

As noted in Section 2.3.2, when an algorithm contains a recursive call to itself, its running time can often be described by a recurrence. A *recurrence* is an equation or inequality that describes a function in terms of its value on smaller inputs. For example, we saw in Section 2.3.2 that the worst-case running time  $T(n)$  of the MERGE-SORT procedure could be described by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1, \end{cases} \quad (4.1)$$

whose solution was claimed to be  $T(n) = \Theta(n \lg n)$ .

This chapter offers three methods for solving recurrences—that is, for obtaining asymptotic “ $\Theta$ ” or “ $O$ ” bounds on the solution. In the *substitution method*, we guess a bound and then use mathematical induction to prove our guess correct. The *recursion-tree method* converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion; we use techniques for bounding summations to solve the recurrence. The *master method* provides bounds for recurrences of the form

$$T(n) = aT(n/b) + f(n),$$

where  $a \geq 1$ ,  $b > 1$ , and  $f(n)$  is a given function; it requires memorization of three cases, but once you do that, determining asymptotic bounds for many simple recurrences is easy.

### Technicalities

In practice, we neglect certain technical details when we state and solve recurrences. A good example of a detail that is often glossed over is the assumption of integer arguments to functions. Normally, the running time  $T(n)$  of an algorithm is only defined when  $n$  is an integer, since for most algorithms, the size of the input is always an integer. For example, the recurrence describing the worst-case running time of MERGE-SORT is really

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1. \end{cases} \quad (4.2)$$

Boundary conditions represent another class of details that we typically ignore. Since the running time of an algorithm on a constant-sized input is a constant, the recurrences that arise from the running times of algorithms generally have  $T(n) = \Theta(1)$  for sufficiently small  $n$ . Consequently, for convenience, we shall generally omit statements of the boundary conditions of recurrences and assume that  $T(n)$  is constant for small  $n$ . For example, we normally state recurrence (4.1) as

$$T(n) = 2T(n/2) + \Theta(n), \quad (4.3)$$

without explicitly giving values for small  $n$ . The reason is that although changing the value of  $T(1)$  changes the solution to the recurrence, the solution typically doesn't change by more than a constant factor, so the order of growth is unchanged.

When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions. We forge ahead without these details and later determine whether or not they matter. They usually don't, but it is important to know when they do. Experience helps, and so do some theorems stating that these details don't affect the asymptotic bounds of many recurrences encountered in the analysis of algorithms (see Theorem 4.1). In this chapter, however, we shall address some of these details to show the fine points of recurrence solution methods.

## 4.1 The substitution method

The substitution method for solving recurrences entails two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

The name comes from the substitution of the guessed answer for the function when the inductive hypothesis is applied to smaller values. This method is powerful, but it obviously can be applied only in cases when it is easy to guess the form of the answer.

The substitution method can be used to establish either upper or lower bounds on a recurrence. As an example, let us determine an upper bound on the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n, \quad (4.4)$$

which is similar to recurrences (4.2) and (4.3). We guess that the solution is  $T(n) = O(n \lg n)$ . Our method is to prove that  $T(n) \leq cn \lg n$  for an appropriate choice of

the constant  $c > 0$ . We start by assuming that this bound holds for  $\lfloor n/2 \rfloor$ , that is, that  $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$ . Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n, \end{aligned}$$

where the last step holds as long as  $c \geq 1$ .

Mathematical induction now requires us to show that our solution holds for the boundary conditions. Typically, we do so by showing that the boundary conditions are suitable as base cases for the inductive proof. For the recurrence (4.4), we must show that we can choose the constant  $c$  large enough so that the bound  $T(n) \leq cn \lg n$  works for the boundary conditions as well. This requirement can sometimes lead to problems. Let us assume, for the sake of argument, that  $T(1) = 1$  is the sole boundary condition of the recurrence. Then for  $n = 1$ , the bound  $T(n) \leq cn \lg n$  yields  $T(1) \leq c1 \lg 1 = 0$ , which is at odds with  $T(1) = 1$ . Consequently, the base case of our inductive proof fails to hold.

