4 Divide-and-Conquer

The divide-and-conquer method is a powerful strategy for designing asymptotically efficient algorithms. We saw an example of divide-and-conquer in Section 2.3.1 when learning about merge sort. In this chapter, we'll explore applications of the divide-and-conquer method and acquire valuable mathematical tools that you can use to solve the recurrences that arise when analyzing divide-and-conquer algorithms.

Recall that for divide-and-conquer, you solve a given problem (instance) recursively. If the problem is small enough—the ***base case***—you just solve it directly without recursing. Otherwise—the ***recursive case***—you perform three characteristic steps:

Divide the problem into one or more subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

Combine the subproblem solutions to form a solution to the original problem.

A divide-and-conquer algorithm breaks down a large problem into smaller subproblems, which themselves may be broken down into even smaller subproblems, and so forth. The recursion ***bottoms out*** when it reaches a base case and the subproblem is small enough to solve directly without further recursing.

Recurrences

To analyze recursive divide-and-conquer algorithms, we'll need some mathematical tools. A ***recurrence*** is an equation that describes a function in terms of its value on other, typically smaller, arguments. Recurrences go hand in hand with the divide-and-conquer method because they give us a natural way to characterize the running times of recursive algorithms mathematically. You saw an example of a recurrence in Section 2.3.2 when we analyzed the worst-case running time of merge sort.

For the divide-and-conquer matrix-multiplication algorithms presented in Sections 4.1 and 4.2, we'll derive recurrences that describe their worst-case running times. To understand why these two divide-and-conquer algorithms perform the way they do, you'll need to learn how to solve the recurrences that describe their running times. Sections 4.3–4.7 teach several methods for solving recurrences. These sections also explore the mathematics behind recurrences, which can give you stronger intuition for designing your own divide-and-conquer algorithms.

We want to get to the algorithms as soon as possible. So, let's just cover a few recurrence basics now, and then we'll look more deeply at recurrences, especially how to solve them, after we see the matrix-multiplication examples.

The general form of a recurrence is an equation or inequality that describes a function over the integers or reals using the function itself. It contains two or more cases, depending on the argument. If a case involves the recursive invocation of the function on different (usually smaller) inputs, it is a ***recursive case***. If a case does not involve a recursive invocation, it is a ***base case***. There may be zero, one, or many functions that satisfy the statement of the recurrence. The recurrence is ***well defined*** if there is at least one function that satisfies it, and ***ill defined*** otherwise.

Algorithmic recurrences

We'll be particularly interested in recurrences that describe the running times of divide-and-conquer algorithms. A recurrence $T(n)$ is ***algorithmic*** if, for every sufficiently large ***threshold*** constant $n_0 > 0$, the following two properties hold:

1. For all $n < n_0$, we have $T(n) = \Theta(1)$.
2. For all $n \geq n_0$, every path of recursion terminates in a defined base case within a finite number of recursive invocations.

Similar to how we sometimes abuse asymptotic notation (see page 60), when a function is not defined for all arguments, we understand that this definition is constrained to values of n for which $T(n)$ is defined.

Why would a recurrence $T(n)$ that represents a (correct) divide-and-conquer algorithm's worst-case running time satisfy these properties for all sufficiently large threshold constants? The first property says that there exist constants c_1, c_2 such that $0 < c_1 \leq T(n) \leq c_2$ for $n < n_0$. For every legal input, the algorithm must output the solution to the problem it's solving in finite time (see Section 1.1). Thus we can let c_1 be the minimum amount of time to call and return from a procedure, which must be positive, because machine instructions need to be executed to invoke a procedure. The running time of the algorithm may not be defined for some values of n if there are no legal inputs of that size, but it must be defined for at least one, or else the "algorithm" doesn't solve any problem. Thus we can let c_2 be the algorithm's maximum running time on any input of size $n < n_0$, where n_0 is

sufficiently large that the algorithm solves at least one problem of size less than n_0. The maximum is well defined, since there are at most a finite number of inputs of size less than n_0, and there is at least one if n_0 is sufficiently large. Consequently, $T(n)$ satisfies the first property. If the second property fails to hold for $T(n)$, then the algorithm isn't correct, because it would end up in an infinite recursive loop or otherwise fail to compute a solution. Thus, it stands to reason that a recurrence for the worst-case running time of a correct divide-and-conquer algorithm would be algorithmic.

Conventions for recurrences

We adopt the following convention:

> *Whenever a recurrence is stated without an explicit base case, we assume that the recurrence is algorithmic.*

That means you're free to pick any sufficiently large threshold constant n_0 for the range of base cases where $T(n) = \Theta(1)$. Interestingly, the asymptotic solutions of most algorithmic recurrences you're likely to see when analyzing algorithms don't depend on the choice of threshold constant, as long as it's large enough to make the recurrence well defined.

Asymptotic solutions of algorithmic divide-and-conquer recurrences also don't tend to change when we drop any floors or ceilings in a recurrence defined on the integers to convert it to a recurrence defined on the reals. Section 4.7 gives a sufficient condition for ignoring floors and ceilings that applies to most of the divide-and-conquer recurrences you're likely to see. Consequently, we'll frequently state algorithmic recurrences without floors and ceilings. Doing so generally simplifies the statement of the recurrences, as well as any math that we do with them.

You may sometimes see recurrences that are not equations, but rather inequalities, such as $T(n) \le 2T(n/2) + \Theta(n)$. Because such a recurrence states only an upper bound on $T(n)$, we express its solution using O-notation rather than Θ-notation. Similarly, if the inequality is reversed to $T(n) \ge 2T(n/2) + \Theta(n)$, then, because the recurrence gives only a lower bound on $T(n)$, we use Ω-notation in its solution.

Divide-and-conquer and recurrences

This chapter illustrates the divide-and-conquer method by presenting and using recurrences to analyze two divide-and-conquer algorithms for multiplying $n \times n$ matrices. Section 4.1 presents a simple divide-and-conquer algorithm that solves a matrix-multiplication problem of size n by breaking it into four subproblems of size $n/2$, which it then solves recursively. The running time of the algorithm can be characterized by the recurrence

$$T(n) = 8T(n/2) + \Theta(1) ,$$

which turns out to have the solution $T(n) = \Theta(n^3)$. Although this divide-and-conquer algorithm is no faster than the straightforward method that uses a triply nested loop, it leads to an asymptotically faster divide-and-conquer algorithm due to V. Strassen, which we'll explore in Section 4.2. Strassen's remarkable algorithm divides a problem of size n into seven subproblems of size $n/2$ which it solves recursively. The running time of Strassen's algorithm can be described by the recurrence

$$T(n) = 7T(n/2) + \Theta(n^2) ,$$

which has the solution $T(n) = \Theta(n^{\lg 7}) = O(n^{2.81})$. Strassen's algorithm beats the straightforward looping method asymptotically.

These two divide-and-conquer algorithms both break a problem of size n into several subproblems of size $n/2$. Although it is common when using divide-and-conquer for all the subproblems to have the same size, that isn't always the case. Sometimes it's productive to divide a problem of size n into subproblems of different sizes, and then the recurrence describing the running time reflects the irregularity. For example, consider a divide-and-conquer algorithm that divides a problem of size n into one subproblem of size $n/3$ and another of size $2n/3$, taking $\Theta(n)$ time to divide the problem and combine the solutions to the subproblems. Then the algorithm's running time can be described by the recurrence

$$T(n) = T(n/3) + T(2n/3) + \Theta(n) ,$$

which turns out to have solution $T(n) = \Theta(n \lg n)$. We'll even see an algorithm in Chapter 9 that solves a problem of size n by recursively solving a subproblem of size $n/5$ and another of size $7n/10$, taking $\Theta(n)$ time for the divide and combine steps. Its performance satisfies the recurrence

$$T(n) = T(n/5) + T(7n/10) + \Theta(n) ,$$

which has solution $T(n) = \Theta(n)$.

Although divide-and-conquer algorithms usually create subproblems with sizes a constant fraction of the original problem size, that's not always the case. For example, a recursive version of linear search (see Exercise 2.1-4) creates just one subproblem, with one element less than the original problem. Each recursive call takes constant time plus the time to recursively solve a subproblem with one less element, leading to the recurrence

$$T(n) = T(n-1) + \Theta(1) ,$$

which has solution $T(n) = \Theta(n)$. Nevertheless, the vast majority of efficient divide-and-conquer algorithms solve subproblems that are a constant fraction of the size of the original problem, which is where we'll focus our efforts.

