{aligned}$$

(Note that we used $kn/3 = kn - 2kn/3$ because we wanted to compare $T(n)$ with kn.) Thus, as long as $c - 2k/3 \leq 0$, that is, $k \geq (3/2)c$, we may conclude, by mathematical induction, that $T(n) \leq kn$ for all $n \geq 1$. Again, the elements of an inductive proof are in the preceding discussion; you should try to learn how to read the argument we just finished as a valid inductive proof. However, we now present something that looks more like an inductive proof.

We choose k to be the maximum of $T(1)$ and $3c/2$, and we choose $n_0 = 1$. To prove by induction that $T(x) \leq kx$, we begin by observing that $T(1) \leq k \cdot 1$. Next we assume that $n > 1$, and we assume inductively that for m with $1 \leq m < n$, we have $T(m) \leq km$. Now we may write

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{3}\right) + cn \\ &\leq \frac{kn}{3} + cn \\ &= kn + \left(c - \frac{2k}{3}\right) n \\ &\leq kn, \end{aligned}$$

because we chose k to be at least as large as $3c/2$, making $c - 2k/3$ negative or 0. Thus, by the principle of mathematical induction, we have $T(n) \leq kn$ for all $n \geq 1$, and so $T(n) = O(n)$.

Now let's analyze Exercise 4.5-3. We won't dot all the i's and cross all the t's here because there is only one major difference between this exercise and the previous one. We wish to prove that there are an n_0 and a k such that $T(n) \leq kn^2$ for $n > n_0$. Assuming we have chosen n_0 and k so that the base case holds, we can bound $T(n)$ inductively by assuming that $T(m) \leq km^2$

for $m < n$ and reasoning as follows:

$$\begin{aligned} T(n) &\leq 4T\left(\frac{n}{2}\right) + cn \\ &\leq 4\left(k\left(\frac{n}{2}\right)^2\right) + cn \\ &= 4\left(\frac{kn^2}{4}\right) + cn \\ &= kn^2 + cn. \end{aligned}$$

To proceed as before, we would like to choose a value of k so that $cn \leq 0$. But we have a problem because both c and n are always positive! We have a statement that we know is true, by the master theorem, for example, and we have a proof method (induction) that worked nicely for similar problems. So, what went wrong?

The usual way to describe the problem we are facing is that although the statement is true, it is too weak to be proved by induction. To make the inductive proof work, we have to make an inductive hypothesis that puts some sort of negative quantity, such as a term like $-kn$, into the last line of our inequality. Let's see if we can prove something that is actually stronger than we were originally trying to prove—namely, $T(n) \leq k_1 n^2 - k_2 n$ for some positive constants k_1 and k_2. Proceeding as before, we get

$$\begin{aligned} T(n) &\leq 4T\left(\frac{n}{2}\right) + cn \\ &\leq 4\left(k_1\left(\frac{n}{2}\right)^2 - k_2\left(\frac{n}{2}\right)\right) + cn \\ &= 4\left(\frac{k_1 n^2}{4} - k_2\left(\frac{n}{2}\right)\right) + cn \\ &= k_1 n^2 - 2k_2 n + cn \\ &= k_1 n^2 - k_2 n + (c - k_2)n. \end{aligned}$$

Now we have to make $(c - k_2)n \leq 0$ for the last line to be at most $k_1 n^2 - k_2 n$. So, we choose $k_2 \geq c$. Once we pick a value of k_2, we can then choose k_1 large enough to make the base case work. Thus, we have proved inductively that $T(n) \leq k_1 n^2 - k_2 n$ for some constants k_1 and k_2; so, $T(n) = O(n^2)$.

At first glance, this approach seems paradoxical: Why is it easier to prove a stronger statement than it is to prove a weaker one? The answer is related to the nature of induction, in which the proof of $p(n)$ depends on the proof of $p(m)$ for $m < n$. Therefore, if your statement is too weak, the base case may be easier to prove, but the weakness will hinder your ability to prove the statement for larger values of n. In other words, when you want to prove something about $p(n)$, you are using $p(1) \wedge \cdots \wedge p(n-1)$. Thus, if these are stronger, they will be of greater help in proving $p(n)$. In the case above, the problem was that the statements $p(1), \ldots, p(n-1)$ were too weak, and thus we were not able to use them to prove $p(n)$. By using a stronger $p(1), \ldots, p(n-1)$, however, we were able to prove a stronger $p(n)$, one that implied the original $p(n)$ we wanted. When we give an induction proof in this way, we are using a **stronger inductive hypothesis.**

Dealing with Functions Other Than n^c

Our statement of the master theorem involved a recursive term plus an added term that was $\Theta(n^c)$. Sometimes algorithmic problems lead us to consider other kinds of functions for the added term. The most common such example is when that added function involves logarithms. For example, consider the recurrence

$$T(n) = \begin{cases} 2T(n/2) + n\log n & \text{if } n > 1, \\ 1 & \text{if } n = 1, \end{cases} \tag{4.26}$$

where n is a power of 2. Just as before, we can draw a recursion tree; the whole methodology works, but our sums may be a little more complicated. The tree for this recurrence is shown in Figure 4.10.