This difficulty in proving an inductive hypothesis for a specific boundary condition can be easily overcome. For example, in the recurrence (4.4), we take advantage of asymptotic notation only requiring us to prove  $T(n) \leq cn \lg n$  for  $n \geq n_0$ , where  $n_0$  is a constant of our choosing. The idea is to remove the difficult boundary condition  $T(1) = 1$  from consideration in the inductive proof. Observe that for  $n > 3$ , the recurrence does not depend directly on  $T(1)$ . Thus, we can replace  $T(1)$  by  $T(2)$  and  $T(3)$  as the base cases in the inductive proof, letting  $n_0 = 2$ . Note that we make a distinction between the base case of the recurrence ( $n = 1$ ) and the base cases of the inductive proof ( $n = 2$  and  $n = 3$ ). We derive from the recurrence that  $T(2) = 4$  and  $T(3) = 5$ . The inductive proof that  $T(n) \leq cn \lg n$  for some constant  $c \geq 1$  can now be completed by choosing  $c$  large enough so that  $T(2) \leq c2 \lg 2$  and  $T(3) \leq c3 \lg 3$ . As it turns out, any choice of  $c \geq 2$  suffices for the base cases of  $n = 2$  and  $n = 3$  to hold. For most of the recurrences we shall examine, it is straightforward to extend boundary conditions to make the inductive assumption work for small  $n$ .

### Making a good guess

Unfortunately, there is no general way to guess the correct solutions to recurrences. Guessing a solution takes experience and, occasionally, creativity. Fortunately, though, there are some heuristics that can help you become a good guesser. You can also use recursion trees, which we shall see in Section 4.2, to generate good guesses.

If a recurrence is similar to one you have seen before, then guessing a similar solution is reasonable. As an example, consider the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n ,$$

which looks difficult because of the added “17” in the argument to  $T$  on the right-hand side. Intuitively, however, this additional term cannot substantially affect the solution to the recurrence. When  $n$  is large, the difference between  $T(\lfloor n/2 \rfloor)$  and  $T(\lfloor n/2 \rfloor + 17)$  is not that large: both cut  $n$  nearly evenly in half. Consequently, we make the guess that  $T(n) = O(n \lg n)$ , which you can verify as correct by using the substitution method (see Exercise 4.1-5).

Another way to make a good guess is to prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainty. For example, we might start with a lower bound of  $T(n) = \Omega(n)$  for the recurrence (4.4), since we have the term  $n$  in the recurrence, and we can prove an initial upper bound of  $T(n) = O(n^2)$ . Then, we can gradually lower the upper bound and raise the lower bound until we converge on the correct, asymptotically tight solution of  $T(n) = \Theta(n \lg n)$ .

### Subtleties

There are times when you can correctly guess at an asymptotic bound on the solution of a recurrence, but somehow the math doesn’t seem to work out in the induction. Usually, the problem is that the inductive assumption isn’t strong enough to prove the detailed bound. When you hit such a snag, revising the guess by subtracting a lower-order term often permits the math to go through.

Consider the recurrence

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 .$$

We guess that the solution is  $O(n)$ , and we try to show that  $T(n) \leq cn$  for an appropriate choice of the constant  $c$ . Substituting our guess in the recurrence, we obtain

$$\begin{aligned} T(n) &\leq c \lfloor n/2 \rfloor + c \lceil n/2 \rceil + 1 \\ &= cn + 1 , \end{aligned}$$

which does not imply  $T(n) \leq cn$  for any choice of  $c$ . It’s tempting to try a larger guess, say  $T(n) = O(n^2)$ , which can be made to work, but in fact, our guess that the solution is  $T(n) = O(n)$  is correct. In order to show this, however, we must make a stronger inductive hypothesis.

Intuitively, our guess is nearly right: we’re only off by the constant 1, a lower-order term. Nevertheless, mathematical induction doesn’t work unless we prove the exact form of the inductive hypothesis. We overcome our difficulty by *subtracting* a lower-order term from our previous guess. Our new guess is  $T(n) \leq cn - b$ ,

where  $b \geq 0$  is constant. We now have

$$\begin{aligned} T(n) &\leq (c \lfloor n/2 \rfloor - b) + (c \lceil n/2 \rceil - b) + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b, \end{aligned}$$

as long as  $b \geq 1$ . As before, the constant  $c$  must be chosen large enough to handle the boundary conditions.

Most people find the idea of subtracting a lower-order term counterintuitive. After all, if the math doesn't work out, shouldn't we be increasing our guess? The key to understanding this step is to remember that we are using mathematical induction: we can prove something stronger for a given value by assuming something stronger for smaller values.