Solving recurrences

After learning about divide-and-conquer algorithms for matrix multiplication in Sections 4.1 and 4.2, we'll explore several mathematical tools for solving recurrences—that is, for obtaining asymptotic Θ-, O-, or Ω-bounds on their solutions. We want simple-to-use tools that can handle the most commonly occurring situations. But we also want general tools that work, perhaps with a little more effort, for less common cases. This chapter offers four methods for solving recurrences:

- In the ***substitution method*** (Section 4.3), you guess the form of a bound and then use mathematical induction to prove your guess correct and solve for constants. This method is perhaps the most robust method for solving recurrences, but it also requires you to make a good guess and to produce an inductive proof.
- The ***recursion-tree method*** (Section 4.4) models the recurrence as a tree whose nodes represent the costs incurred at various levels of the recursion. To solve the recurrence, you determine the costs at each level and add them up, perhaps using techniques for bounding summations from Section A.2. Even if you don't use this method to formally prove a bound, it can be helpful in guessing the form of the bound for use in the substitution method.
- The ***master method*** (Sections 4.5 and 4.6) is the easiest method, when it applies. It provides bounds for recurrences of the form

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

  where $a > 0$ and $b > 1$ are constants and $f(n)$ is a given "driving" function. This type of recurrence tends to arise more frequently in the study of algorithms than any other. It characterizes a divide-and-conquer algorithm that creates a subproblems, each of which is $1/b$ times the size of the original problem, using $f(n)$ time for the divide and combine steps. To apply the master method, you need to memorize three cases, but once you do, you can easily determine asymptotic bounds on running times for many divide-and-conquer algorithms.
- The ***Akra-Bazzi method*** (Section 4.7) is a general method for solving divide-and-conquer recurrences. Although it involves calculus, it can be used to attack more complicated recurrences than those addressed by the master method.

4.1 Multiplying square matrices

We can use the divide-and-conquer method to multiply square matrices. If you've seen matrices before, then you probably know how to multiply them. (Otherwise,

you should read Section D.1.) Let $A = (a_{ik})$ and $B = (b_{jk})$ be square $n \times n$ matrices. The matrix product $C = A \cdot B$ is also an $n \times n$ matrix, where for $i, j = 1, 2, \ldots, n$, the (i, j) entry of C is given by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \ . \tag{4.1}$$

Generally, we'll assume that the matrices are ***dense***, meaning that most of the n^2 entries are not 0, as opposed to ***sparse***, where most of the n^2 entries are 0 and the nonzero entries can be stored more compactly than in an $n \times n$ array.

Computing the matrix C requires computing n^2 matrix entries, each of which is the sum of n pairwise products of input elements from A and B. The MATRIX-MULTIPLY procedure implements this strategy in a straightforward manner, and it generalizes the problem slightly. It takes as input three $n \times n$ matrices A, B, and C, and it adds the matrix product $A \cdot B$ to C, storing the result in C. Thus, it computes $C = C + A \cdot B$, instead of just $C = A \cdot B$. If only the product $A \cdot B$ is needed, just initialize all n^2 entries of C to 0 before calling the procedure, which takes an additional $\Theta(n^2)$ time. We'll see that the cost of matrix multiplication asymptotically dominates this initialization cost.

```
MATRIX-MULTIPLY(A, B, C, n)
1  for i = 1 to n                    // compute entries in each of n rows
2      for j = 1 to n                // compute n entries in row i
3          for k = 1 to n
4              c_ij = c_ij + a_ik · b_kj   // add in another term of equation (4.1)
```

The pseudocode for MATRIX-MULTIPLY works as follows. The **for** loop of lines 1–4 computes the entries of each row i, and within a given row i, the **for** loop of lines 2–4 computes each of the entries c_{ij} for each column j. Each iteration of the **for** loop of lines 3–4 adds in one more term of equation (4.1).

Because each of the triply nested **for** loops runs for exactly n iterations, and each execution of line 4 takes constant time, the MATRIX-MULTIPLY procedure operates in $\Theta(n^3)$ time. Even if we add in the $\Theta(n^2)$ time for initializing C to 0, the running time is still $\Theta(n^3)$.

A simple divide-and-conquer algorithm

Let's see how to compute the matrix product $A \cdot B$ using divide-and-conquer. For $n > 1$, the divide step partitions the $n \times n$ matrices into four $n/2 \times n/2$ submatrices. We'll assume that n is an exact power of 2, so that as the algorithm recurses, we are guaranteed that the submatrix dimensions are integer. (Exercise 4.1-1 asks you

to relax this assumption.) As with MATRIX-MULTIPLY, we'll actually compute $C = C + A \cdot B$. But to simplify the math behind the algorithm, let's assume that C has been initialized to the zero matrix, so that we are indeed computing $C = A \cdot B$.

The divide step views each of the $n \times n$ matrices A, B, and C as four $n/2 \times n/2$ submatrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}. \tag{4.2}$$

Then we can write the matrix product as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{4.3}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix}, \tag{4.4}$$

which corresponds to the equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}\,, \tag{4.5}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}\,, \tag{4.6}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}\,, \tag{4.7}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}\,. \tag{4.8}$$

Equations (4.5)–(4.8) involve eight $n/2 \times n/2$ multiplications and four additions of $n/2 \times n/2$ submatrices.

As we look to transform these equations to an algorithm that can be described with pseudocode, or even implemented for real, there are two common approaches for implementing the matrix partitioning.

One strategy is to allocate temporary storage to hold A's four submatrices A_{11}, A_{12}, A_{21}, and A_{22} and B's four submatrices B_{11}, B_{12}, B_{21}, and B_{22}. Then copy each element in A and B to its corresponding location in the appropriate submatrix. After the recursive conquer step, copy the elements in each of C's four submatrices C_{11}, C_{12}, C_{21}, and C_{22} to their corresponding locations in C. This approach takes $\Theta(n^2)$ time, since $3n^2$ elements are copied.

The second approach uses index calculations and is faster and more practical. A submatrix can be specified within a matrix by indicating where within the matrix the submatrix lies without touching any matrix elements. Partitioning a matrix (or recursively, a submatrix) only involves arithmetic on this location information, which has constant size independent of the size of the matrix. Changes to the submatrix elements update the original matrix, since they occupy the same storage.

Going forward, we'll assume that index calculations are used and that partitioning can be performed in $\Theta(1)$ time. Exercise 4.1-3 asks you to show that it makes no difference to the overall asymptotic running time of matrix multiplication, however, whether the partitioning of matrices uses the first method of copying or the

second method of index calculation. But for other divide-and-conquer matrix calculations, such as matrix addition, it can make a difference, as Exercise 4.1-4 asks you to show.

The procedure MATRIX-MULTIPLY-RECURSIVE uses equations (4.5)–(4.8) to implement a divide-and-conquer strategy for square-matrix multiplication. Like MATRIX-MULTIPLY, the procedure MATRIX-MULTIPLY-RECURSIVE computes $C = C + A \cdot B$ since, if necessary, C can be initialized to 0 before the procedure is called in order to compute only $C = A \cdot B$.

```
MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
 1  if n == 1
 2   // Base case.
 3       c_11 = c_11 + a_11 · b_11
 4       return
 5   // Divide.
 6   partition A, B, and C into n/2 × n/2 submatrices
         A_11, A_12, A_21, A_22; B_11, B_12, B_21, B_22;
         and C_11, C_12, C_21, C_22; respectively
 7   // Conquer.
 8   MATRIX-MULTIPLY-RECURSIVE(A_11, B_11, C_11, n/2)
 9   MATRIX-MULTIPLY-RECURSIVE(A_11, B_12, C_12, n/2)
10   MATRIX-MULTIPLY-RECURSIVE(A_21, B_11, C_21, n/2)
11   MATRIX-MULTIPLY-RECURSIVE(A_21, B_12, C_22, n/2)
12   MATRIX-MULTIPLY-RECURSIVE(A_12, B_21, C_11, n/2)
13   MATRIX-MULTIPLY-RECURSIVE(A_12, B_22, C_12, n/2)
14   MATRIX-MULTIPLY-RECURSIVE(A_22, B_21, C_21, n/2)
15   MATRIX-MULTIPLY-RECURSIVE(A_22, B_22, C_22, n/2)
```

As we walk through the pseudocode, we'll derive a recurrence to characterize its running time. Let $T(n)$ be the worst-case time to multiply two $n \times n$ matrices using this procedure.