This tree is similar to the tree for $T(n) = 2T(n/2) + n$, except that the work on Level i is $n\log(n/2^i)$ for $i \geq 2$, and, for the bottom level, it is n (the number of subproblems) times 1. Thus, if we sum the work per level, we get

$$\begin{aligned} \sum_{i=0}^{\log(n)-1} n\log\left(\frac{n}{2^i}\right) + n &= n\left(\sum_{i=0}^{\log(n)-1} \log\left(\frac{n}{2^i}\right) + 1\right) \\ &= n\left(\sum_{i=0}^{\log(n)-1} (\log n - \log 2^i) + 1\right) \\ &= n\left(\sum_{i=0}^{\log(n)-1} \log n - \sum_{i=0}^{\log n - 1} i\right) + n \end{aligned}$$

$$= n\left((\log n)(\log n) - \frac{(\log n)\big(\log(n)-1\big)}{2}\right) + n$$
$$= O(n \log^2 n).$$

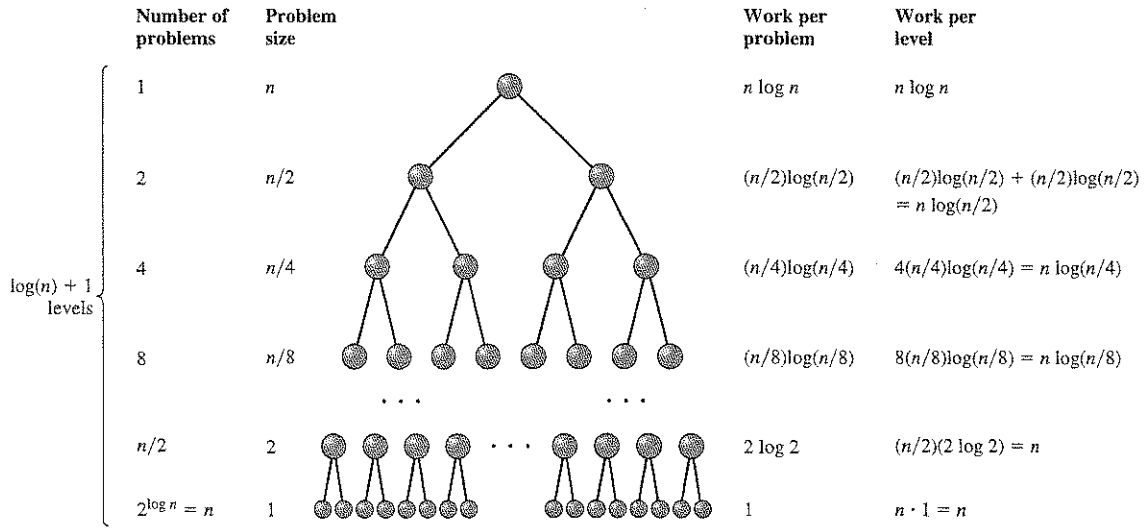


Figure 4.10: The recursion tree for Recurrence 4.26

Notice that in the second-to-last line, there are two places where we multiplied $\log n$ by itself. Because of the 2 in the denominator, the second product will not cancel out the first (and the other terms we get by carrying out the indicated multiplications are smaller than $n \log^2 n$). Thus, our solution is in fact $\Theta(n \log^2 n)$.

Exercise 4.5-4 Find the best big O bound you can on the solution to the recurrence

$$T(n) = \begin{cases} T(n/2) + n \log n & \text{if } n > 1, \\ 1 & \text{if } n = 1, \end{cases} \tag{4.27}$$

assuming n is a power of 2. Is this bound a big Θ bound?

The tree for this recurrence is in Figure 4.11.

Notice that the work done at the bottom node of the tree is determined by the statement $T(1) = 1$ in our recurrence; it is not $1 \log 1$. Summing the

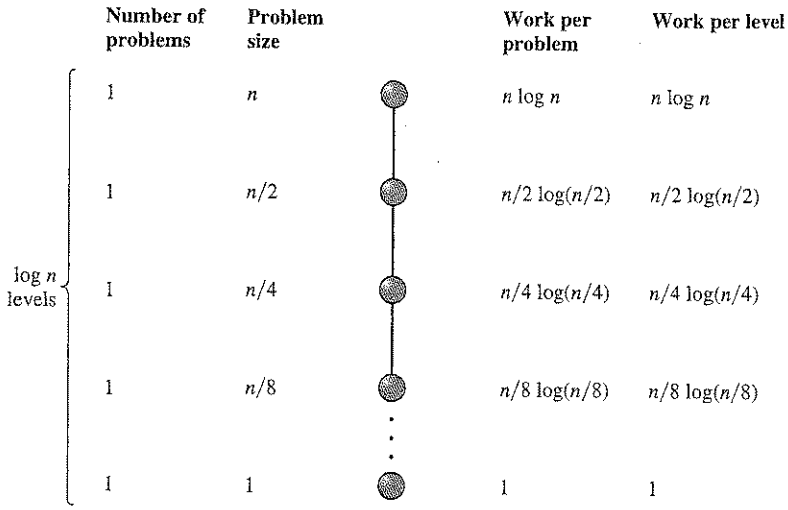


Figure 4.11: The recursion tree for Recurrence 4.27

work, we get

$$\begin{aligned}
1 + \sum_{i=0}^{\log(n)-1} \frac{n}{2^i} \log \frac{n}{2^i} &= 1 + n \left(\sum_{i=0}^{\log(n)-1} \frac{1}{2^i} (\log n - \log 2^i) \right) \\
&= 1 + n \left(\sum_{i=0}^{\log(n)-1} \left(\frac{1}{2}\right)^i \big(\log(n) - i\big) \right) \\
&\leq 1 + n \left(\log n \sum_{i=0}^{\log(n)-1} \left(\frac{1}{2}\right)^i \right) \\
&\leq 1 + n(\log n)(2) \\
&= O(n \log n).
\end{aligned}$$

Note that the largest term in the sum in our third-to-last line of equations and inequalities is $\log(n)$ and that none of the terms in the sum are negative. This means that n times the sum is at least $n \log n$. Therefore, we have that $T(n) = \Theta(n \log n)$.