### Avoiding pitfalls

It is easy to err in the use of asymptotic notation. For example, in the recurrence (4.4) we can falsely "prove"  $T(n) = O(n)$  by guessing  $T(n) \leq cn$  and then arguing

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor) + n \\ &\leq cn + n \\ &= O(n), \quad \Leftarrow \text{wrong!!} \end{aligned}$$

since  $c$  is a constant. The error is that we haven't proved the *exact form* of the inductive hypothesis, that is, that  $T(n) \leq cn$ .

### Changing variables

Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one you have seen before. As an example, consider the recurrence

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n,$$

which looks difficult. We can simplify this recurrence, though, with a change of variables. For convenience, we shall not worry about rounding off values, such as  $\sqrt{n}$ , to be integers. Renaming  $m = \lg n$  yields

$$T(2^m) = 2T(2^{m/2}) + m.$$

We can now rename  $S(m) = T(2^m)$  to produce the new recurrence

$$S(m) = 2S(m/2) + m,$$

which is very much like recurrence (4.4). Indeed, this new recurrence has the same solution:  $S(m) = O(m \lg m)$ . Changing back from  $S(m)$  to  $T(n)$ , we obtain  $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$ .

**Exercises****4.1-1**

Show that the solution of  $T(n) = T(\lceil n/2 \rceil) + 1$  is  $O(\lg n)$ .

**4.1-2**

We saw that the solution of  $T(n) = 2T(\lfloor n/2 \rfloor) + n$  is  $O(n \lg n)$ . Show that the solution of this recurrence is also  $\Omega(n \lg n)$ . Conclude that the solution is  $\Theta(n \lg n)$ .

**4.1-3**

Show that by making a different inductive hypothesis, we can overcome the difficulty with the boundary condition  $T(1) = 1$  for the recurrence (4.4) without adjusting the boundary conditions for the inductive proof.

**4.1-4**

Show that  $\Theta(n \lg n)$  is the solution to the “exact” recurrence (4.2) for merge sort.

**4.1-5**

Show that the solution to  $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$  is  $O(n \lg n)$ .

**4.1-6**

Solve the recurrence  $T(n) = 2T(\sqrt{n}) + 1$  by making a change of variables. Your solution should be asymptotically tight. Do not worry about whether values are integral.

---

**4.2 The recursion-tree method**

Although the substitution method can provide a succinct proof that a solution to a recurrence is correct, it is sometimes difficult to come up with a good guess. Drawing out a recursion tree, as we did in our analysis of the merge sort recurrence in Section 2.3.2, is a straightforward way to devise a good guess. In a *recursion tree*, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion. Recursion trees are particularly useful when the recurrence describes the running time of a divide-and-conquer algorithm.

A recursion tree is best used to generate a good guess, which is then verified by the substitution method. When using a recursion tree to generate a good guess, you can often tolerate a small amount of “sloppiness,” since you will be verifying your guess later on. If you are very careful when drawing out a recursion tree and summing the costs, however, you can use a recursion tree as a direct proof of a

solution to a recurrence. In this section, we will use recursion trees to generate good guesses, and in Section 4.4, we will use recursion trees directly to prove the theorem that forms the basis of the master method.

For example, let us see how a recursion tree would provide a good guess for the recurrence  $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ . We start by focusing on finding an upper bound for the solution. Because we know that floors and ceilings are usually insubstantial in solving recurrences (here's an example of sloppiness that we can tolerate), we create a recursion tree for the recurrence  $T(n) = 3T(n/4) + cn^2$ , having written out the implied constant coefficient  $c > 0$ .