In the base case, when $n = 1$, line 3 performs just the one scalar multiplication and one addition, which means that $T(1) = \Theta(1)$. As is our convention for constant base cases, we can omit this base case in the statement of the recurrence.

The recursive case occurs when $n > 1$. As discussed, we'll use index calculations to partition the matrices in line 6, taking $\Theta(1)$ time. Lines 8–15 recursively call MATRIX-MULTIPLY-RECURSIVE a total of eight times. The first four recursive calls compute the first terms of equations (4.5)–(4.8), and the subsequent four recursive calls compute and add in the second terms. Each recursive call adds the product of a submatrix of A and a submatrix of B to the appropriate submatrix

of C in place, thanks to index calculations. Because each recursive call multiplies two $n/2 \times n/2$ matrices, thereby contributing $T(n/2)$ to the overall running time, the time taken by all eight recursive calls is $8T(n/2)$. There is no combine step, because the matrix C is updated in place. The total time for the recursive case, therefore, is the sum of the partitioning time and the time for all the recursive calls, or $\Theta(1) + 8T(n/2)$.

Thus, omitting the statement of the base case, our recurrence for the running time of MATRIX-MULTIPLY-RECURSIVE is

$$T(n) = 8T(n/2) + \Theta(1) \; . \tag{4.9}$$

As we'll see from the master method in Section 4.5, recurrence (4.9) has the solution $T(n) = \Theta(n^3)$, which means that it has the same asymptotic running time as the straightforward MATRIX-MULTIPLY procedure.

Why is the $\Theta(n^3)$ solution to this recurrence so much larger than the $\Theta(n \lg n)$ solution to the merge-sort recurrence (2.3) on page 41? After all, the recurrence for merge sort contains a $\Theta(n)$ term, whereas the recurrence for recursive matrix multiplication contains only a $\Theta(1)$ term.

Let's think about what the recursion tree for recurrence (4.9) would look like as compared with the recursion tree for merge sort, illustrated in Figure 2.5 on page 43. The factor of 2 in the merge-sort recurrence determines how many children each tree node has, which in turn determines how many terms contribute to the sum at each level of the tree. In comparison, for the recurrence (4.9) for MATRIX-MULTIPLY-RECURSIVE, each internal node in the recursion tree has eight children, not two, leading to a "bushier" recursion tree with many more leaves, despite the fact that the internal nodes are each much smaller. Consequently, the solution to recurrence (4.9) grows much more quickly than the solution to recurrence (2.3), which is borne out in the actual solutions: $\Theta(n^3)$ versus $\Theta(n \lg n)$.

Exercises

Note: You may wish to read Section 4.5 before attempting some of these exercises.

4.1-1

Generalize MATRIX-MULTIPLY-RECURSIVE to multiply $n \times n$ matrices for which n is not necessarily an exact power of 2. Give a recurrence describing its running time. Argue that it runs in $\Theta(n^3)$ time in the worst case.

4.1-2

How quickly can you multiply a $kn \times n$ matrix (kn rows and n columns) by an $n \times kn$ matrix, where $k \geq 1$, using MATRIX-MULTIPLY-RECURSIVE as a subroutine? Answer the same question for multiplying an $n \times kn$ matrix by a $kn \times n$ matrix. Which is asymptotically faster, and by how much?

4.1-3

Suppose that instead of partitioning matrices by index calculation in MATRIX-MULTIPLY-RECURSIVE, you copy the appropriate elements of A, B, and C into separate $n/2 \times n/2$ submatrices $A_{11}, A_{12}, A_{21}, A_{22}$; $B_{11}, B_{12}, B_{21}, B_{22}$; and $C_{11}, C_{12}, C_{21}, C_{22}$, respectively. After the recursive calls, you copy the results from C_{11}, C_{12}, C_{21}, and C_{22} back into the appropriate places in C. How does recurrence (4.9) change, and what is its solution?

4.1-4

Write pseudocode for a divide-and-conquer algorithm MATRIX-ADD-RECURSIVE that sums two $n \times n$ matrices A and B by partitioning each of them into four $n/2 \times n/2$ submatrices and then recursively summing corresponding pairs of submatrices. Assume that matrix partitioning uses $\Theta(1)$-time index calculations. Write a recurrence for the worst-case running time of MATRIX-ADD-RECURSIVE, and solve your recurrence. What happens if you use $\Theta(n^2)$-time copying to implement the partitioning instead of index calculations?

4.2 Strassen's algorithm for matrix multiplication

You might find it hard to imagine that any matrix multiplication algorithm could take less than $\Theta(n^3)$ time, since the natural definition of matrix multiplication requires n^3 scalar multiplications. Indeed, many mathematicians presumed that it was not possible to multiply matrices in $o(n^3)$ time until 1969, when V. Strassen [424] published a remarkable recursive algorithm for multiplying $n \times n$ matrices. Strassen's algorithm runs in $\Theta(n^{\lg 7})$ time. Since $\lg 7 = 2.8073549\ldots$, Strassen's algorithm runs in $O(n^{2.81})$ time, which is asymptotically better than the $\Theta(n^3)$ MATRIX-MULTIPLY and MATRIX-MULTIPLY-RECURSIVE procedures.

The key to Strassen's method is to use the divide-and-conquer idea from the MATRIX-MULTIPLY-RECURSIVE procedure, but make the recursion tree less bushy. We'll actually increase the work for each divide and combine step by a constant factor, but the reduction in bushiness will pay off. We won't reduce the bushiness from the eight-way branching of recurrence (4.9) all the way down to the two-way branching of recurrence (2.3), but we'll improve it just a little, and that will make a big difference. Instead of performing eight recursive multiplications of $n/2 \times n/2$ matrices, Strassen's algorithm performs only seven. The cost of eliminating one matrix multiplication is several new additions and subtractions of $n/2 \times n/2$ matrices, but still only a constant number. Rather than saying "additions and subtractions" everywhere, we'll adopt the common terminology of call-

ing them both "additions" because subtraction is structurally the same computation as addition, except for a change of sign.

To get an inkling how the number of multiplications might be reduced, as well as why reducing the number of multiplications might be desirable for matrix calculations, suppose that you have two numbers x and y, and you want to calculate the quantity $x^2 - y^2$. The straightforward calculation requires two multiplications to square x and y, followed by one subtraction (which you can think of as a "negative addition"). But let's recall the old algebra trick $x^2 - y^2 = x^2 - xy + xy - y^2 = x(x - y) + y(x - y) = (x + y)(x - y)$. Using this formulation of the desired quantity, you could instead compute the sum $x + y$ and the difference $x - y$ and then multiply them, requiring only a single multiplication and two additions. At the cost of an extra addition, only one multiplication is needed to compute an expression that looks as if it requires two. If x and y are scalars, there's not much difference: both approaches require three scalar operations. If x and y are large matrices, however, the cost of multiplying outweighs the cost of adding, in which case the second method outperforms the first, although not asymptotically.

Strassen's strategy for reducing the number of matrix multiplications at the expense of more matrix additions is not at all obvious—perhaps the biggest understatement in this book! As with MATRIX-MULTIPLY-RECURSIVE, Strassen's algorithm uses the divide-and-conquer method to compute $C = C + A \cdot B$, where A, B, and C are all $n \times n$ matrices and n is an exact power of 2. Strassen's algorithm computes the four submatrices C_{11}, C_{12}, C_{21}, and C_{22} of C from equations (4.5)–(4.8) on page 82 in four steps. We'll analyze costs as we go along to develop a recurrence $T(n)$ for the overall running time. Let's see how it works:

1. If $n = 1$, the matrices each contain a single element. Perform a single scalar multiplication and a single scalar addition, as in line 3 of MATRIX-MULTIPLY-RECURSIVE, taking $\Theta(1)$ time, and return. Otherwise, partition the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.2). This step takes $\Theta(1)$ time by index calculation, just as in MATRIX-MULTIPLY-RECURSIVE.

2. Create $n/2 \times n/2$ matrices $S_1, S_2, \ldots, S_{10}$, each of which is the sum or difference of two submatrices from step 1. Create and zero the entries of seven $n/2 \times n/2$ matrices $P_1, P_2, \ldots, P_7$ to hold seven $n/2 \times n/2$ matrix products. All 17 matrices can be created, and the P_i initialized, in $\Theta(n^2)$ time.

