4.2-5

Show how to multiply the complex numbers $a + bi$ and $c + di$ using only three multiplications of real numbers. The algorithm should take a, b, c, and d as input and produce the real component $ac - bd$ and the imaginary component $ad + bc$ separately.

4.2-6

Suppose that you have a $\Theta(n^\alpha)$-time algorithm for squaring $n \times n$ matrices, where $\alpha \geq 2$. Show how to use that algorithm to multiply two different $n \times n$ matrices in $\Theta(n^\alpha)$ time.

4.3 The substitution method for solving recurrences

Now that you have seen how recurrences characterize the running times of divide-and-conquer algorithms, let's learn how to solve them. We start in this section with the ***substitution method***, which is the most general of the four methods in this chapter. The substitution method comprises two steps:

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

To apply the inductive hypothesis, you substitute the guessed solution for the function on smaller values—hence the name "substitution method." This method is powerful, but you must guess the form of the answer. Although generating a good guess might seem difficult, a little practice can quickly improve your intuition.

You can use the substitution method to establish either an upper or a lower bound on a recurrence. It's usually best not to try to do both at the same time. That is, rather than trying to prove a Θ-bound directly, first prove an O-bound, and then prove an Ω-bound. Together, they give you a Θ-bound (Theorem 3.1 on page 56).

As an example of the substitution method, let's determine an asymptotic upper bound on the recurrence:

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n) . \tag{4.11}$$

This recurrence is similar to recurrence (2.3) on page 41 for merge sort, except for the floor function, which ensures that $T(n)$ is defined over the integers. Let's guess that the asymptotic upper bound is the same—$T(n) = O(n \lg n)$—and use the substitution method to prove it.

We'll adopt the inductive hypothesis that $T(n) \leq cn \lg n$ for all $n \geq n_0$, where we'll choose the specific constants $c > 0$ and $n_0 > 0$ later, after we see what constraints they need to obey. If we can establish this inductive hypothesis, we can conclude that $T(n) = O(n \lg n)$. It would be dangerous to use $T(n) = O(n \lg n)$ as the inductive hypothesis because the constants matter, as we'll see in a moment in our discussion of pitfalls.

Assume by induction that this bound holds for all numbers at least as big as n_0 and less than n. In particular, therefore, if $n \geq 2n_0$, it holds for $\lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Substituting into recurrence (4.11)—hence the name "substitution" method—yields

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

where the last step holds if we constrain the constants n_0 and c to be sufficiently large that for $n \geq 2n_0$, the quantity cn dominates the anonymous function hidden by the $\Theta(n)$ term.

We've shown that the inductive hypothesis holds for the inductive case, but we also need to prove that the inductive hypothesis holds for the base cases of the induction, that is, that $T(n) \leq cn \lg n$ when $n_0 \leq n < 2n_0$. As long as $n_0 > 1$ (a new constraint on n_0), we have $\lg n > 0$, which implies that $n \lg n > 0$. So let's pick $n_0 = 2$. Since the base case of recurrence (4.11) is not stated explicitly, by our convention, $T(n)$ is algorithmic, which means that $T(2)$ and $T(3)$ are constant (as they should be if they describe the worst-case running time of any real program on inputs of size 2 or 3). Picking $c = \max\{T(2), T(3)\}$ yields $T(2) \leq c < (2 \lg 2)c$ and $T(3) \leq c < (3 \lg 3)c$, establishing the inductive hypothesis for the base cases.

Thus, we have $T(n) \leq cn \lg n$ for all $n \geq 2$, which implies that the solution to recurrence (4.11) is $T(n) = O(n \lg n)$.

In the algorithms literature, people rarely carry out their substitution proofs to this level of detail, especially in their treatment of base cases. The reason is that for most algorithmic divide-and-conquer recurrences, the base cases are all handled in pretty much the same way. You ground the induction on a range of values from a convenient positive constant n_0 up to some constant $n'_0 > n_0$ such that for $n \geq n'_0$, the recurrence always bottoms out in a constant-sized base case between n_0 and n'_0. (This example used $n'_0 = 2n_0$.) Then, it's usually apparent, without spelling out the details, that with a suitably large choice of the leading constant (such as c for this example), the inductive hypothesis can be made to hold for all the values in the range from n_0 to n'_0.

Making a good guess

Unfortunately, there is no general way to correctly guess the tightest asymptotic solution to an arbitrary recurrence. Making a good guess takes experience and, occasionally, creativity. Fortunately, learning some recurrence-solving heuristics, as well as playing around with recurrences to gain experience, can help you become a good guesser. You can also use recursion trees, which we'll see in Section 4.4, to help generate good guesses.

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

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

defined on the reals. This recurrence looks somewhat like the merge-sort recurrence (2.3), but it's more complicated because of the added "17" in the argument to T on the right-hand side. Intuitively, however, this additional term shouldn't substantially affect the solution to the recurrence. When n is large, the relative difference between $n/2$ and $n/2 + 17$ is not that large: both cut n nearly in half. Consequently, it makes sense to guess that $T(n) = O(n \lg n)$, which you can verify is correct using the substitution method (see Exercise 4.3-1).