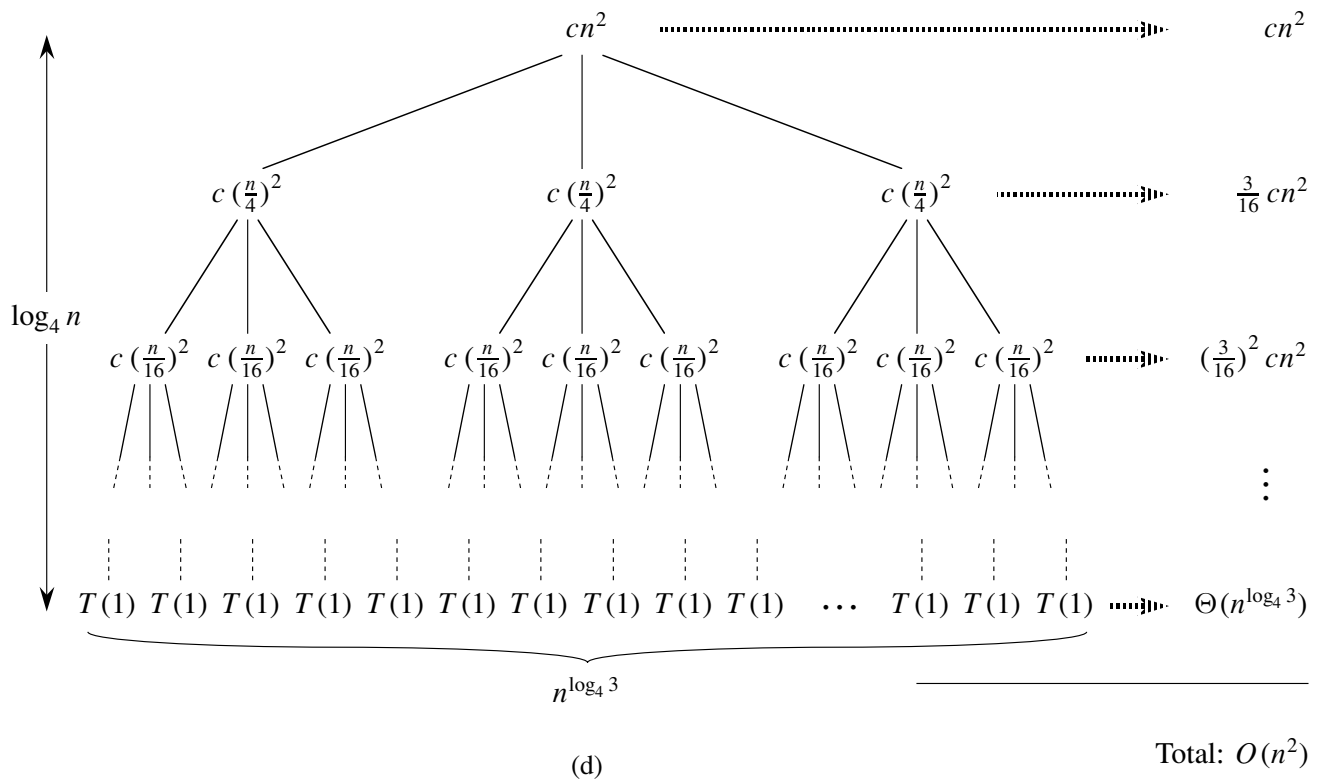
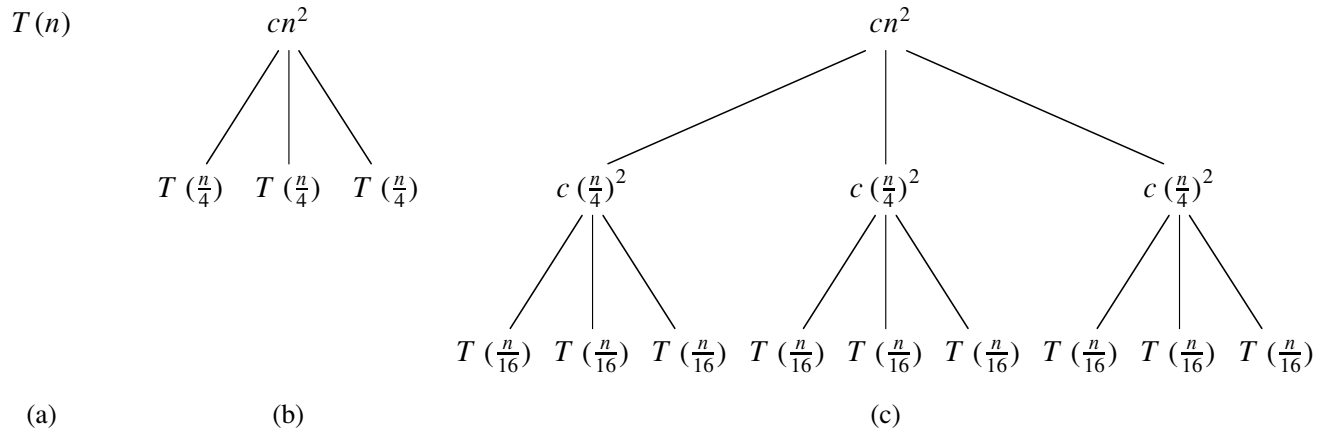
Figure 4.1 shows the derivation of the recursion tree for  $T(n) = 3T(n/4) + cn^2$ . For convenience, we assume that  $n$  is an exact power of 4 (another example of tolerable sloppiness). Part (a) of the figure shows  $T(n)$ , which is expanded in part (b) into an equivalent tree representing the recurrence. The  $cn^2$  term at the root represents the cost at the top level of recursion, and the three subtrees of the root represent the costs incurred by the subproblems of size  $n/4$ . Part (c) shows this process carried one step further by expanding each node with cost  $T(n/4)$  from part (b). The cost for each of the three children of the root is  $c(n/4)^2$ . We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence.