3. Using the submatrices from step 1 and the matrices $S_1, S_2, \ldots, S_{10}$ created in step 2, recursively compute each of the seven matrix products $P_1, P_2, \ldots, P_7$, taking $7T(n/2)$ time.

4. Update the four submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding or subtracting various P_i matrices, which takes $\Theta(n^2)$ time.

We'll see the details of steps 2–4 in a moment, but we already have enough information to set up a recurrence for the running time of Strassen's method. As is common, the base case in step 1 takes $\Theta(1)$ time, which we'll omit when stating the recurrence. When $n > 1$, steps 1, 2, and 4 take a total of $\Theta(n^2)$ time, and step 3 requires seven multiplications of $n/2 \times n/2$ matrices. Hence, we obtain the following recurrence for the running time of Strassen's algorithm:

$$T(n) = 7T(n/2) + \Theta(n^2) . \tag{4.10}$$

Compared with MATRIX-MULTIPLY-RECURSIVE, we have traded off one recursive submatrix multiplication for a constant number of submatrix additions. Once you understand recurrences and their solutions, you'll be able to see why this tradeoff actually leads to a lower asymptotic running time. By the master method in Section 4.5, recurrence (4.10) has the solution $T(n) = \Theta(n^{\lg 7}) = O(n^{2.81})$, beating the $\Theta(n^3)$-time algorithms.

Now, let's delve into the details. Step 2 creates the following 10 matrices:

$$\begin{aligned}
S_1 &= B_{12} - B_{22} , \\
S_2 &= A_{11} + A_{12} , \\
S_3 &= A_{21} + A_{22} , \\
S_4 &= B_{21} - B_{11} , \\
S_5 &= A_{11} + A_{22} , \\
S_6 &= B_{11} + B_{22} , \\
S_7 &= A_{12} - A_{22} , \\
S_8 &= B_{21} + B_{22} , \\
S_9 &= A_{11} - A_{21} , \\
S_{10} &= B_{11} + B_{12} .
\end{aligned}$$

This step adds or subtracts $n/2 \times n/2$ matrices 10 times, taking $\Theta(n^2)$ time.

Step 3 recursively multiplies $n/2 \times n/2$ matrices 7 times to compute the following $n/2 \times n/2$ matrices, each of which is the sum or difference of products of A and B submatrices:

$$\begin{aligned}
P_1 &= A_{11} \cdot S_1 \quad (= A_{11} \cdot B_{12} - A_{11} \cdot B_{22}) , \\
P_2 &= S_2 \cdot B_{22} \quad (= A_{11} \cdot B_{22} + A_{12} \cdot B_{22}) , \\
P_3 &= S_3 \cdot B_{11} \quad (= A_{21} \cdot B_{11} + A_{22} \cdot B_{11}) , \\
P_4 &= A_{22} \cdot S_4 \quad (= A_{22} \cdot B_{21} - A_{22} \cdot B_{11}) , \\
P_5 &= S_5 \cdot S_6 \quad (= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}) , \\
P_6 &= S_7 \cdot S_8 \quad (= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}) , \\
P_7 &= S_9 \cdot S_{10} \quad (= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}) .
\end{aligned}$$

The only multiplications that the algorithm performs are those in the middle column of these equations. The right-hand column just shows what these products equal in terms of the original submatrices created in step 1, but the terms are never explicitly calculated by the algorithm.

Step 4 adds to and subtracts from the four $n/2 \times n/2$ submatrices of the product C the various P_i matrices created in step 3. We start with

$$C_{11} = C_{11} + P_5 + P_4 - P_2 + P_6 .$$

Expanding the calculation on the right-hand side, with the expansion of each P_i on its own line and vertically aligning terms that cancel out, we see that the update to C_{11} equals

$$\begin{array}{rrrr}
A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} & & & \\
- A_{22} \cdot B_{11} & + A_{22} \cdot B_{21} & & \\
- A_{11} \cdot B_{22} & & - A_{12} \cdot B_{22} & \\
- A_{22} \cdot B_{22} - A_{22} \cdot B_{21} + A_{12} \cdot B_{22} + A_{12} \cdot B_{21} & & & \\
\hline
A_{11} \cdot B_{11} & & & + A_{12} \cdot B_{21} ,
\end{array}$$

which corresponds to equation (4.5). Similarly, setting

$$C_{12} = C_{12} + P_1 + P_2$$

means that the update to C_{12} equals

$$\begin{array}{rrr}
A_{11} \cdot B_{12} - A_{11} \cdot B_{22} & & \\
+ A_{11} \cdot B_{22} + A_{12} \cdot B_{22} & & \\
\hline
A_{11} \cdot B_{12} & & + A_{12} \cdot B_{22} ,
\end{array}$$

corresponding to equation (4.6). Setting

$$C_{21} = C_{21} + P_3 + P_4$$

means that the update to C_{21} equals

$$\begin{array}{rrr}
A_{21} \cdot B_{11} + A_{22} \cdot B_{11} & & \\
- A_{22} \cdot B_{11} + A_{22} \cdot B_{21} & & \\
\hline
A_{21} \cdot B_{11} & & + A_{22} \cdot B_{21} ,
\end{array}$$

corresponding to equation (4.7). Finally, setting

$$C_{22} = C_{22} + P_5 + P_1 - P_3 - P_7$$

means that the update to C_{22} equals

$$\begin{array}{llll} A_{11}\cdot B_{11} + A_{11}\cdot B_{22} + A_{22}\cdot B_{11} + A_{22}\cdot B_{22} & & & \\ \quad - A_{11}\cdot B_{22} & & + A_{11}\cdot B_{12} & \\ & - A_{22}\cdot B_{11} & & - A_{21}\cdot B_{11} \\ -A_{11}\cdot B_{11} & & - A_{11}\cdot B_{12} + A_{21}\cdot B_{11} + A_{21}\cdot B_{12} & \\ \hline & A_{22}\cdot B_{22} & & + A_{21}\cdot B_{12}\ , \end{array}$$

which corresponds to equation (4.8). Altogether, since we add or subtract $n/2 \times n/2$ matrices 12 times in step 4, this step indeed takes $\Theta(n^2)$ time.

We can see that Strassen's remarkable algorithm, comprising steps 1–4, produces the correct matrix product using 7 submatrix multiplications and 18 submatrix additions. We can also see that recurrence (4.10) characterizes its running time. Since Section 4.5 shows that this recurrence has the solution $T(n) = \Theta(n^{\lg 7}) = o(n^3)$, Strassen's method asymptotically beats the $\Theta(n^3)$ MATRIX-MULTIPLY and MATRIX-MULTIPLY-RECURSIVE procedures.

Exercises

Note: You may wish to read Section 4.5 before attempting some of these exercises.

4.2-1

Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}\begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}.$$

Show your work.

4.2-2

Write pseudocode for Strassen's algorithm.

4.2-3

What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in $o(n^{\lg 7})$ time? What is the running time of this algorithm?

4.2-4

V. Pan discovered a way of multiplying 68×68 matrices using 132,464 multiplications, a way of multiplying 70×70 matrices using 143,640 multiplications, and a way of multiplying 72×72 matrices using 155,424 multiplications. Which method yields the best asymptotic running time when used in a divide-and-conquer matrix-multiplication algorithm? How does it compare with Strassen's algorithm?