Another way to make a good guess is to determine loose upper and lower bounds on the recurrence and then reduce your range of uncertainty. For example, you might start with a lower bound of $T(n) = \Omega(n)$ for recurrence (4.11), since the recurrence includes the term $\Theta(n)$, and you can prove an initial upper bound of $T(n) = O(n^2)$. Then split your time between trying to lower the upper bound and trying to raise the lower bound until you converge on the correct, asymptotically tight solution, which in this case is $T(n) = \Theta(n \lg n)$.

A trick of the trade: subtracting a low-order term

Sometimes, you might correctly guess a tight asymptotic bound on the solution of a recurrence, but somehow the math fails to work out in the induction proof. The problem frequently turns out to be that the inductive assumption is not strong enough. The trick to resolving this problem is to revise your guess by *subtracting* a lower-order term when you hit such a snag. The math then often goes through.

Consider the recurrence

$$T(n) = 2T(n/2) + \Theta(1) \tag{4.12}$$

defined on the reals. Let's guess that the solution is $T(n) = O(n)$ and try to show that $T(n) \le cn$ for $n \ge n_0$, where we choose the constants $c, n_0 > 0$ suitably. Substituting our guess into the recurrence, we obtain

$$\begin{aligned} T(n) &\le 2(c(n/2)) + \Theta(1) \\ &= cn + \Theta(1) , \end{aligned}$$

which, unfortunately, does not imply that $T(n) \le cn$ for *any* choice of c. We might be tempted to try a larger guess, say $T(n) = O(n^2)$. Although this larger guess works, it provides only a loose upper bound. It turns out that our original guess of $T(n) = O(n)$ is correct and tight. In order to show that it is correct, however, we must strengthen our inductive hypothesis.

Intuitively, our guess is nearly right: we are off only by $\Theta(1)$, a lower-order term. Nevertheless, mathematical induction requires us to prove the *exact* form of the inductive hypothesis. Let's try our trick of subtracting a lower-order term from our previous guess: $T(n) \le cn - d$, where $d \ge 0$ is a constant. We now have

$$\begin{aligned} T(n) &\le 2(c(n/2) - d) + \Theta(1) \\ &= cn - 2d + \Theta(1) \\ &\le cn - d - (d - \Theta(1)) \\ &\le cn - d \end{aligned}$$

as long as we choose d to be larger than the anonymous upper-bound constant hidden by the Θ-notation. Subtracting a lower-order term works! Of course, we must not forget to handle the base case, which is to choose the constant c large enough that $cn - d$ dominates the implicit base cases.

You might find the idea of subtracting a lower-order term to be counterintuitive. After all, if the math doesn't work out, shouldn't you increase your guess? Not necessarily! When the recurrence contains more than one recursive invocation (recurrence (4.12) contains two), if you add a lower-order term to the guess, then you end up adding it once for each of the recursive invocations. Doing so takes you even further away from the inductive hypothesis. On the other hand, if you subtract a lower-order term from the guess, then you get to subtract it once for each of the recursive invocations. In the above example, we subtracted the constant d twice because the coefficient of $T(n/2)$ is 2. We ended up with the inequality $T(n) \le cn - d - (d - \Theta(1))$, and we readily found a suitable value for d.

Avoiding pitfalls

Avoid using asymptotic notation in the inductive hypothesis for the substitution method because it's error prone. For example, for recurrence (4.11), we can falsely "prove" that $T(n) = O(n)$ if we unwisely adopt $T(n) = O(n)$ as our inductive hypothesis:

$$\begin{aligned} T(n) &\le 2 \cdot O(\lfloor n/2 \rfloor) + \Theta(n) \\ &= 2 \cdot O(n) + \Theta(n) \\ &= O(n) . \qquad \Longleftarrow \textit{wrong!} \end{aligned}$$

The problem with this reasoning is that the constant hidden by the O-notation changes. We can expose the fallacy by repeating the "proof" using an explicit constant. For the inductive hypothesis, assume that $T(n) \le cn$ for all $n \ge n_0$, where $c, n_0 > 0$ are constants. Repeating the first two steps in the inequality chain yields

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor) + \Theta(n) \\ &\le cn + \Theta(n) . \end{aligned}$$

Now, indeed $cn + \Theta(n) = O(n)$, but the constant hidden by the O-notation must be larger than c because the anonymous function hidden by the $\Theta(n)$ is asymptotically positive. We cannot take the third step to conclude that $cn + \Theta(n) \le cn$, thus exposing the fallacy.

When using the substitution method, or more generally mathematical induction, you must be careful that the constants hidden by any asymptotic notation are the same constants throughout the proof. Consequently, it's best to avoid asymptotic notation in your inductive hypothesis and to name constants explicitly.

Here's another fallacious use of the substitution method to show that the solution to recurrence (4.11) is $T(n) = O(n)$. We guess $T(n) \le cn$ and then argue

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor) + \Theta(n) \\ &\le cn + \Theta(n) \\ &= O(n) , \qquad \Longleftarrow \textit{wrong!} \end{aligned}$$

since c is a positive constant. The mistake stems from the difference between our goal—to prove that $T(n) = O(n)$—and our inductive hypothesis—to prove that $T(n) \le cn$. When using the substitution method, or in any inductive proof, you must prove the *exact* statement of the inductive hypothesis. In this case, we must explicitly prove that $T(n) \le cn$ to show that $T(n) = O(n)$.