Important Concepts, Formulas, and Theorems

1. *Recurrence inequality. Recurrence inequalities* state that $T(n)$ is less than or equal to some expression involving values of $T(m)$ for $m < n$. A *solution* to a recurrence inequality is a function T that satisfies the inequality.

2. *Recursion trees for recurrence inequalities.* We can analyze recurrence inequalities via a recursion tree. The process is virtually identical to our previous use of recursion trees. We must, however, keep in mind that on each level, we are really computing an upper bound on the work done on that level.
3. *Discovering necessary assumptions for an inductive proof.* Suppose we are trying to prove a statement that there is a value k such that an inequality of the form $f(n) \leq kg(n)$ is true or that some other statement that involves the parameter k is true. We may start an inductive proof without knowing a value for k and determine conditions on k that make the proof valid by analyzing the assumptions that we need to make in order for the inductive proof to work. When written properly, such an explanation is a valid proof.
4. *Making a stronger inductive hypothesis.* If we are trying to prove by induction a statement of the form $p(n) \Rightarrow q(n)$ and we have a statement $s(n)$ such that $s(n) \Rightarrow q(n)$, it is sometimes useful to try to prove the statement $p(n) \Rightarrow s(n)$. This process is known as proving a *stronger* statement or making a *stronger* inductive hypothesis. It sometimes works because it gives an inductive hypothesis that suffices to prove the stronger statement, even though our original statement $q(n)$ did not give an inductive hypothesis sufficient to prove the original statement. However, we must be careful in our choice of $s(n)$, because we have to be able to succeed in proving $p(n) \Rightarrow s(n)$.
5. *When the master theorem does not apply.* To deal with recurrences of the form

$$T(n) = \begin{cases} aT(\lceil n/b \rceil) + f(n) & \text{if } n > 1, \\ d & \text{if } n = 1, \end{cases}$$

where $f(n)$ is not $\Theta(n^c)$, recursion trees and iterating the recurrence are appropriate tools even though the master theorem does not apply. The same holds for recurrence inequalities.

Problems

All problems with blue boxes have an answer or hint available at the end of the book.

1. Suppose that c is a real number greater than 0. Show by induction that any solution $T(n)$ to the recurrence

$$T(n) \leq T\left(\frac{n}{4}\right) + cn,$$

with n restricted to integer powers of 4, has $T(n) = O(n)$.

2. Prove by induction that if $T(n) \leq 4T(n/2) + n^2$, then $T(n) = O(n^2 \log n)$ (assuming n is a power of 2).

3. Show by induction that any solution to a recurrence of the form

$$T(n) \leq 2T\left(\frac{n}{3}\right) + c \log_3 n$$

is $O(n \log_3 n)$. What happens if you replace 2 with 3? Explain why. Would it make a difference if you used a different base for the logarithm (only an intuitive explanation is needed here)?

4. What happens if you replace the 2 in Problem 3 with 4? Do you still get the same big O upper bound? If not, what do you get? (*Hint:* One way to attack this is with recursion trees. It might also be helpful to ask what happens if you replace the $\log_3 n$ with 1 and then with n.)

5. Is the big O upper bound in Problem 3 actually a big Θ bound?

6. **a.** Find the best big O upper bound you can to any solution to the recurrence

$$T(n) = \begin{cases} 4T(n/2) + n \log n & \text{if } n > 1, \\ 1 & \text{if } n = 1. \end{cases}$$

b. Assuming that you were able to guess the result you got in part a, prove by induction that your answer is correct.

7. Is the big O upper bound in Problem 7 actually a big Θ bound?

8. Show by induction that

$$T(n) = \begin{cases} 8T(n/2) + n \log n & \text{if } n > 1, \\ d & \text{if } n = 1, \end{cases}$$

has $T(n) = O(n^3)$ for any solution $T(n)$.

9. Is the big O upper bound in Problem 9 actually a big Θ bound?

10. Give the best big O upper bound you can for the solution to the recurrence

$$T(n) = 2T\left(\frac{n}{3} - 3\right) + n$$

(making an informed guess is not a bad idea here). Then prove by induction that your upper bound is correct.

HARD Find the best big O upper bound you can to any solution to the recurrence defined on nonnegative integers by

$$T(n) \leq 2T\left(\left\lceil \frac{n}{2} \right\rceil + 1\right) + cn.$$

(There is nothing wrong with informed guesswork.) Prove by induction that your answer is correct.

4.6 RECURRENCES AND SELECTION

The Idea of Selection

One common problem that arises in algorithms is that of **selection**. In this situation, we are given n distinct data items from some set that has an underlying order. That is, given any two items a and b from that set, we can determine whether $a < b$. (Integers satisfy this property, but colors do not.) Given these n items and some value i with $1 \leq i \leq n$, we are asked to find the ith-smallest item in the set. For example, in the set

$$S = \{3, 2, 8, 6, 4, 11, 7\}, \tag{4.28}$$

the first smallest ($i = 1$) is 2, the third smallest ($i = 3$) is 4, and the seventh smallest ($i = n = 7$) is 11. An important special case is that of finding the **median**, which is the case of $i = \lceil n/2 \rceil$. Another important special case is finding percentiles; for example, the 90th percentile is the case $i = \lceil 0.9n \rceil$. As this suggests, i is frequently given as some fraction of n.

Exercise 4.6-1 How do you find the minimum ($i = 1$) or maximum ($i = n$) in a set? What is the running time? How do you find the second-smallest element? Does this approach extend to finding the ith smallest? What is the running time?

Exercise 4.6-2 Give the fastest algorithm you can to find the median ($i = \lceil n/2 \rceil$).