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 $T(n) \leq cn^2$ 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 $T(n) \leq c2^n$, 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

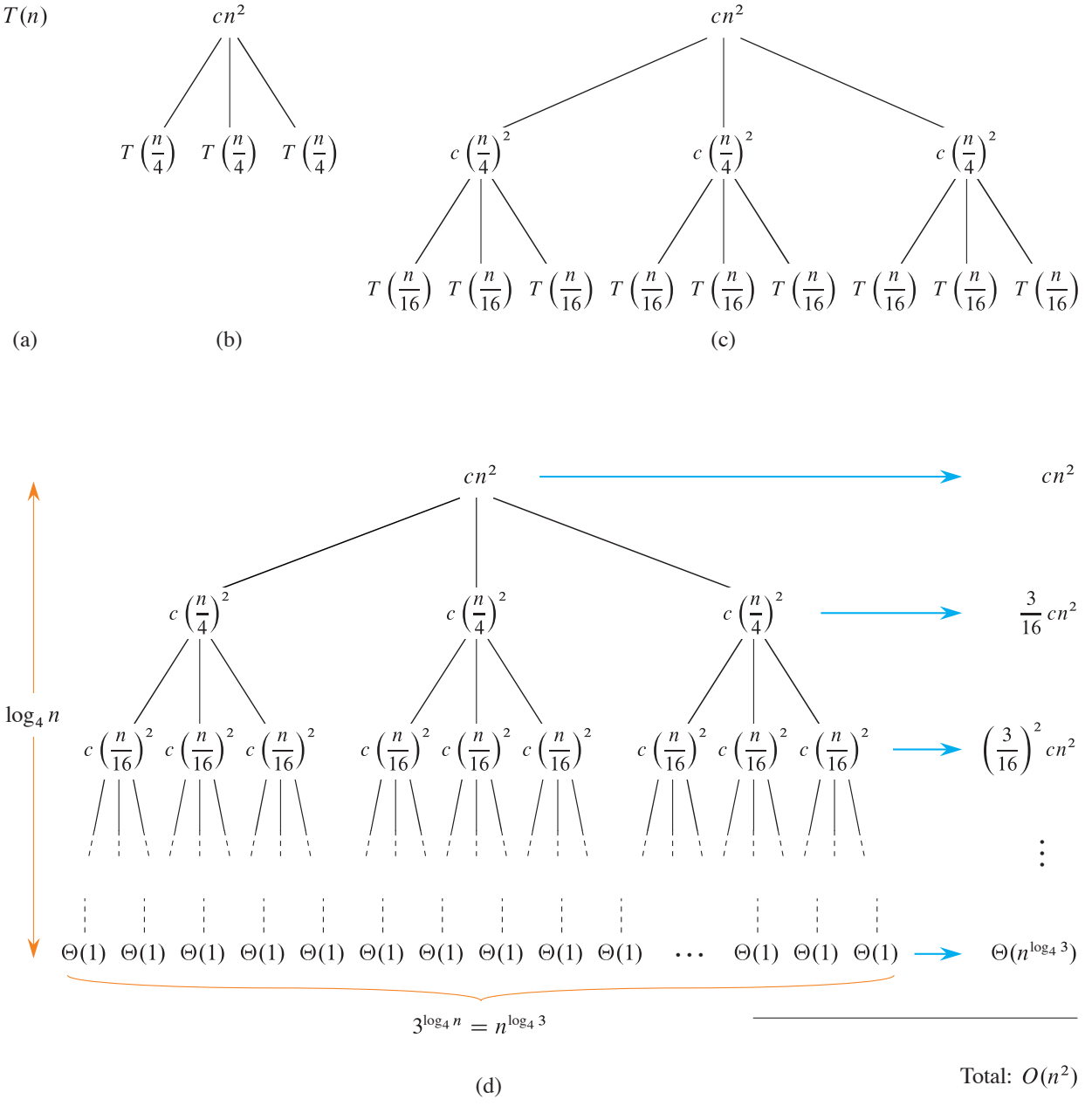

Figure 4.1 Constructing a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in **(d)** has height $\log_4 n$.

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 $T(n) \leq dn^2$ for some constant $d > 0$. Using the same constant $c > 0$ as before, we have

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

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

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 $dn^2 \geq d \geq T(n)$ for $1 \leq n < n_0$, 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 $c' \geq c$ 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 $n \geq n_0$. 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.

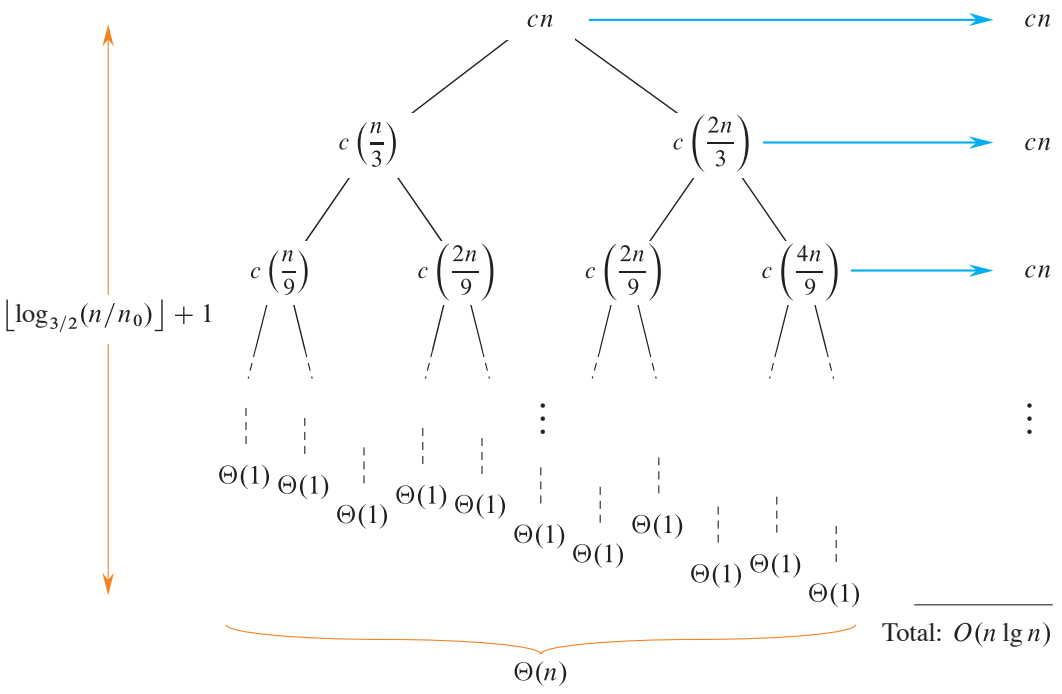

Figure 4.2 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

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

$2^h = 2^{\lfloor \log_{3/2} n \rfloor + 1} \leq 2n^{\log_{3/2} 2}$ 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

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

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 $L(n) \leq dn$ for some constant $d > 0$, and assuming that the inductive hypothesis holds for all values less than n, we have

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

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 $\epsilon \leq 1$, 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 $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n)$ 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 $af(n/b) \leq cf(n)$ 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.

4.5-5

Show that for suitable constants a, b, and ϵ, the function $f(n) = 2^{\lceil \lg n \rceil}$ satisfies all the conditions in case 3 of the master theorem except the regularity condition.

⋆ 4.6 Proof of the continuous master theorem

Proving the master theorem (Theorem 4.1) in its full generality, especially dealing with the knotty technical issue of floors and ceilings, is beyond the scope of this book. This section, however, states and proves a variant of the master theorem, called the ***continuous master theorem***[1] in which the master recurrence (4.17) is defined over sufficiently large positive real numbers. The proof of this version, uncomplicated by floors and ceilings, contains the main ideas needed to understand how master recurrences behave. Section 4.7 discusses floors and ceilings in divide-and-conquer recurrences at greater length, presenting sufficient conditions for them not to affect the asymptotic solutions.

Of course, since you need not understand the proof of the master theorem in order to apply the master method, you may choose to skip this section. But if you wish to study more-advanced algorithms beyond the scope of this textbook, you may appreciate a better understanding of the underlying mathematics, which the proof of the continuous master theorem provides.

Although we usually assume that recurrences are algorithmic and don't require an explicit statement of a base case, we must be much more careful for proofs that justify the practice. The lemmas and theorem in this section explicitly state the base cases, because the inductive proofs require mathematical grounding. It is common in the world of mathematics to be extraordinarily careful proving theorems that justify acting more casually in practice.

The proof of the continuous master theorem involves two lemmas. Lemma 4.2 uses a slightly simplified master recurrence with a threshold constant of $n_0 = 1$, rather than the more general $n_0 > 0$ threshold constant implied by the unstated base case. The lemma employs a recursion tree to reduce the solution of the simplified master recurrence to that of evaluating a summation. Lemma 4.3 then provides asymptotic bounds for the summation, mirroring the three cases of the master theorem. Finally, the continuous master theorem itself (Theorem 4.4) gives asymptotic bounds for master recurrences, while generalizing to an arbitrary threshold constant $n_0 > 0$ as implied by the unstated base case.

[1] This terminology does not mean that either $T(n)$ or $f(n)$ need be continuous, only that the domain of $T(n)$ is the real numbers, as opposed to integers.

Some of the proofs use the properties described in Problem 3-5 on pages 72–73 to combine and simplify complicated asymptotic expressions. Although Problem 3-5 addresses only Θ-notation, the properties enumerated there can be extended to O-notation and Ω-notation as well.