Exercises

4.3-1

Use the substitution method to show that each of the following recurrences defined on the reals has the asymptotic solution specified:

a. $T(n) = T(n-1) + n$ has solution $T(n) = O(n^2)$.

b. $T(n) = T(n/2) + \Theta(1)$ has solution $T(n) = O(\lg n)$.

c. $T(n) = 2T(n/2) + n$ has solution $T(n) = \Theta(n \lg n)$.

d. $T(n) = 2T(n/2 + 17) + n$ has solution $T(n) = O(n \lg n)$.

e. $T(n) = 2T(n/3) + \Theta(n)$ has solution $T(n) = \Theta(n)$.

f. $T(n) = 4T(n/2) + \Theta(n)$ has solution $T(n) = \Theta(n^2)$.

4.3-2
The solution to the recurrence $T(n) = 4T(n/2) + n$ turns out to be $T(n) = \Theta(n^2)$. Show that a substitution proof with the assumption $T(n) \le cn^2$ fails. Then show how to subtract a lower-order term to make a substitution proof work.

4.3-3
The recurrence $T(n) = 2T(n-1) + 1$ has the solution $T(n) = O(2^n)$. Show that a substitution proof fails with the assumption $T(n) \le c2^n$, where $c > 0$ is constant. Then show how to subtract a lower-order term to make a substitution proof work.

4.4 The recursion-tree method for solving recurrences

Although you can use the substitution method to prove that a solution to a recurrence is correct, you might have trouble coming 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, can help. In a ***recursion tree***, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. You typically sum the costs within each level of the tree to obtain the per-level costs, and then you sum all the per-level costs to determine the total cost of all levels of the recursion. Sometimes, however, adding up the total cost takes more creativity.

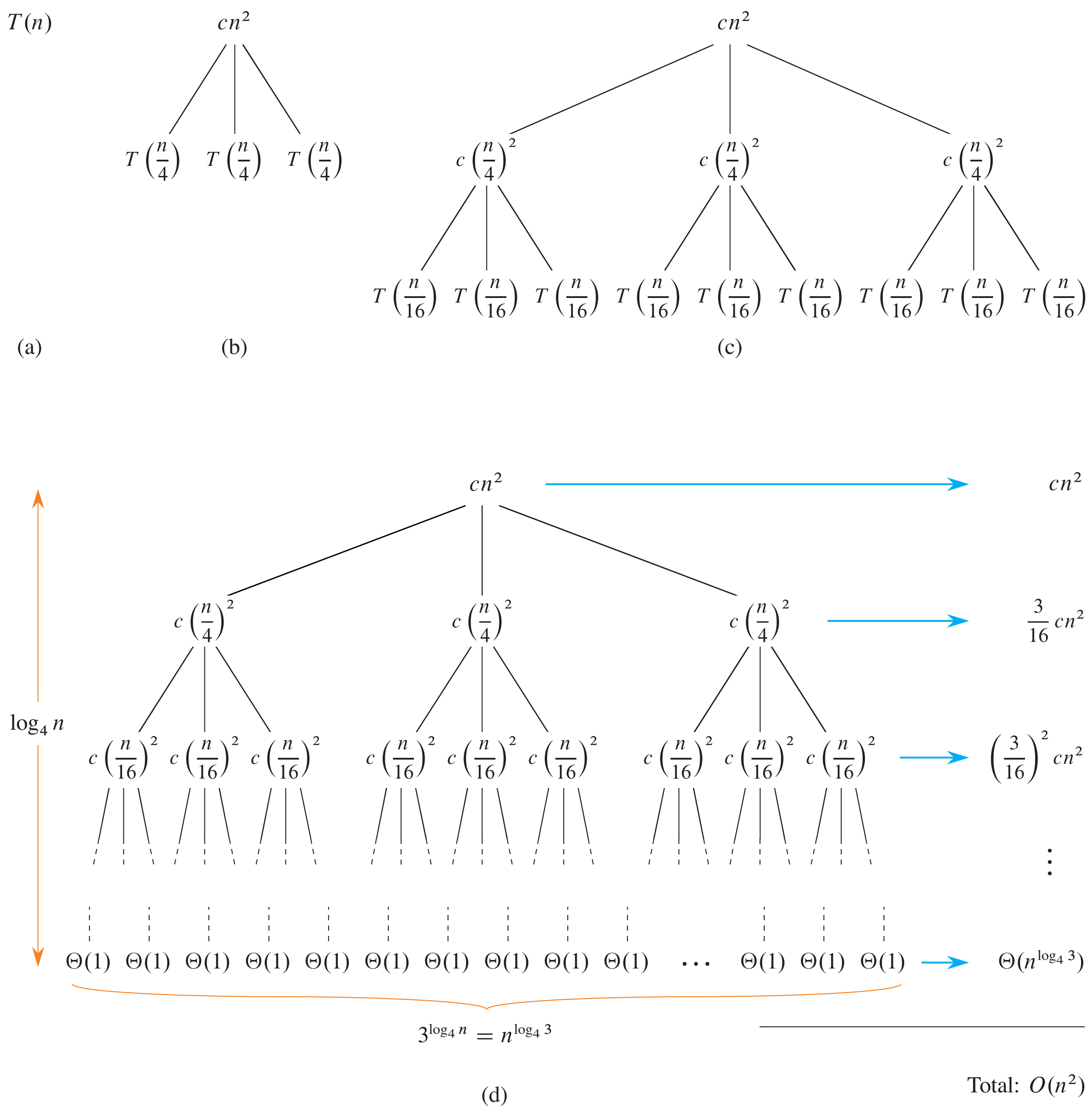
A recursion tree is best used to generate intuition for a good guess, which you can then verify by the substitution method. If you are meticulous when drawing out a recursion tree and summing the costs, however, you can use a recursion tree as a direct proof of a solution to a recurrence. But if you use it only to generate a good guess, you can often tolerate a small amount of "sloppiness," which can simplify the math. When you verify your guess with the substitution method later on, your math should be precise. This section demonstrates how you can use recursion trees to solve recurrences, generate good guesses, and gain intuition for recurrences.