In Exercise 4.6-1, the simple $O(n)$ time algorithm of going through the list and keeping track of the minimum value seen so far will suffice to find the minimum. Similarly, if we want to find the second smallest, we can go through the list once to find the smallest, remove it, and then go through the new list to find the smallest. This takes $O(n + n - 1) = O(n)$ time. If we extend this to finding the ith smallest, the algorithm will take $O(in)$

time. Thus, for finding the median, this method takes $O(n^2)$ time. In fact, it takes $\Theta(n^2)$ time.

A better idea for finding the median is first to sort the items and then to take the item in position $n/2$. Because we can sort in $O(n \log n)$ time, this algorithm will take $O(n \log n)$ time. Thus, if $i = O(\log n)$, we might want to run the algorithm of the previous paragraph; otherwise, we would run this algorithm.[9]

All of these approaches, when applied to the median, take at least some multiple of $(n \log n)$ units of time.[10] The best sorting algorithms take $O(n \log n)$ time also, and one can prove every comparison-based sorting algorithm takes $\Omega(n \log n)$ time. This raises the natural question of whether it is possible to do selection any faster than sorting. In other words, is finding the median element or finding the ith-smallest element of a set significantly easier than ordering (sorting) the whole set?

A Recursive Selection Algorithm

Suppose that we magically knew how to find the median in $O(n)$ time. That is, we have a routine MagicMedian that returns the median when given a set A as input. We could then use this routine in a divide-and-conquer algorithm for Select, as follows.

```
Select(A,i,n)

     // Selects the ith-smallest element in set A,
     // where n = |A|
(1)  if (n == 1)
(2)       return the one item in A
(3)  else
(4)       p = MagicMedian(A)
(5)       Let H be the set of elements greater than p
(6)       Let L be the set of elements less than or equal to p
(7)       if (i ≤ |L|)
(8)            return Select(L, i,|L|)
(9)       else
(10)           return Select(H, i - |L|,|H|)
```

[9] We also note (for those who know about heaps) that the running time can be improved to $O(n + i \log n)$ by first creating a *heap*, which takes $O(n)$ time, and then performing a delete-min operation i times.

[10] An alternate notation for $f(x) = O(g(x))$ is $g(x) = \Omega(f(x))$. Notice the change in roles of f and g. In this notation, we say that all these algorithms take $\Omega(n \log n)$ time. (In analytic number theory, Ω is used in several different contexts with somewhat different

By H, we do not mean the elements that come after p in the list; rather, we mean the elements of the list that are larger than p in the underlying ordering of our set. This algorithm is based on the following simple observation: If we could divide the set A into a "lower" half (L) and an "upper" half (H), then we know in which of these two sets the ith-smallest element in A will be. Namely, if $i \leq \lceil n/2 \rceil$, it will be in L, and otherwise, it will be in H. Thus, we can recursively look in one or the other set. We can easily partition the data into two sets by making one pass through the data, copying the numbers less than or equal to p into L, and copying the numbers larger than p into H.[11]

The only additional detail is that if we look in H, then we no longer look for the ith smallest. Instead, we look for the $i - \lceil n/2 \rceil$th smallest, because H is formed by removing the $\lceil n/2 \rceil$-smallest elements from A.

For example, if the input is the set given in Equation 4.28, and if $p = 6$, then the set L would be $\{3, 2, 6, 4\}$, and H would be $\{8, 11, 7\}$. If i were 2, we would recurse on the set L, with $i = 2$. On the other hand, if i were 6, we would recurse on the set H, with $i = 6 - 4 = 2$. Observe that the second-smallest element in H is 8, as is the sixth-smallest element in S.

We can express the running time of Select by the following recurrence:

$$T(n) \leq T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn.$$

From the master theorem, we know that any function that satisfies this recurrence has $T(n) = O(n)$.

So, we can conclude that if we already know how to find the median in linear time, we can design a divide-and-conquer algorithm that will solve the selection problem in linear time.[12] However, this is nothing to write home about (yet!).

Selection without Knowing the Median in Advance

Sometimes a knowledge of solving recurrences can help us design algorithms. What kinds of recurrences do we know about that have solutions $T(n)$ with $T(n) = O(n)$? In particular, consider recurrences of the form $T(n) \leq T(n/b) + cn$, and ask when they have solutions with $T(n) = O(n)$. Using the master theorem, we see that because $\log_b 1 = 0 < 1$ for any b, then for any b allowed by the master theorem, all solutions to this recurrence will have $T(n) = O(n)$. (Note that b does not have to be an integer.)

[11] We can do this more efficiently, and "in place," using the partition algorithm of quicksort.

[12] We say an algorithm runs in linear time if its running time on an input of size n is $O(n)$.

If we let $b' = 1/b$, then we can say equivalently that as long as we can solve a problem of size n by solving (recursively) a problem of size $b'n$ for some $b' < 1$ and by doing $O(n)$ additional work, our algorithm will run in $O(n)$ time. Interpreting this in the selection problem, it says that as long as we can choose p in $O(n)$ time to ensure that both L and H have size at most $b'n$, then we will have a linear-time algorithm. (You might ask, "What about actually dividing our set into L and H? Doesn't that take some time, too?" Yes it does, but we already know we can do the division into H and L in $O(n)$ time; so, if we can find p in $O(n)$ time as well, then we can do both these things in $O(n)$ time.)

In particular, suppose that we can choose p in $O(n)$ time to ensure that both L and H have size at most $(3/4)n$. Then the running time is described by the recurrence $T(n) \leq T(3n/4) + O(n)$, and we will be able to solve the selection problem in linear time.