Here's the first lemma.

Lemma 4.2

Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a function defined over real numbers $n \geq 1$. Then the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } 0 \leq n < 1 \,, \\ aT(n/b) + f(n) & \text{if } n \geq 1 \end{cases}$$

has solution

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) \,. \tag{4.18}$$

Proof Consider the recursion tree in Figure 4.3. Let's look first at its internal nodes. The root of the tree has cost $f(n)$, and it has a children, each with cost $f(n/b)$. (It is convenient to think of a as being an integer, especially when visualizing the recursion tree, but the mathematics does not require it.) Each of these children has a children, making a^2 nodes at depth 2, and each of the a children has cost $f(n/b^2)$. In general, there are a^j nodes at depth j, and each node has cost $f(n/b^j)$.

Now, let's move on to understanding the leaves. The tree grows downward until n/b^j becomes less than 1. Thus, the tree has height $\lfloor \log_b n \rfloor + 1$, because $n/b^{\lfloor \log_b n \rfloor} \geq n/b^{\log_b n} = 1$ and $n/b^{\lfloor \log_b n \rfloor + 1} < n/b^{\log_b n} = 1$. Since, as we have observed, the number of nodes at depth j is a^j and all the leaves are at depth $\lfloor \log_b n \rfloor + 1$, the tree contains $a^{\lfloor \log_b n \rfloor + 1}$ leaves. Using the identity (3.21) on page 66, we have $a^{\lfloor \log_b n \rfloor + 1} \leq a^{\log_b n + 1} = an^{\log_b a} = O(n^{\log_b a})$, since a is constant, and $a^{\lfloor \log_b n \rfloor + 1} \geq a^{\log_b n} = n^{\log_b a} = \Omega(n^{\log_b a})$. Consequently, the total number of leaves is $\Theta(n^{\log_b a})$—asymptotically, the watershed function.

We are now in a position to derive equation (4.18) by summing the costs of the nodes at each depth in the tree, as shown in the figure. The first term in the equation is the total costs of the leaves. Since each leaf is at depth $\lfloor \log_b n \rfloor + 1$ and $n/b^{\lfloor \log_b n \rfloor + 1} < 1$, the base case of the recurrence gives the cost of a leaf: $T(n/b^{\lfloor \log_b n \rfloor + 1}) = \Theta(1)$. Hence the cost of all $\Theta(n^{\log_b a})$ leaves is $\Theta(n^{\log_b a}) \cdot \Theta(1) = \Theta(n^{\log_b a})$ by Problem 3-5(d). The second term in equation (4.18) is the cost of the internal nodes, which, in the underlying divide-and-conquer algorithm, represents the costs of dividing problems into subproblems and

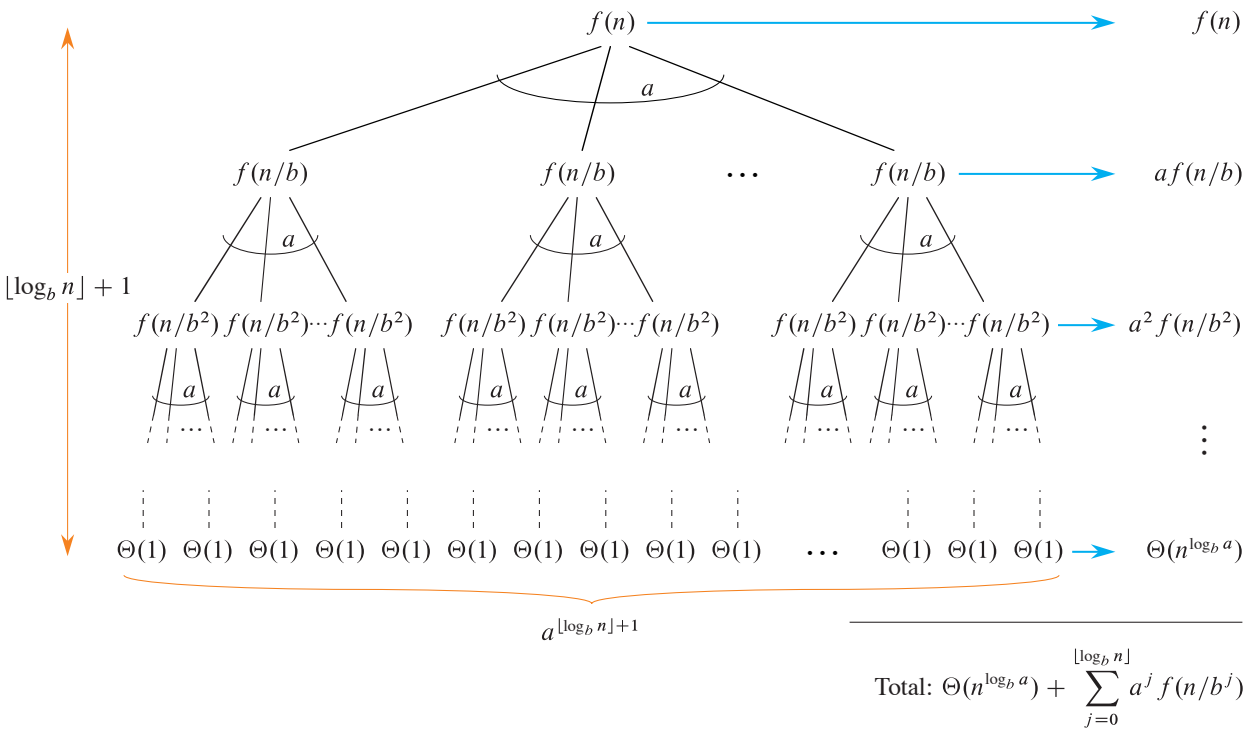

Figure 4.3 The recursion tree generated by $T(n) = aT(n/b) + f(n)$. The tree is a complete a-ary tree with $a^{\lfloor \log_b n \rfloor + 1}$ leaves and height $\lfloor \log_b n \rfloor + 1$. The cost of the nodes at each depth is shown at the right, and their sum is given in equation (4.18).

then recombining the subproblems. Since the cost for all the internal nodes at depth j is $a^j f(n/b^j)$, the total cost of all internal nodes is

$$\sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) \; .$$

■

As we'll see, the three cases of the master theorem depend on the distribution of the total cost across levels of the recursion tree:

Case 1: The costs increase geometrically from the root to the leaves, growing by a constant factor with each level.

Case 2: The costs depend on the value of k in the theorem. With $k = 0$, the costs are equal for each level; with $k = 1$, the costs grow linearly from the root to the leaves; with $k = 2$, the growth is quadratic; and in general, the costs grow polynomially in k.

Case 3: The costs decrease geometrically from the root to the leaves, shrinking by a constant factor with each level.

The summation in equation (4.18) describes the cost of the dividing and combining steps in the underlying divide-and-conquer algorithm. The next lemma provides asymptotic bounds on the summation's growth.

Lemma 4.3
Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a function defined over real numbers $n \geq 1$. Then the asymptotic behavior of the function

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) , \tag{4.19}$$

defined for $n \geq 1$, can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $g(n) = O(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $g(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant c in the range $0 < c < 1$ such that $0 < a f(n/b) \leq c f(n)$ for all $n \geq 1$, then $g(n) = \Theta(f(n))$.

Proof For case 1, we have $f(n) = O(n^{\log_b a - \epsilon})$, which implies that $f(n/b^j) = O((n/b^j)^{\log_b a - \epsilon})$. Substituting into equation (4.19) yields

$$\begin{aligned}
g(n) &= \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j O\left(\left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right) \\
&= O\left(\sum_{j=0}^{\lfloor \log_b n \rfloor} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right) && \text{(by Problem 3-5(c), repeatedly)} \\
&= O\left(n^{\log_b a - \epsilon} \sum_{j=0}^{\lfloor \log_b n \rfloor} \left(\frac{a b^\epsilon}{b^{\log_b a}}\right)^j\right) \\
&= O\left(n^{\log_b a - \epsilon} \sum_{j=0}^{\lfloor \log_b n \rfloor} (b^\epsilon)^j\right) && \text{(by equation (3.17) on page 66)} \\
&= O\left(n^{\log_b a - \epsilon} \left(\frac{b^{\epsilon(\lfloor \log_b n \rfloor + 1)} - 1}{b^\epsilon - 1}\right)\right) && \text{(by equation (A.6) on page 1142)} ,
\end{aligned}$$

the last series being geometric. Since b and ϵ are constants, the $b^\epsilon - 1$ denominator doesn't affect the asymptotic growth of $g(n)$, and neither does the -1 in

the numerator. Since $b^{\epsilon(\lfloor \log_b n \rfloor + 1)} \le (b^{\log_b n + 1})^{\epsilon} = b^{\epsilon} n^{\epsilon} = O(n^{\epsilon})$, we obtain $g(n) = O(n^{\log_b a - \epsilon} \cdot O(n^{\epsilon})) = O(n^{\log_b a})$, thereby proving case 1.