An illustrative example

Let's see how a recursion tree can provide a good guess for an upper-bound solution to the recurrence

$$T(n) = 3T(n/4) + \Theta(n^2) . \tag{4.13}$$

Figure 4.1 shows how to derive the recursion tree for $T(n) = 3T(n/4) + cn^2$, where the constant $c > 0$ is the upper-bound constant in the $\Theta(n^2)$ term. Part (a) of the figure shows $T(n)$, which part (b) expands 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



$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
$c\left(\frac{n}{16}\right)^2$
$\Theta(1)$
(a)
(b)
(c)
$\log_4 n$
$\frac{3}{16}cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\Theta(n^{\log_4 3})$
$3^{\log_4 n} = n^{\log_4 3}$
(d)
Total: $O(n^2)$

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

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 by a factor of 4 every time we go down one level, the recursion must eventually bottom out in a base case where $n < n_0$. By convention, the base case is $T(n) = \Theta(1)$ for $n < n_0$, where $n_0 > 0$ is any threshold constant sufficiently large that the recurrence is well defined. For the purpose of intuition, however, let's simplify the math a little. Let's assume that n is an exact power of 4 and that the base case is $T(1) = \Theta(1)$. As it turns out, these assumptions don't affect the asymptotic solution.

What's the height of the recursion tree? The subproblem size for a node at depth i is $n/4^i$. As we descend the tree from the root, the subproblem size hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$. Thus, the tree has internal nodes at depths $0, 1, 2, \ldots, \log_4 n - 1$ and leaves at depth $\log_4 n$.

Part (d) of Figure 4.1 shows the cost at each level of the tree. Each level has three times as many nodes as 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 further from the root, each internal node at depth $i = 0, 1, 2, \ldots, \log_4 n - 1$ has a cost of $c(n/4^i)^2$. Multiplying, we see that the total cost of all nodes at a given depth i is $3^i c(n/4^i)^2 = (3/16)^i cn^2$. The bottom level, at depth $\log_4 n$, contains $3^{\log_4 n} = n^{\log_4 3}$ leaves (using equation (3.21) on page 66). Each leaf contributes $\Theta(1)$, leading to a total leaf cost of $\Theta(n^{\log_4 3})$.

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

$$\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \qquad \text{(by equation (A.7) on page 1142)} \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) \qquad (\Theta(n^{\log_4 3}) = O(n^{0.8}) = O(n^2)) .
\end{aligned}$$

We've derived the guess of $T(n) = O(n^2)$ for the original recurrence. In this example, the coefficients of cn^2 form a decreasing geometric series. By equation (A.7), 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 cost of the root dominates the total cost of the tree.

In fact, if $O(n^2)$ is indeed an upper bound for the recurrence (as we'll 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.

Let's now use the substitution method to verify that our guess is correct, namely, that $T(n) = O(n^2)$ is an upper bound for the recurrence $T(n) = 3T(n/4) + \Theta(n^2)$. We want to show that $T(n) \le dn^2$ for some constant $d > 0$. Using the same constant $c > 0$ as before, we have

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

where the last step holds if we choose $d \ge (16/13)c$.

For the base case of the induction, let $n_0 > 0$ be a sufficiently large threshold constant that the recurrence is well defined when $T(n) = \Theta(1)$ for $n < n_0$. We can pick d large enough that d dominates the constant hidden by the Θ, in which case $dn^2 \ge d \ge T(n)$ for $1 \le n < n_0$, completing the proof of the base case.

The substitution proof we just saw involves two named constants, c and d. We named c and used it to stand for the upper-bound constant hidden and guaranteed to exist by the Θ-notation. We cannot pick c arbitrarily—it's given to us—although, for any such c, any constant $c' \ge c$ also suffices. We also named d, but we were free to choose any value for it that fit our needs. In this example, the value of d happened to depend on the value of c, which is fine, since d is constant if c is constant.

An irregular example

Let's find an asymptotic upper bound for another, more irregular, example. Figure 4.2 shows the recursion tree for the recurrence

$$T(n) = T(n/3) + T(2n/3) + \Theta(n) . \tag{4.14}$$

This recursion tree is unbalanced, with different root-to-leaf paths having different lengths. Going left at any node produces a subproblem of one-third the size, and going right produces a subproblem of two-thirds the size. Let $n_0 > 0$ be the implicit threshold constant such that $T(n) = \Theta(1)$ for $0 < n < n_0$, and let c represent the upper-bound constant hidden by the $\Theta(n)$ term for $n \ge n_0$. There are actually two n_0 constants here—one for the threshold in the recurrence, and the other for the threshold in the Θ-notation, so we'll let n_0 be the larger of the two constants.