To see why $(3/4)n$ is relevant, suppose that instead of the "black box" MagicMedian, we have a much weaker magic black box that only guarantees that it will return some number in the middle half of our set in $O(n)$ time. In other words, it will return a number that is guaranteed to be somewhere between the $(n/4\text{th})$-smallest number and the $(3n/4\text{th})$-smallest number. If we use the number given by this magic box to divide our set into H and L, then neither set will have size more than $3n/4$. We will call this black box a MagicMiddle box, and we use it in the following algorithm:

```
Select1(A,i,n)

    // Selects the ith-smallest element in set A,
    // where n = |A|
(1) if (n == 1)
(2)       return the one item in A
(3) else
(4)       p = MagicMiddle(A)
(5)       Let H be the set of elements greater than p
(6)       Let L be the set of elements less than or equal to p
(7)       if (i ≤ |L|)
(8)             return Select1(L,i,|L|)
(9)       else
(10)            return Select1(H,i−|L|,|H|)
```

The Select1 algorithm is similar to Select. The only difference is that p is now only guaranteed to be in the middle half. When we recurse in Select1, we decide whether to recurse on L or H based on whether i is less than

or equal to $|L|$. The element p is called a **partition element** because it is used to partition our set A into the two sets L and H.

We have made progress, because now we don't need to assume that we can find the median in order to have a linear-time algorithm; we only need to assume that we can find one number in the middle half of the set. This problem seems simpler than the original problem, and, conceptually, it is. Thus, our knowledge of which recurrences have solutions that are $O(n)$ led us toward a more plausible algorithm.

An Algorithm to Find an Element in the Middle Half

It takes a clever algorithm to find an item in the middle half of our set. We now describe such an algorithm in which we first choose a subset of the numbers and then *recursively* find the median of that subset. (The condition that $n < 60$ in Line 2 is a technical condition that will be justified later.)

```
MagicMiddle(A)

(1) Let n = |A|
(2) if (n < 60)
(3)      use sorting to return the median of A
(4) else
(5)      Break A into k = ⌈n/5⌉ groups G_1, ..., G_k
         with ⌊n/5⌋ of size 5 and perhaps one of smaller size
(6)      for i = 1 to k
(7)           find m_i, the median of G_i (by sorting)
(8)      Let M = {m_1, ..., m_k}
(9)      return Select1(M, ⌈k/2⌉, k)
```

We first give a visual description of why the median of medians is in the middle half of A in the special case where the size of A is a multiple of 5; then we prove in general that it is. Assume $|A|$ is a multiple of 5. Then $|A| = 5k$.

Consider arranging the elements as follows. List each set G_i of 5 vertically in sorted order, with the smallest element on top. Then line up all $n/5$ of these lists, with those with median less than the median of the medians on the left (and those with median larger than the median of the medians on the right). We get the picture in Figure 4.10. In this figure, the medians are in white and the median of medians is in blue. The figure includes all the inequalities that we know from the ordering information we have. We use arrows to indicate that the medians on the left are less than the median of medians and those on the right are greater than the median of medians.

We use m^* to denote the median of medians, which is returned by MagicMiddle. We show that m^* must be in the middle half of set A when $|A|$ is large enough by considering a set S of elements guaranteed to be smaller than m^* and a set B of elements guaranteed to be bigger. Then we determine how large $|A|$ must be to ensure that $|S|$ and $|B|$ are always at least $|A|/4$.

We call the medians smaller than m^* "small medians" and those bigger "big medians." If m_i is a small median, then m_i and the two elements less than it in G_i (and thus above it in Figure 4.12) are less than m^*. The two elements above m^* in its column are less than m^*. In Figure 4.13, we draw a curve around the set S of elements.

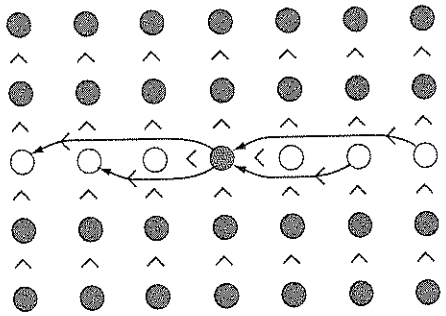


Figure 4.12: Dividing a set into $n/5$ parts of size 5, finding the median of each part, and finding the median of the medians

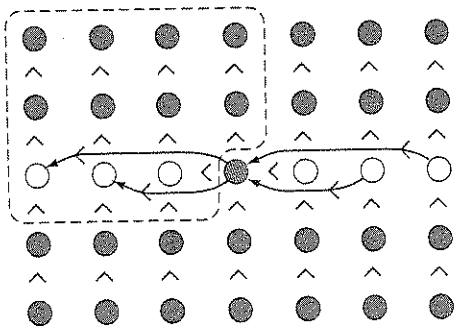


Figure 4.13: The enclosed elements are less than the median of the medians

Symmetrically, every big median is larger than m^*, as are the two elements below m^* in its column. The set B of elements is enclosed in a curve in Figure 4.14.

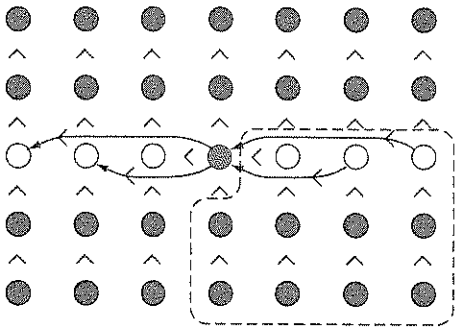


Figure 4.14: The enclosed elements are greater than the median of the medians

If we can choose n so that S and B each have at least one-fourth of the elements of A, we will know that m^*, which cannot be in either S or B, must be in the middle half of A. For this reason, we try to compute the sizes of S and B in terms of n.