Case 2 assumes that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, from which we can conclude that $f(n/b^j) = \Theta((n/b^j)^{\log_b a} \lg^k (n/b^j))$. Substituting into equation (4.19) and repeatedly applying Problem 3-5(c) yields

$$
\begin{aligned}
g(n) &= \Theta\left(\sum_{j=0}^{\lfloor \log_b n \rfloor} a^j \left(\frac{n}{b^j}\right)^{\log_b a} \lg^k \left(\frac{n}{b^j}\right)\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \frac{a^j}{b^{j \log_b a}} \lg^k \left(\frac{n}{b^j}\right)\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \lg^k \left(\frac{n}{b^j}\right)\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \left(\frac{\log_b (n/b^j)}{\log_b 2}\right)^k\right) && \text{(by equation (3.19) on page 66)} \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \left(\frac{\log_b n - j}{\log_b 2}\right)^k\right) && \text{(by equations (3.17), (3.18), and (3.20))} \\
&= \Theta\left(\frac{n^{\log_b a}}{\log_b^k 2} \sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k\right) && (b > 1 \text{ and } k \text{ are constants}) .
\end{aligned}
$$

The summation within the Θ-notation can be bounded from above as follows:

$$
\begin{aligned}
\sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k &\le \sum_{j=0}^{\lfloor \log_b n \rfloor} (\lfloor \log_b n \rfloor + 1 - j)^k \\
&= \sum_{j=1}^{\lfloor \log_b n \rfloor + 1} j^k && \text{(reindexing—pages 1143–1144)} \\
&= O((\lfloor \log_b n \rfloor + 1)^{k+1}) && \text{(by Exercise A.1-5 on page 1144)} \\
&= O(\log_b^{k+1} n) && \text{(by Exercise 3.3-3 on page 70) .}
\end{aligned}
$$

Exercise 4.6-1 asks you to show that the summation can similarly be bounded from below by $\Omega(\log_b^{k+1} n)$. Since we have tight upper and lower bounds, the summation is $\Theta(\log_b^{k+1} n)$, from which we can conclude that $g(n) = \Theta\left(n^{\log_b a} \log_b^{k+1} n\right)$, thereby completing the proof of case 2.

For case 3, observe that $f(n)$ appears in the definition (4.19) of $g(n)$ (when $j = 0$) and that all terms of $g(n)$ are positive. Therefore, we must have $g(n) = \Omega(f(n))$, and it only remains to prove that $g(n) = O(f(n))$. Performing j iterations of the inequality $af(n/b) \le cf(n)$ yields $a^j f(n/b^j) \le c^j f(n)$. Substituting into equation (4.19), we obtain

$$\begin{aligned} g(n) &= \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) \\ &\le \sum_{j=0}^{\lfloor \log_b n \rfloor} c^j f(n) \\ &\le f(n) \sum_{j=0}^{\infty} c^j \\ &= f(n) \left(\frac{1}{1-c} \right) \qquad \text{(by equation (A.7) on page 1142 since } |c| < 1) \\ &= O(f(n)) \, . \end{aligned}$$

Thus, we can conclude that $g(n) = \Theta(f(n))$. With case 3 proved, the entire proof of the lemma is complete. ■

We can now state and prove the continuous master theorem.

Theorem 4.4 (Continuous 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 algorithmic recurrence $T(n)$ on the positive real numbers by

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

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 $k \ge 0$ 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 $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Proof The idea is to bound the summation (4.18) from Lemma 4.2 by applying Lemma 4.3. But we must account for Lemma 4.2 using a base case for $0 < n < 1$,

whereas this theorem uses an implicit base case for $0 < n < n_0$, where $n_0 > 0$ is an arbitrary threshold constant. Since the recurrence is algorithmic, we can assume that $f(n)$ is defined for $n \geq n_0$.

For $n > 0$, let us define two auxiliary functions $T'(n) = T(n_0 n)$ and $f'(n) = f(n_0 n)$. We have

$$\begin{aligned} T'(n) &= T(n_0 n) \\ &= \begin{cases} \Theta(1) & \text{if } n_0 n < n_0 , \\ aT(n_0 n/b) + f(n_0 n) & \text{if } n_0 n \geq n_0 \end{cases} \\ &= \begin{cases} \Theta(1) & \text{if } n < 1 , \\ aT'(n/b) + f'(n) & \text{if } n \geq 1 . \end{cases} \end{aligned} \tag{4.20}$$

We have obtained a recurrence for $T'(n)$ that satisfies the conditions of Lemma 4.2, and by that lemma, the solution is

$$T'(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f'(n/b^j) . \tag{4.21}$$

To solve $T'(n)$, we first need to bound $f'(n)$. Let's examine the individual cases in the theorem.

The condition for case 1 is $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$. We have

$$\begin{aligned} f'(n) &= f(n_0 n) \\ &= O((n_0 n)^{\log_b a - \epsilon}) \\ &= O(n^{\log_b a - \epsilon}) , \end{aligned}$$

since a, b, n_0, and ϵ are all constant. The function $f'(n)$ satisfies the conditions of case 1 of Lemma 4.3, and the summation in equation (4.18) of Lemma 4.2 evaluates to $O(n^{\log_b a})$. Because a, b and n_0 are all constants, we have

$$\begin{aligned} T(n) &= T'(n/n_0) \\ &= \Theta((n/n_0)^{\log_b a}) + O((n/n_0)^{\log_b a}) \\ &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}) \qquad \text{(by Problem 3-5(b))} , \end{aligned}$$

thereby completing case 1 of the theorem.

The condition for case 2 is $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$. We have

$$\begin{aligned} f'(n) &= f(n_0 n) \\ &= \Theta((n_0 n)^{\log_b a} \lg^k (n_0 n)) \\ &= \Theta(n^{\log_b a} \lg^k n) \qquad \text{(by eliminating the constant terms)} . \end{aligned}$$

Similar to the proof of case 1, the function $f'(n)$ satisfies the conditions of case 2 of Lemma 4.3. The summation in equation (4.18) of Lemma 4.2 is therefore $\Theta(n^{\log_b a} \lg^{k+1} n)$, which implies that

$$\begin{aligned} T(n) &= T'(n/n_0) \\ &= \Theta((n/n_0)^{\log_b a}) + \Theta((n/n_0)^{\log_b a} \lg^{k+1}(n/n_0)) \\ &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg^{k+1} n) \\ &= \Theta(n^{\log_b a} \lg^{k+1} n) \qquad \text{(by Problem 3-5(c))} , \end{aligned}$$

which proves case 2 of the theorem.

Finally, the condition for case 3 is $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ additionally satisfies the regularity condition $af(n/b) \le cf(n)$ for all $n \ge n_0$ and some constants $c < 1$ and $n_0 > 1$. The first part of case 3 is like case 1:

$$\begin{aligned} f'(n) &= f(n_0 n) \\ &= \Omega((n_0 n)^{\log_b a+\epsilon}) \\ &= \Omega(n^{\log_b a+\epsilon}) . \end{aligned}$$

Using the definition of $f'(n)$ and the fact that $n_0 n \ge n_0$ for all $n \ge 1$, we have for $n \ge 1$ that

$$\begin{aligned} af'(n/b) &= af(n_0 n/b) \\ &\le cf(n_0 n) \\ &= cf'(n) . \end{aligned}$$

Thus $f'(n)$ satisfies the requirements for case 3 of Lemma 4.3, and the summation in equation (4.18) of Lemma 4.2 evaluates to $\Theta(f'(n))$, yielding

$$\begin{aligned} T(n) &= T'(n/n_0) \\ &= \Theta((n/n_0)^{\log_b a}) + \Theta(f'(n/n_0)) \\ &= \Theta(f'(n/n_0)) \\ &= \Theta(f(n)) , \end{aligned}$$

which completes the proof of case 3 of the theorem and thus the whole theorem. ■

Exercises

4.6-1

Show that $\sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k = \Omega(\log_b^{k+1} n)$.