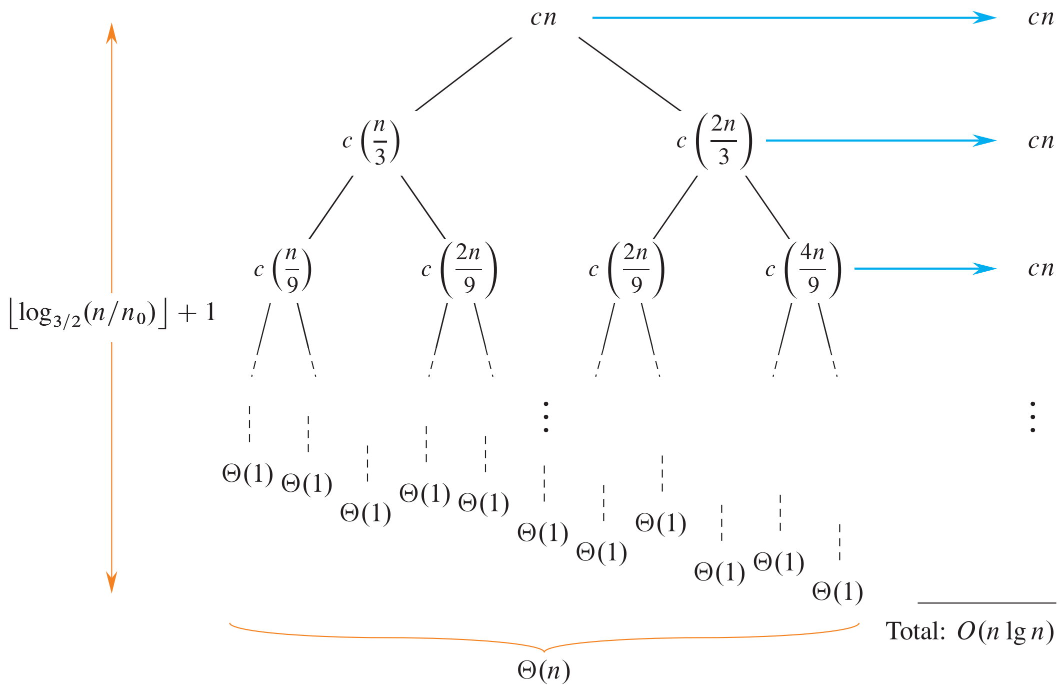

cn
$c\left(\frac{n}{3}\right)$
$c\left(\frac{2n}{3}\right)$
$c\left(\frac{n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{4n}{9}\right)$
cn
cn
cn
$\lfloor \log_{3/2}(n/n_0) \rfloor + 1$
$\Theta(1)$
$\Theta(n)$
Total: $O(n \lg n)$

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

The height of the tree runs down the right edge of the tree, corresponding to subproblems of sizes $n, (2/3)n, (4/9)n, \ldots, \Theta(1)$ with costs bounded by $cn, c(2n/3), c(4n/9), \ldots, \Theta(1)$, respectively. We hit the rightmost leaf when $(2/3)^h n < n_0 \le (2/3)^{h-1} n$, which happens when $h = \lfloor \log_{3/2}(n/n_0) \rfloor + 1$ since, applying the floor bounds in equation (3.2) on page 64 with $x = \log_{3/2}(n/n_0)$, we have $(2/3)^h n = (2/3)^{\lfloor x \rfloor + 1} n < (2/3)^x n = (n_0/n)n = n_0$ and $(2/3)^{h-1} n = (2/3)^{\lfloor x \rfloor} n > (2/3)^x n = (n_0/n)n = n_0$. Thus, the height of the tree is $h = \Theta(\lg n)$.

We're now in a position to understand the upper bound. Let's postpone dealing with the leaves for a moment. Summing the costs of internal nodes across each level, we have at most cn per level times the $\Theta(\lg n)$ tree height for a total cost of $O(n \lg n)$ for all internal nodes.

It remains to deal with the leaves of the recursion tree, which represent base cases, each costing $\Theta(1)$. How many leaves are there? It's tempting to upper-bound their number by the number of leaves in a complete binary tree of height $h = \lfloor \log_{3/2}(n/n_0) \rfloor + 1$, since the recursion tree is contained within such a complete binary tree. But this approach turns out to give us a poor bound. The complete binary tree has 1 node at the root, 2 nodes at depth 1, and generally 2^k nodes at depth k. Since the height is $h = \lfloor \log_{3/2} n \rfloor + 1$, there are

$2^h = 2^{\lfloor \log_{3/2} n \rfloor + 1} \le 2n^{\log_{3/2} 2}$ leaves in the complete binary tree, which is an upper bound on the number of leaves in the recursion tree. Because the cost of each leaf is $\Theta(1)$, this analysis says that the total cost of all leaves in the recursion tree is $O(n^{\log_{3/2} 2}) = O(n^{1.71})$, which is an asymptotically greater bound than the $O(n \lg n)$ cost of all internal nodes. In fact, as we're about to see, this bound is not tight. The cost of all leaves in the recursion tree is $O(n)$—asymptotically *less* than $O(n \lg n)$. In other words, the cost of the internal nodes dominates the cost of the leaves, not vice versa.