Using k as in MagicMiddle, m^* is in column $\lceil k/2 \rceil$ of Figure 4.11. Therefore, S has three elements from each of the first $\lceil k/2 \rceil - 1$ columns and two from column $\lceil k/2 \rceil$. Using $k = n/5$, we get

$$|S| = 3\left(\left\lceil \frac{n}{2 \cdot 5} \right\rceil - 1\right) + 2 = 3\left\lceil \frac{n}{10} \right\rceil - 1.$$

Because $\lceil n/10 \rceil > n/10$, we can make $|S| \geq n/4$ by making $3n/10 - 1 \geq n/4$, which gives us $0.05n \geq 1$, or $n \geq 20$.

The number of columns to the right of column $\lceil k/2 \rceil$ is $k - \lceil k/2 \rceil$, so the size of B is

$$3\left(k - \left\lceil \frac{k}{2} \right\rceil\right) + 2 = 3\left(\frac{n}{5} - \left\lceil \frac{n}{10} \right\rceil\right) + 2.$$

Because $\lceil n/10 \rceil < 1 + n/10$, we have

$$|B| > 3\left(\frac{n}{5} - \frac{n}{10} - 1\right) + 2 = \frac{3n}{10} - 1.$$

Thus, we can make $|B| > n/4$ by making $0.3n - 1 \geq 0.25n$, or $0.05n \geq 1$, which gives us $n \geq 20$. Therefore, in the case where n is divisible by 5, as long as $n \geq 20$, we have that m^* is in the middle half of A. We now turn to the general case in which n need not be a multiple of 5.

Lemma 4.14 The value returned by MagicMiddle(A) is in the middle half of A.

Proof We let m^* denote the output of MagicMiddle(A), so that m^* is the $\lceil k/2 \rceil$th element of the m_i's in sorted order. Thus, $\lceil k/2 \rceil - 1$ medians m_i are less than m^*, as are the elements in G_i less than m_i. Choose j so that $m^* \in G_j$. Then the elements of G_j less than m_j are less than m^*. However, for all but perhaps one G_i (including G_j) with $m_i \leq m^*$, there are two elements less than m_i, so that the set S' of elements less than m^* has size at least $3(\lceil k/2 \rceil - 1)$. Because k is at least $n/5$ and $\lceil n/10 \rceil \geq n/10$, we have that

$$|S'| \geq 3\left(\left\lceil \frac{n}{10} \right\rceil - 1\right) \geq 3\left(\frac{n}{10} - 1\right).$$

Thus, if we choose n so that

$$3\left(\frac{n}{10} - 1\right) = 0.3n - 3 \geq \frac{n}{4}, \tag{4.29}$$

we will have $S' \geq n/4$. But Equation 4.29 gives us $0.3n - 3 \geq 0.25n$, or $n \geq 60$. Now because there are $k - \lceil k/2 \rceil$ medians m_i greater than m^*, we have, as with S', that if B' is the set of elements of A larger than m^*, then B' has at least $3(k - \lceil k/2 \rceil)$ elements. Because $\lceil k/2 \rceil < k/2 + 1$, we have

$$|B| \geq 3\left(k - \frac{k}{2} - 1\right) = 3\left(\frac{k}{2} - 1\right) = 3\left(\frac{1}{2}\left\lceil \frac{n}{5} \right\rceil - 1\right) \geq 3\frac{n}{10} - 3 = 0.3n - 3.$$

Thus, if we choose n so that Equation 4.29 holds—that is, so that $n \geq 60$—then we have both $|S'| > n/4$ and $|B'| > n/4$. Therefore, m^* is in the middle half of A.

Note that we don't actually identify all the nodes that are guaranteed to be, say, less than the median of medians; we are just guaranteed that the proper number exists.

Because we only have the guarantee that MagicMiddle gives an element in the middle half of the set if the set has at least 60 elements, we modify Select1 to start by checking whether $n < 60$ and then sorting the set to find the element in position i if $n < 60$. Because 60 is a constant, sorting and finding the desired element takes, at most, a constant amount of time.

An Analysis of the Revised Selection Algorithm

Exercise 4.6-3 Let $T(n)$ be the running time of the modified Select1 on n items. How can you express the running time of MagicMiddle in terms of $T(n)$?

Exercise 4.6-4 What is a recurrence for the running time of Select1? (*Hint:* How could Exercise 4.6-3 help you?)

Exercise 4.6-5 Can you prove by induction that each solution to the recurrence for Select1 is $O(n)$?

For Exercise 4.6-3, we have the following steps.

1. Divide the items into sets of five; this takes $O(n)$ time.
2. Find the median of each five-element set. (We can find this median by any straightforward method we choose and still only take, at most, a constant amount of time; we don't use recursion here.) There are $n/5$ sets, and we spend no more than some constant time per set, so the total time is $O(n)$.
3. Recursively call Select1 to find the median of medians; this takes $T(n/5)$ time.
4. Partition A into those elements less than or equal to the "magic middle" and those that are not, which takes $O(n)$ time.

Thus, the total running time is $T(n/5) + O(n)$, which implies that for some n_0 there is a constant $c_0 > 0$ such that the running time is no more than $c_0 n$ for all $n > n_0$. Even if $n_0 > 60$, there are only finitely many cases between 60 and n_0, which means there is a constant c such that the running time of MagicMiddle is no more than $T(n/5) + cn$ for $n \geq 60$.