Because subproblem sizes decrease as we get further from the root, we eventually must reach a boundary condition. How far from the root do we reach one? The subproblem size for a node at depth  $i$  is  $n/4^i$ . Thus, the subproblem size hits  $n = 1$  when  $n/4^i = 1$  or, equivalently, when  $i = \log_4 n$ . Thus, the tree has  $\log_4 n + 1$  levels  $(0, 1, 2, \dots, \log_4 n)$ .

Next we determine the cost at each level of the tree. Each level has three times more nodes than the level above, and so the number of nodes at depth  $i$  is  $3^i$ . Because subproblem sizes reduce by a factor of 4 for each level we go down from the root, each node at depth  $i$ , for  $i = 0, 1, 2, \dots, \log_4 n - 1$ , has a cost of  $c(n/4^i)^2$ . Multiplying, we see that the total cost over all nodes at depth  $i$ , for  $i = 0, 1, 2, \dots, \log_4 n - 1$ , is  $3^i c(n/4^i)^2 = (3/16)^i cn^2$ . The last level, at depth  $\log_4 n$ , has  $3^{\log_4 n} = n^{\log_4 3}$  nodes, each contributing cost  $T(1)$ , for a total cost of  $n^{\log_4 3} T(1)$ , which is  $\Theta(n^{\log_4 3})$ .

Now we add up the costs over all levels to determine the cost for the entire tree:

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16} cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}). \end{aligned}$$



**Figure 4.1** The construction of a recursion tree for the recurrence  $T(n) = 3T(n/4) + cn^2$ . Part (a) shows  $T(n)$ , which is progressively expanded in (b)–(d) to form the recursion tree. The fully expanded tree in part (d) has height  $\log_4 n$  (it has  $\log_4 n + 1$  levels).



This last formula looks somewhat messy until we realize that we can again take advantage of small amounts of sloppiness and use an infinite decreasing geometric series as an upper bound. Backing up one step and applying equation (A.6), we have

$$\begin{aligned}
 T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
 &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
 &= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
 &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
 &= O(n^2).
 \end{aligned}$$

Thus, we have derived a guess of  $T(n) = O(n^2)$  for our original recurrence  $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ . In this example, the coefficients of  $cn^2$  form a decreasing geometric series and, by equation (A.6), the sum of these coefficients is bounded from above by the constant  $16/13$ . Since the root's contribution to the total cost is  $cn^2$ , the root contributes a constant fraction of the total cost. In other words, the total cost of the tree is dominated by the cost of the root.

In fact, if  $O(n^2)$  is indeed an upper bound for the recurrence (as we shall verify in a moment), then it must be a tight bound. Why? The first recursive call contributes a cost of  $\Theta(n^2)$ , and so  $\Omega(n^2)$  must be a lower bound for the recurrence.