Rather than analyzing the leaves, we could quit right now and prove by substitution that $T(n) = \Theta(n \lg n)$. This approach works (see Exercise 4.4-3), but it's instructive to understand how many leaves this recursion tree has. You may see recurrences for which the cost of leaves dominates the cost of internal nodes, and then you'll be in better shape if you've had some experience analyzing the number of leaves.

To figure out how many leaves there really are, let's write a recurrence $L(n)$ for the number of leaves in the recursion tree for $T(n)$. Since all the leaves in $T(n)$ belong either to the left subtree or the right subtree of the root, we have

$$L(n) = \begin{cases} 1 & \text{if } n < n_0 , \\ L(n/3) + L(2n/3) & \text{if } n \ge n_0 . \end{cases} \tag{4.15}$$

This recurrence is similar to recurrence (4.14), but it's missing the $\Theta(n)$ term, and it contains an explicit base case. Because this recurrence omits the $\Theta(n)$ term, it is much easier to solve. Let's apply the substitution method to show that it has solution $L(n) = O(n)$. Using the inductive hypothesis $L(n) \le dn$ for some constant $d > 0$, and assuming that the inductive hypothesis holds for all values less than n, we have

$$\begin{aligned} L(n) &= L(n/3) + L(2n/3) \\ &\le dn/3 + 2(dn)/3 \\ &\le dn , \end{aligned}$$

which holds for any $d > 0$. We can now choose d large enough to handle the base case $L(n) = 1$ for $0 < n < n_0$, for which $d = 1$ suffices, thereby completing the substitution method for the upper bound on leaves. (Exercise 4.4-2 asks you to prove that $L(n) = \Theta(n)$.)

Returning to recurrence (4.14) for $T(n)$, it now becomes apparent that the total cost of leaves over all levels must be $L(n) \cdot \Theta(1) = \Theta(n)$. Since we have derived the bound of $O(n \lg n)$ on the cost of the internal nodes, it follows that the solution to recurrence (4.14) is $T(n) = O(n \lg n) + \Theta(n) = O(n \lg n)$. (Exercise 4.4-3 asks you to prove that $T(n) = \Theta(n \lg n)$.)

It's wise to verify any bound obtained with a recursion tree by using the substitution method, especially if you've made simplifying assumptions. But another strategy altogether is to use more-powerful mathematics, typically in the form of the master method in the next section (which unfortunately doesn't apply to recurrence (4.14)) or the Akra-Bazzi method (which does, but requires calculus). Even if you use a powerful method, a recursion tree can improve your intuition for what's going on beneath the heavy math.

Exercises

4.4-1
For each of the following recurrences, sketch its recursion tree, and guess a good asymptotic upper bound on its solution. Then use the substitution method to verify your answer.

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

b. $T(n) = 4T(n/3) + n$.

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

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

4.4-2
Use the substitution method to prove that recurrence (4.15) has the asymptotic lower bound $L(n) = \Omega(n)$. Conclude that $L(n) = \Theta(n)$.

4.4-3
Use the substitution method to prove that recurrence (4.14) has the solution $T(n) = \Omega(n \lg n)$. Conclude that $T(n) = \Theta(n \lg n)$.

4.4-4
Use a recursion tree to justify a good guess for the solution to the recurrence $T(n) = T(\alpha n) + T((1-\alpha)n) + \Theta(n)$, where α is a constant in the range $0 < \alpha < 1$.

4.5 The master method for solving recurrences

The master method provides a "cookbook" method for solving algorithmic recurrences of the form

$$T(n) = aT(n/b) + f(n) , \tag{4.16}$$

where $a > 0$ and $b > 1$ are constants. We call $f(n)$ a ***driving function***, and we call a recurrence of this general form a ***master recurrence***. To use the master method, you need to memorize three cases, but then you'll be able to solve many master recurrences quite easily.

A master recurrence describes the running time of a divide-and-conquer algorithm that divides a problem of size n into a subproblems, each of size $n/b<n$. The algorithm solves the a subproblems recursively, each in $T(n/b)$ time. The driving function $f(n)$ encompasses the cost of dividing the problem before the recursion, as well as the cost of combining the results of the recursive solutions to subproblems. For example, the recurrence arising from Strassen's algorithm is a master recurrence with $a = 7, b = 2$, and driving function $f(n) = \Theta(n^2)$.

As we have mentioned, in solving a recurrence that describes the running time of an algorithm, one technicality that we'd often prefer to ignore is the requirement that the input size n be an integer. For example, we saw that the running time of merge sort can be described by recurrence (2.3), $T(n) = 2T(n/2) + \Theta(n)$, on page 41. But if n is an odd number, we really don't have two problems of exactly half the size. Rather, to ensure that the problem sizes are integers, we round one subproblem down to size $\lfloor n/2 \rfloor$ and the other up to size $\lceil n/2 \rceil$, so the true recurrence is $T(n) = T(\lceil n/2 \rceil + T(\lfloor n/2 \rfloor) + \Theta(n)$. But this floors-and-ceilings recurrence is longer to write and messier to deal with than recurrence (2.3), which is defined on the reals. We'd rather not worry about floors and ceilings, if we don't have to, especially since the two recurrences have the same $\Theta(n \lg n)$ solution.