We now get a recurrence for the running time of Select1. Note that for $n \geq 60$, Select1 has to call MagicMiddle and then recurse on either L or H, each of which has size at most $3n/4$. For $n < 60$, note that it takes no more than some constant amount d of time to find the median by sorting. Therefore, we get the following recurrence for the running time of Select1:

$$T(n) \leq \begin{cases} T(3n/4) + T(n/5) + c'n & \text{if } n \geq 60, \\ d & \text{if } n < 60. \end{cases}$$

This answers Exercise 4.6-4.

As Exercise 4.6-5 requests, we can now verify by induction that $T(n) = O(n)$. What we want to prove is that there is a constant k such that $T(n) \leq kn$. What the recurrence tells us is that there are constants c and d such that $T(n) \leq T(3n/4) + T(n/5) + cn$ if $n \geq 60$; otherwise, $T(n) \leq d$. For the base case, we have $T(n) \leq d \leq dn$ for $n < 60$, so we choose k to be at least d; then $T(n) \leq kn$ for $n < 60$. We now assume that $n \geq 60$ and $T(m) \leq km$

for values $m < n$. We get

$$\begin{aligned} T(n) &\leq T\left(\frac{3n}{4}\right) + T\left(\frac{n}{5}\right) + cn \\ &\leq \frac{3kn}{4} + \frac{kn}{5} + cn \\ &= \frac{19kn}{20} + cn \\ &= kn + \left(c - \frac{k}{20}\right)n. \end{aligned}$$

As long as $k \geq 20c$, this is at most kn; so we simply choose k this big, and by the principle of mathematical induction, we have $T(n) < kn$ for all positive integers n.

Theorem 4.15 The revised Select1 algorithm runs in time $T(n) = O(n)$.

Proof The proof is given in the discussion of Exercises 4.6-3 through 4.6-5.

Uneven Divisions

The kind of recurrence we found for the running time of Select1 is actually an instance of a more general class, which we now explore.

Exercise 4.6-6 We already know that when $g(n) = O(n)$, every solution of $T(n) = T(n/2) + g(n)$ satisfies $T(n) = O(n)$. Use the master theorem to find a big O bound to the solution of $T(n) = T(cn) + g(n)$ for any constant $c < 1$, assuming that $g(n) = O(n)$.

Exercise 4.6-7 Use the master theorem to find big O bounds to all solutions of $T(n) = 2T(cn) + g(n)$ for any constant $c < 1/2$, assuming $g(n) = O(n)$.

Exercise 4.6-8 Suppose $g(n) = O(n)$ and you have a recurrence of the form $T(n) = T(an) + T(bn) + g(n)$ for some nonnegative constants a and b. What conditions on a and b guarantee that all solutions to this recurrence have $T(n) = O(n)$?

Using the master theorem for Exercise 4.6-6, we get $T(n) = O(n)$, because $\log_{1/c} 1 < 1$. We also get $T(n) = O(n)$ for Exercise 4.6-7, because

$\log_{1/c} 2 < 1$ for $c < 1/2$. You might now guess that as long as $a + b < 1$, any solution to the recurrence $T(n) \leq T(an) + T(bn) + cn$ has $T(n) = O(n)$. We will now see why this is the case.

First, let's return to the recurrence $T(n) = T(3n/4) + T(n/5) + g(n)$, where $g(n) = O(n)$. Let's try to draw a recursion tree. This recurrence doesn't quite fit our model for recursion trees, because the two subproblems have unequal size (thus, we can't even write the problem size on the left), but we will try to draw a recursion tree in Figure 4.15 anyway and see what happens.

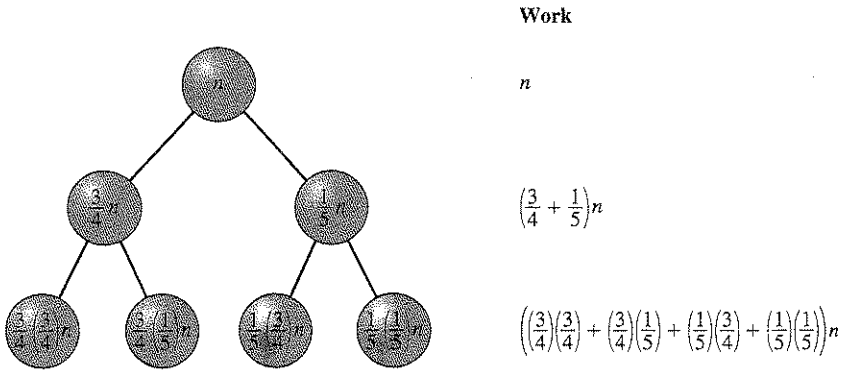


Figure 4.15: Attempting a recursion tree for $T(n) = T(3/4)n + T(n/5) + g(n)$

As we draw Levels 1 and 2, we see that at Level 1, we have $(3/4 + 1/5)n$ work. At Level 2, we have

$$\left(\left(\frac{3}{4}\right)^2 + 2\left(\frac{3}{4}\right)\left(\frac{1}{5}\right) + \left(\frac{1}{5}\right)^2\right)n$$

work. Were we to work out the third level, we would see that we have

$$\left(\left(\frac{3}{4}\right)^3 + 3\left(\frac{3}{4}\right)^2\left(\frac{1}{5}\right) + 3\left(\frac{3}{4}\right)\left(\frac{1}{5}\right)^2 + \left(\frac{1}{5}\right)^3\right)n.$$

Thus, we can see a pattern emerging. At Level 1, we have $(3/4 + 1/5)n$ work. At Level 2, we have, by the binomial theorem, $(3/4 + 1/5)^2 n$ work. At Level 3, we have, by the binomial theorem, $(3/4 + 1/5)^3 n$ work. And, similarly, at Level i, we have $\big((3/4) + (1/5)\big)^i n = (19/20)^i n$ work. Thus,

when we sum over all levels, we get

$$\sum_{i=0}^{O(\log n)} \left(\frac{19}{20}\right)^i n \leq \left(\frac{1}{1-\frac{19}{20}}\right) n = 20n$$

for an upper bound on the total work. We have actually ignored one detail here. In contrast to a recursion tree in which all subproblems at a level have equal size, the "bottom" of the tree is more complicated. Different branches of the tree will reach problems of size 1 and terminate at different levels. For example, the branch that follows all 3/4s will bottom out after $\log_{4/3} n$ levels, while the one that follows all 1/5s will bottom out after $\log_5 n$ levels. However, the analysis above *overestimates the work*—that is, it assumes that nothing bottoms out until everything bottoms out, which occurs at $\log_{20/19} n$ levels. In fact, the upper bound we gave on the sum is what we would get by assuming that the recurrence never bottoms out.