Now we can use the substitution method to verify that our guess was correct, that is,  $T(n) = O(n^2)$  is an upper bound for the recurrence  $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ . We want to show that  $T(n) \leq dn^2$  for some constant  $d > 0$ . Using the same constant  $c > 0$  as before, we have

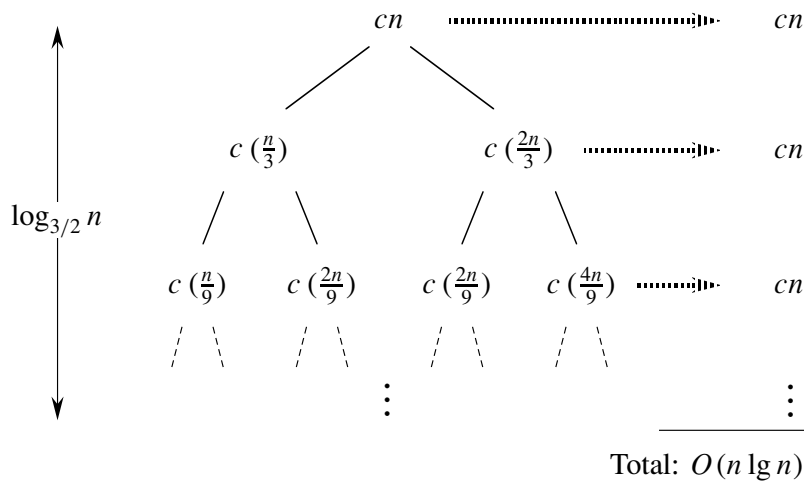
$$\begin{aligned}
 T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \\
 &\leq 3d \lfloor n/4 \rfloor^2 + cn^2 \\
 &\leq 3d(n/4)^2 + cn^2 \\
 &= \frac{3}{16} dn^2 + cn^2 \\
 &\leq dn^2,
 \end{aligned}$$

where the last step holds as long as  $d \geq (16/13)c$ .

As another, more intricate example, Figure 4.2 shows the recursion tree for

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

(Again, we omit floor and ceiling functions for simplicity.) As before, we let  $c$  represent the constant factor in the  $O(n)$  term. When we add the values across the



**Figure 4.2** A recursion tree for the recurrence  $T(n) = T(n/3) + T(2n/3) + cn$ .

levels of the recursion tree, we get a value of  $cn$  for every level. The longest path from the root to a leaf is  $n \rightarrow (2/3)n \rightarrow (2/3)^2n \rightarrow \dots \rightarrow 1$ . Since  $(2/3)^k n = 1$  when  $k = \log_{3/2} n$ , the height of the tree is  $\log_{3/2} n$ .

Intuitively, we expect the solution to the recurrence to be at most the number of levels times the cost of each level, or  $O(cn \log_{3/2} n) = O(n \lg n)$ . The total cost is evenly distributed throughout the levels of the recursion tree. There is a complication here: we have yet to consider the cost of the leaves. If this recursion tree were a complete binary tree of height  $\log_{3/2} n$ , there would be  $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$  leaves. Since the cost of each leaf is a constant, the total cost of all leaves would then be  $\Theta(n^{\log_{3/2} 2})$ , which is  $\omega(n \lg n)$ . This recursion tree is not a complete binary tree, however, and so it has fewer than  $n^{\log_{3/2} 2}$  leaves. Moreover, as we go down from the root, more and more internal nodes are absent. Consequently, not all levels contribute a cost of exactly  $cn$ ; levels toward the bottom contribute less. We could work out an accurate accounting of all costs, but remember that we are just trying to come up with a guess to use in the substitution method. Let us tolerate the sloppiness and attempt to show that a guess of  $O(n \lg n)$  for the upper bound is correct.

Indeed, we can use the substitution method to verify that  $O(n \lg n)$  is an upper bound for the solution to the recurrence. We show that  $T(n) \leq dn \lg n$ , where  $d$  is a suitable positive constant. We have

$$\begin{aligned}
T(n) &\leq T(n/3) + T(2n/3) + cn \\
&\leq d(n/3) \lg(n/3) + d(2n/3) \lg(2n/3) + cn \\
&= (d(n/3) \lg n - d(n/3) \lg 3) \\
&\quad + (d(2n/3) \lg n - d(2n/3) \lg(3/2)) + cn \\
&= dn \lg n - d((n/3) \lg 3 + (2n/3) \lg(3/2)) + cn \\
&= dn \lg n - d((n/3) \lg 3 + (2n/3) \lg 3 - (2n/3) \lg 2) + cn \\
&= dn \lg n - dn(\lg 3 - 2/3) + cn \\
&\leq dn \lg n,
\end{aligned}$$

as long as  $d \geq c/(\lg 3 - 2/3)$ . Thus, we did not have to perform a more accurate accounting of costs in the recursion tree.