The master method allows you to state a master recurrence without floors and ceilings and implicitly infer them. No matter how the arguments are rounded up or down to the nearest integer, the asymptotic bounds that it provides remain the same. Moreover, as we'll see in Section 4.6, if you define your master recurrence on the reals, without implicit floors and ceilings, the asymptotic bounds still don't change. Thus you can ignore floors and ceilings for master recurrences. Section 4.7 gives sufficient conditions for ignoring floors and ceilings in more general divide-and-conquer recurrences.

The master theorem

The master method depends upon the following theorem.

Theorem 4.1 (Master theorem)
Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by

$$T(n) = aT(n/b) + f(n) , \tag{4.17}$$

where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a' \geq 0$ and $a'' \geq 0$ satisfying $a = a' + a''$. Then the asymptotic behavior of $T(n)$ can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ 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 few moments to understand broadly what it says. The function $n^{\log_b a}$ is called the ***watershed function***. In each of the three cases, we compare the driving function $f(n)$ to the watershed function $n^{\log_b a}$. Intuitively, if the watershed function grows asymptotically faster than the driving function, then case 1 applies. Case 2 applies if the two functions grow at nearly the same asymptotic rate. Case 3 is the "opposite" of case 1, where the driving function grows asymptotically faster than the watershed function. But the technical details matter.

In case 1, not only must the watershed function grow asymptotically faster than the driving function, it must grow *polynomially* faster. That is, the watershed function $n^{\log_b a}$ must be asymptotically larger than the driving function $f(n)$ by at least a factor of $\Theta(n^\epsilon)$ for some constant $\epsilon > 0$. The master theorem then says that the solution is $T(n) = \Theta(n^{\log_b a})$. In this case, if we look at the recursion tree for the recurrence, the cost per level grows at least geometrically from root to leaves, and the total cost of leaves dominates the total cost of the internal nodes.

In case 2, the watershed and driving functions grow at nearly the same asymptotic rate. But more specifically, the driving function grows faster than the watershed function by a factor of $\Theta(\lg^k n)$, where $k \geq 0$. The master theorem says that we tack on an extra $\lg n$ factor to $f(n)$, yielding the solution $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. In this case, each level of the recursion tree costs approximately the same—$\Theta(n^{\log_b a} \lg^k n)$—and there are $\Theta(\lg n)$ levels. In practice, the most common situation for case 2 occurs when $k = 0$, in which case the watershed and driving functions have the same asymptotic growth, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n)$.

Case 3 mirrors case 1. Not only must the driving function grow asymptotically faster than the watershed function, it must grow *polynomially* faster. That is, the driving function $f(n)$ must be asymptotically larger than the watershed function $n^{\log_b a}$ by at least a factor of $\Theta(n^\epsilon)$ for some constant $\epsilon > 0$. Moreover, the driving function must satisfy the regularity condition that $af(n/b) \leq cf(n)$. This condition is satisfied by most of the polynomially bounded functions that you're likely to encounter when applying case 3. The regularity condition might not be satisfied if the driving function grows slowly in local areas, yet relatively quickly overall. (Exercise 4.5-5 gives an example of such a function.) For case 3, the master theorem says that the solution is $T(n) = \Theta(f(n))$. If we look at the recursion tree, the cost per level drops at least geometrically from the root to the leaves, and the root cost dominates the cost of all other nodes.

It's worth looking again at the requirement that there be polynomial separation between the watershed function and the driving function for either case 1 or case 3 to apply. The separation doesn't need to be much, but it must be there, and it must grow polynomially. For example, for the recurrence $T(n) = 4T(n/2) + n^{1.99}$ (admittedly not a recurrence you're likely to see when analyzing an algorithm), the watershed function is $n^{\log_b a} = n^2$. Hence the driving function $f(n) = n^{1.99}$ is polynomially smaller by a factor of $n^{0.01}$. Thus case 1 applies with $\epsilon = 0.01$.

Using the master method

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

As a first example, consider the recurrence $T(n) = 9T(n/3) + n$. For this recurrence, we have $a = 9$ and $b = 3$, which implies that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = n = O(n^{2-\epsilon})$ for any constant $\epsilon \le 1$, we can apply case 1 of the master theorem to conclude that the solution is $T(n) = \Theta(n^2)$.

Now consider the recurrence $T(n) = T(2n/3) + 1$, which has $a = 1$ and $b = 3/2$, which means that the watershed function is $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies since $f(n) = 1 = \Theta(n^{\log_b a} \lg^0 n) = \Theta(1)$. The solution to the recurrence is $T(n) = \Theta(\lg n)$.

For the recurrence $T(n) = 3T(n/4) + n \lg n$, we have $a = 3$ and $b = 4$, which means that $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = n \lg n = \Omega(n^{\log_4 3 + \epsilon})$, where ϵ can be as large as approximately 0.2, case 3 applies as long as the regularity condition holds for $f(n)$. It does, because for sufficiently large n, we have that $af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n \lg n = cf(n)$ for $c = 3/4$. By case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

Next, let's look at the recurrence $T(n) = 2T(n/2) + n \lg n$, where we have $a = 2$, $b = 2$, and $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies since $f(n) = n \lg n = \Theta(n^{\log_b a} \lg^1 n)$. We conclude that the solution is $T(n) = \Theta(n \lg^2 n)$.