We see here something general happening. It seems as if to understand a recurrence of the form $T(n) = T(an) + T(bn) + g(n)$, with $g(n) = O(n)$, we can study the simpler recurrence $T(n) = T\big((a+b)n\big) + g(n)$ instead. For a more precise formulation, see Problem 4, which solves Exercise 4.6-8. This simplifies things enough (in particular, it lets us use the master theorem) to let us analyze a larger class of recurrences. Turning to the median algorithm, it tells us that the important thing that happened there was that the sizes of the two recursive calls, namely, $3n/4$ and $n/5$, summed to a proper fraction of n. As long as an algorithm has a recurrence of the form $T(n) = T(an) + T(bn) + g(n)$, has $a + b < 1$, and has $g(n) = O(n)$, the algorithm will work in $O(n)$ time.

Important Concepts, Formulas, and Theorems

1. *Median.* The *median* of a set (with an underlying order) of n elements is the element that would be in position $\lceil n/2 \rceil$ if the set were listed in order.
2. *Percentile.* The pth percentile of a set (with an underlying order) is the element that would be in position $\lceil (p/100)n \rceil$ if the set were listed in order.
3. *Selection.* Given an n-element set with some underlying order, the problem of *selection* of the ith-smallest element is that of finding the element that would be in the ith position if the set were listed in order. Note that i is often expressed as a fraction of n.
4. *Partition element.* A *partition element* in an algorithm is an element of a set (with an underlying order) that is used to divide the set into

two parts, those that come before or are equal to the element (in the underlying order) and then the remaining elements. Notice that the order in which the set is given to the algorithm is not necessarily (in fact, not usually) the underlying order.

5. *Linear-time algorithms.* If the running time of an algorithm satisfies a recurrence of the form $T(n) \leq T(an) + cn$, with $0 \leq a < 1$, or a recurrence of the form $T(n) \leq T(an) + T(bn) + cn$, with a and b nonnegative and $a + b < 1$, then $T(n) = O(n)$.
6. *Finding a good partition element.* If a set (with an underlying order) has 60 or more elements, then the procedure of breaking the set into pieces of size 5 (plus one leftover piece, if necessary), finding the median of each piece, and then finding the median of the medians gives an element guaranteed to be in the middle half of the set.
7. *Selection algorithm.* The selection algorithm that runs in linear time sorts a set of size less than 60 to find the element in the ith position; otherwise,
   - it recursively uses the median of medians of five to find a partition element,
   - it uses that partition element to divide the set into two pieces, and
   - then it looks for the appropriate element in the appropriate piece recursively.

Problems

All problems with blue boxes have an answer or hint available at the end of the book.

1. Find the best big O bound you can on $T(n)$ if it satisfies the recurrence $T(n) \leq T(n/4) + T(n/2) + n$, with $T(n) = 1$ if $n < 4$.

2. In the MagicMiddle algorithm, suppose you broke the data into $n/7$ sets of size 7. What would the running time of Select1 be?

3. Let

$$T(n) = \begin{cases} T(n/3) + T(n/2) + n & \text{if } n \geq 6, \\ 1 & \text{otherwise}, \end{cases}$$

and let

$$S(n) = \begin{cases} S(5n/6) + n & \text{if } n \geq 6, \\ 1 & \text{otherwise.} \end{cases}$$

Draw recursion trees for T and S. What are the big O bounds on solutions to the recurrences? Use the recursion trees to argue that $T(n) \leq S(n)$ for all n.

4. Suppose you are given that a and b are nonnegative real numbers, with $a + b < 1$, and that c is a nonnegative real number. Explain why it is that if $T(n) \leq T(an) + T(bn) + cn$, then $T(n) = O(n)$. Explain how this solves Exercise 4.6-8.
5. Find a big Θ bound (the best you know how to get) on solutions to the recurrence $T(n) = T(n/3) + T(n/6) + T(n/4) + n$, with $T(1) = 1$.
6. Find a big Θ bound to solutions to the recurrences $T(n/4) + T(3n/4) + dn \leq T(n) \leq T(n/4) + T(3n/4) + cn$.
7. In the MagicMiddle algorithm, suppose you broke your data into $n/3$ sets of size 3. What would the running time of Select1 be?
8. Find a big O upper bound (the best you know how to get) on solutions to the recurrence $T(n) = T(n/4) + T(n/2) + n^2$, with $T(n) = 1$ if $n < 4$.
9. Note that we have chosen the median of an n-element set to be the element in position $\lceil n/2 \rceil$. We have also chosen to put the median of the medians into the set L of algorithm Select1. Show that this allows you to prove that $T(n) \leq T(3n/4) + T(n/5) + cn$ for $n \geq 40$ rather than $n \geq 60$. (You will need to analyze separately the case where $\lceil n/5 \rceil$ is even and the case where it is odd.) Is 40 the least value possible?