### Exercises

#### 4.2-1

Use a recursion tree to determine a good asymptotic upper bound on the recurrence  $T(n) = 3T(\lfloor n/2 \rfloor) + n$ . Use the substitution method to verify your answer.

#### 4.2-2

Argue that the solution to the recurrence  $T(n) = T(n/3) + T(2n/3) + cn$ , where  $c$  is a constant, is  $\Omega(n \lg n)$  by appealing to a recursion tree.

#### 4.2-3

Draw the recursion tree for  $T(n) = 4T(\lfloor n/2 \rfloor) + cn$ , where  $c$  is a constant, and provide a tight asymptotic bound on its solution. Verify your bound by the substitution method.

#### 4.2-4

Use a recursion tree to give an asymptotically tight solution to the recurrence  $T(n) = T(n-a) + T(a) + cn$ , where  $a \geq 1$  and  $c > 0$  are constants.

#### 4.2-5

Use a recursion tree to give an asymptotically tight solution to the recurrence  $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$ , where  $\alpha$  is a constant in the range  $0 < \alpha < 1$  and  $c > 0$  is also a constant.

---

### 4.3 The master method

The master method provides a “cookbook” method for solving recurrences of the form

$$T(n) = aT(n/b) + f(n), \quad (4.5)$$

where  $a \geq 1$  and  $b > 1$  are constants and  $f(n)$  is an asymptotically positive function. The master method requires memorization of three cases, but then the solution of many recurrences can be determined quite easily, often without pencil and paper.

The recurrence (4.5) describes the running time of an algorithm that divides a problem of size  $n$  into  $a$  subproblems, each of size  $n/b$ , where  $a$  and  $b$  are positive constants. The  $a$  subproblems are solved recursively, each in time  $T(n/b)$ . The cost of dividing the problem and combining the results of the subproblems is described by the function  $f(n)$ . (That is, using the notation from Section 2.3.2,  $f(n) = D(n) + C(n)$ .) For example, the recurrence arising from the MERGE-SORT procedure has  $a = 2$ ,  $b = 2$ , and  $f(n) = \Theta(n)$ .

As a matter of technical correctness, the recurrence isn’t actually well defined because  $n/b$  might not be an integer. Replacing each of the  $a$  terms  $T(n/b)$  with either  $T(\lfloor n/b \rfloor)$  or  $T(\lceil n/b \rceil)$  doesn’t affect the asymptotic behavior of the recurrence, however. (We’ll prove this in the next section.) We normally find it convenient, therefore, to omit the floor and ceiling functions when writing divide-and-conquer recurrences of this form.

#### The master theorem

The master method depends on the following theorem.

##### **Theorem 4.1 (Master theorem)**

Let  $a \geq 1$  and  $b > 1$  be constants, let  $f(n)$  be a function, and let  $T(n)$  be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret  $n/b$  to mean either  $\lfloor n/b \rfloor$  or  $\lceil n/b \rceil$ . Then  $T(n)$  can be bounded asymptotically as follows.

1. If  $f(n) = O(n^{\log_b a - \epsilon})$  for some constant  $\epsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$ .
2. If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \lg n)$ .
3. If  $f(n) = \Omega(n^{\log_b a + \epsilon})$  for some constant  $\epsilon > 0$ , and if  $af(n/b) \leq cf(n)$  for some constant  $c < 1$  and all sufficiently large  $n$ , then  $T(n) = \Theta(f(n))$ . ■

Before applying the master theorem to some examples, let's spend a moment trying to understand what it says. In each of the three cases, we are comparing the function  $f(n)$  with the function  $n^{\log_b a}$ . Intuitively, the solution to the recurrence is determined by the larger of the two functions. If, as in case 1, the function  $n^{\log_b a}$  is the larger, then the solution is  $T(n) = \Theta(n^{\log_b a})$ . If, as in case 3, the function  $f(n)$  is the larger, then the solution is  $T(n) = \Theta(f(n))$ . If, as in case 2, the two functions are the same size, we multiply by a logarithmic factor, and the solution is  $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$ .