We can use the master method to solve the recurrences we saw in Sections 2.3.2, 4.1, and 4.2.

Recurrence (2.3), $T(n) = 2T(n/2) + \Theta(n)$, on page 41, characterizes the running time of merge sort. Since $a = 2$ and $b = 2$, the watershed function is $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies because $f(n) = \Theta(n)$, and the solution is $T(n) = \Theta(n \lg n)$.

Recurrence (4.9), $T(n) = 8T(n/2) + \Theta(1)$, on page 84, describes the running time of the simple recursive algorithm for matrix multiplication. We have $a = 8$ and $b = 2$, which means that the watershed function is $n^{\log_b a} = n^{\log_2 8} = n^3$. Since n^3 is polynomially larger than the driving function $f(n) = \Theta(1)$—indeed, we have $f(n) = O(n^{3-\epsilon})$ for any positive $\epsilon < 3$—case 1 applies. We conclude that $T(n) = \Theta(n^3)$.

Finally, recurrence (4.10), $T(n) = 7T(n/2) + \Theta(n^2)$, on page 87, arose from the analysis of Strassen's algorithm for matrix multiplication. For this recurrence, we have $a = 7$ and $b = 2$, and the watershed function is $n^{\log_b a} = n^{\lg 7}$. Observing that $\lg 7 = 2.807355\ldots$, we can let $\epsilon = 0.8$ and bound the driving function $f(n) = \Theta(n^2) = O(n^{\lg 7 - \epsilon})$. Case 1 applies with solution $T(n) = \Theta(n^{\lg 7})$.

When the master method doesn't apply

There are situations where you can't use the master theorem. For example, it can be that the watershed function and the driving function cannot be asymptotically compared. We might have that $f(n) \gg n^{\log_b a}$ for an infinite number of values of n but also that $f(n) \ll n^{\log_b a}$ for an infinite number of different values of n. As a practical matter, however, most of the driving functions that arise in the study of algorithms can be meaningfully compared with the watershed function. If you encounter a master recurrence for which that's not the case, you'll have to resort to substitution or other methods.

Even when the relative growths of the driving and watershed functions can be compared, the master theorem does not cover all the possibilities. There is a gap between cases 1 and 2 when $f(n) = o(n^{\log_b a})$, yet the watershed function does not grow polynomially faster than the driving function. Similarly, there is a gap between cases 2 and 3 when $f(n) = \omega(n^{\log_b a})$ and the driving function grows more than polylogarithmically faster than the watershed function, but it does not grow polynomially faster. If the driving function falls into one of these gaps, or if the regularity condition in case 3 fails to hold, you'll need to use something other than the master method to solve the recurrence.

As an example of a driving function falling into a gap, consider the recurrence $T(n) = 2T(n/2) + n/\lg n$. Since $a = 2$ and $b = 2$, the watershed function is $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. The driving function is $n/\lg n = o(n)$, which means that it grows asymptotically more slowly than the watershed function n. But $n/\lg n$ grows only *logarithmically* slower than n, not *polynomially* slower. More precisely, equation (3.24) on page 67 says that $\lg n = o(n^\epsilon)$ for any constant $\epsilon > 0$, which means that $1/\lg n = \omega(n^{-\epsilon})$ and $n/\lg n = \omega(n^{1-\epsilon}) = \omega(n^{\log_b a - \epsilon})$. Thus no constant $\epsilon > 0$ exists such that $n/\lg n = O(n^{\log_b a - \epsilon})$, which is required for case 1 to apply. Case 2 fails to apply as well, since $n/\lg n = \Theta(n^{\log_b a} \lg^k n)$, where $k = -1$, but k must be nonnegative for case 2 to apply.

To solve this kind of recurrence, you must use another method, such as the substitution method (Section 4.3) or the Akra-Bazzi method (Section 4.7). (Exercise 4.6-3 asks you to show that the answer is $\Theta(n \lg \lg n)$.) Although the master theorem doesn't handle this particular recurrence, it does handle the overwhelming majority of recurrences that tend to arise in practice.

Exercises

4.5-1

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

a. $T(n) = 2T(n/4) + 1$.

b. $T(n) = 2T(n/4) + \sqrt{n}$.

c. $T(n) = 2T(n/4) + \sqrt{n} \lg^2 n$.

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

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

4.5-2

Professor Caesar wants to develop a matrix-multiplication algorithm that is asymptotically faster than Strassen's algorithm. His algorithm will use the divide-and-conquer method, dividing each matrix into $n/4 \times n/4$ submatrices, and the divide and combine steps together will take $\Theta(n^2)$ time. Suppose that the professor's algorithm creates a recursive subproblems of size $n/4$. What is the largest integer value of a for which his algorithm could possibly run asymptotically faster than Strassen's?

4.5-3

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 2.3-6 for a description of binary search.)

4.5-4

Consider the function $f(n) = \lg n$. Argue that although $f(n/2) < f(n)$, the regularity condition $af(n/b) \le cf(n)$ with $a = 1$ and $b = 2$ does not hold for any constant $c < 1$. Argue further that for any $\epsilon > 0$, the condition in case 3 that $f(n) = \Omega(n^{\log_b a + \epsilon})$ does not hold.