Beyond this intuition, there are some technicalities that must be understood. In the first case, not only must  $f(n)$  be smaller than  $n^{\log_b a}$ , it must be *polynomially* smaller. That is,  $f(n)$  must be asymptotically smaller than  $n^{\log_b a}$  by a factor of  $n^\epsilon$  for some constant  $\epsilon > 0$ . In the third case, not only must  $f(n)$  be larger than  $n^{\log_b a}$ , it must be polynomially larger and in addition satisfy the “regularity” condition that  $af(n/b) \leq cf(n)$ . This condition is satisfied by most of the polynomially bounded functions that we shall encounter.

It is important to realize that the three cases do not cover all the possibilities for  $f(n)$ . There is a gap between cases 1 and 2 when  $f(n)$  is smaller than  $n^{\log_b a}$  but not polynomially smaller. Similarly, there is a gap between cases 2 and 3 when  $f(n)$  is larger than  $n^{\log_b a}$  but not polynomially larger. If the function  $f(n)$  falls into one of these gaps, or if the regularity condition in case 3 fails to hold, the master method cannot be used to solve the recurrence.

### Using the master method

To use the master method, we simply determine which case (if any) of the master theorem applies and write down the answer.

As a first example, consider

$$T(n) = 9T(n/3) + n.$$

For this recurrence, we have  $a = 9$ ,  $b = 3$ ,  $f(n) = n$ , and thus we have that  $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$ . Since  $f(n) = O(n^{\log_3 9 - \epsilon})$ , where  $\epsilon = 1$ , we can apply case 1 of the master theorem and conclude that the solution is  $T(n) = \Theta(n^2)$ .

Now consider

$$T(n) = T(2n/3) + 1,$$

in which  $a = 1$ ,  $b = 3/2$ ,  $f(n) = 1$ , and  $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$ . Case 2 applies, since  $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$ , and thus the solution to the recurrence is  $T(n) = \Theta(\lg n)$ .

For the recurrence

$$T(n) = 3T(n/4) + n \lg n,$$

we have  $a = 3$ ,  $b = 4$ ,  $f(n) = n \lg n$ , and  $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$ . Since  $f(n) = \Omega(n^{\log_4 3 + \epsilon})$ , where  $\epsilon \approx 0.2$ , case 3 applies if we can show that the regular-

ity condition holds for  $f(n)$ . For sufficiently large  $n$ ,  $af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n)$  for  $c = 3/4$ . Consequently, by case 3, the solution to the recurrence is  $T(n) = \Theta(n \lg n)$ .

The master method does not apply to the recurrence

$$T(n) = 2T(n/2) + n \lg n,$$

even though it has the proper form:  $a = 2$ ,  $b = 2$ ,  $f(n) = n \lg n$ , and  $n^{\log_b a} = n$ . It might seem that case 3 should apply, since  $f(n) = n \lg n$  is asymptotically larger than  $n^{\log_b a} = n$ . The problem is that it is not *polynomially* larger. The ratio  $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$  is asymptotically less than  $n^\epsilon$  for any positive constant  $\epsilon$ . Consequently, the recurrence falls into the gap between case 2 and case 3. (See Exercise 4.4-2 for a solution.)

## Exercises

### 4.3-1

Use the master method to give tight asymptotic bounds for the following recurrences.

a.  $T(n) = 4T(n/2) + n$ .

b.  $T(n) = 4T(n/2) + n^2$ .

c.  $T(n) = 4T(n/2) + n^3$ .

### 4.3-2

The recurrence  $T(n) = 7T(n/2) + n^2$  describes the running time of an algorithm  $A$ . A competing algorithm  $A'$  has a running time of  $T'(n) = aT'(n/4) + n^2$ . What is the largest integer value for  $a$  such that  $A'$  is asymptotically faster than  $A$ ?

### 4.3-3

Use the master method to show that the solution to the binary-search recurrence  $T(n) = T(n/2) + \Theta(1)$  is  $T(n) = \Theta(\lg n)$ . (See Exercise 2.3-5 for a description of binary search.)

### 4.3-4

Can the master method be applied to the recurrence  $T(n) = 4T(n/2) + n^2 \lg n$ ? Why or why not? Give an asymptotic upper bound for this recurrence.

### 4.3-5 ★

Consider the regularity condition  $af(n/b) \leq cf(n)$  for some constant  $c < 1$ , which is part of case 3 of the master theorem. Give an example of constants  $a \geq 1$  and  $b > 1$  and a function  $f(n)$  that satisfies all the conditions in case 3 of the master theorem except the regularity condition.