★ *4.6-2*

Show that case 3 of the master theorem is overstated (which is also why case 3 of Lemma 4.3 does not require that $f(n) = \Omega(n^{\log_b a+\epsilon})$) in the sense that the

regularity condition $af(n/b) \le cf(n)$ for some constant $c < 1$ implies that there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$.

★ ***4.6-3***
For $f(n) = \Theta(n^{\log_b a}/\lg n)$, prove that the summation in equation (4.19) has solution $g(n) = \Theta(n^{\log_b a} \lg\lg n)$. Conclude that a master recurrence $T(n)$ using $f(n)$ as its driving function has solution $T(n) = \Theta(n^{\log_b a} \lg\lg n)$.

★ 4.7 Akra-Bazzi recurrences

This section provides an overview of two advanced topics related to divide-and-conquer recurrences. The first deals with technicalities arising from the use of floors and ceilings, and the second discusses the Akra-Bazzi method, which involves a little calculus, for solving complicated divide-and-conquer recurrences.

In particular, we'll look at the class of algorithmic divide-and-conquer recurrences originally studied by M. Akra and L. Bazzi [13]. These ***Akra-Bazzi*** recurrences take the form

$$T(n) = f(n) + \sum_{i=1}^{k} a_i T(n/b_i) \,, \tag{4.22}$$

where k is a positive integer; all the constants $a_1, a_2, \ldots, a_k \in \mathbb{R}$ are strictly positive; all the constants $b_1, b_2, \ldots, b_k \in \mathbb{R}$ are strictly greater than 1; and the driving function $f(n)$ is defined on sufficiently large nonnegative reals and is itself nonnegative.

Akra-Bazzi recurrences generalize the class of recurrences addressed by the master theorem. Whereas master recurrences characterize the running times of divide-and-conquer algorithms that break a problem into equal-sized subproblems (modulo floors and ceilings), Akra-Bazzi recurrences can describe the running time of divide-and-conquer algorithms that break a problem into different-sized subproblems. The master theorem, however, allows you to ignore floors and ceilings, but the Akra-Bazzi method for solving Akra-Bazzi recurrences needs an additional requirement to deal with floors and ceilings.

But before diving into the Akra-Bazzi method itself, let's understand the limitations involved in ignoring floors and ceilings in Akra-Bazzi recurrences. As you're aware, algorithms generally deal with integer-sized inputs. The mathematics for recurrences is often easier with real numbers, however, than with integers, where we must cope with floors and ceilings to ensure that terms are well defined. The difference may not seem to be much—especially because that's often the truth with recurrences—but to be mathematically correct, we must be careful with our

assumptions. Since our end goal is to understand algorithms and not the vagaries of mathematical corner cases, we'd like to be casual yet rigorous. How can we treat floors and ceilings casually while still ensuring rigor?

From a mathematical point of view, the difficulty in dealing with floors and ceilings is that some driving functions can be really, really weird. So it's not okay in general to ignore floors and ceilings in Akra-Bazzi recurrences. Fortunately, most of the driving functions we encounter in the study of algorithms behave nicely, and floors and ceilings don't make a difference.

The polynomial-growth condition

If the driving function $f(n)$ in equation (4.22) is well behaved in the following sense, it's okay to drop floors and ceilings.

> A function $f(n)$ defined on all sufficiently large positive reals satisfies the ***polynomial-growth condition*** if there exists a constant $\hat{n} > 0$ such that the following holds: for every constant $\phi \geq 1$, there exists a constant $d > 1$ (depending on ϕ) such that $f(n)/d \leq f(\psi n) \leq df(n)$ for all $1 \leq \psi \leq \phi$ and $n \geq \hat{n}$.

This definition may be one of the hardest in this textbook to get your head around. To a first order, it says that $f(n)$ satisfies the property that $f(\Theta(n)) = \Theta(f(n))$, although the polynomial-growth condition is actually somewhat stronger (see Exercise 4.7-4). The definition also implies that $f(n)$ is asymptotically positive (see Exercise 4.7-3).

Examples of functions that satisfy the polynomial-growth condition include any function of the form $f(n) = \Theta(n^\alpha \lg^\beta n \lg\lg^\gamma n)$, where α, β, and γ are constants. Most of the polynomially bounded functions used in this book satisfy the condition. Exponentials and superexponentials do not (see Exercise 4.7-2, for example), and there also exist polynomially bounded functions that do not.

Floors and ceilings in "nice" recurrences

When the driving function in an Akra-Bazzi recurrence satisfies the polynomial-growth condition, floors and ceilings don't change the asymptotic behavior of the solution. The following theorem, which is presented without proof, formalizes this notion.

Theorem 4.5

Let $T(n)$ be a function defined on the nonnegative reals that satisfies recurrence (4.22), where $f(n)$ satisfies the polynomial-growth condition. Let $T'(n)$ be another function defined on the natural numbers also satisfying recurrence (4.22),

except that each $T(n/b_i)$ is replaced either with $T(\lceil n/b_i \rceil)$ or with $T(\lfloor n/b_i \rfloor)$. Then we have $T'(n) = \Theta(T(n))$. ■

Floors and ceilings represent a minor perturbation to the arguments in the recursion. By inequality (3.2) on page 64, they perturb an argument by at most 1. But much larger perturbations are tolerable. As long as the driving function $f(n)$ in recurrence (4.22) satisfies the polynomial-growth condition, it turns out that replacing any term $T(n/b_i)$ with $T(n/b_i + h_i(n))$, where $|h_i(n)| = O(n/\lg^{1+\epsilon} n)$ for some constant $\epsilon > 0$ and sufficiently large n, leaves the asymptotic solution unaffected. Thus, the divide step in a divide-and-conquer algorithm can be moderately coarse without affecting the solution to its running-time recurrence.

The Akra-Bazzi method

The Akra-Bazzi method, not surprisingly, was developed to solve Akra-Bazzi recurrences (4.22), which by dint of Theorem 4.5, applies in the presence of floors and ceilings or even larger perturbations, as just discussed. The method involves first determining the unique real number p such that $\sum_{i=1}^{k} a_i/b_i^p = 1$. Such a p always exists, because when $p \to -\infty$, the sum goes to ∞; it decreases as p increases; and when $p \to \infty$, it goes to 0. The Akra-Bazzi method then gives the solution to the recurrence as

$$T(n) = \Theta\left(n^p \left(1 + \int_1^n \frac{f(x)}{x^{p+1}}\,dx\right)\right) . \tag{4.23}$$

As an example, consider the recurrence

$$T(n) = T(n/5) + T(7n/10) + n . \tag{4.24}$$

We'll see the similar recurrence (9.1) on page 240 when we study an algorithm for selecting the ith smallest element from a set of n numbers. This recurrence has the form of equation (4.22), where $a_1 = a_2 = 1$, $b_1 = 5$, $b_2 = 10/7$, and $f(n) = n$. To solve it, the Akra-Bazzi method says that we should determine the unique p satisfying

$$\left(\frac{1}{5}\right)^p + \left(\frac{7}{10}\right)^p = 1 .$$

Solving for p is kind of messy—it turns out that $p = 0.83978\ldots$—but we can solve the recurrence without actually knowing the exact value for p. Observe that $(1/5)^0 + (7/10)^0 = 2$ and $(1/5)^1 + (7/10)^1 = 9/10$, and thus p lies in the range $0 < p < 1$. That turns out to be sufficient for the Akra-Bazzi method to give us the solution. We'll use the fact from calculus that if $k \neq -1$, then $\int x^k dx = x^{k+1}/(k+1)$, which we'll apply with $k = -p \neq -1$. The Akra-Bazzi

solution (4.23) gives us

$$\begin{aligned}
T(n) &= \Theta\left(n^p\left(1+\int_1^n \frac{f(x)}{x^{p+1}}\,dx\right)\right) \\
&= \Theta\left(n^p\left(1+\int_1^n x^{-p}\,dx\right)\right) \\
&= \Theta\left(n^p\left(1+\left[\frac{x^{1-p}}{1-p}\right]_1^n\right)\right) \\
&= \Theta\left(n^p\left(1+\left(\frac{n^{1-p}}{1-p}-\frac{1}{1-p}\right)\right)\right) \\
&= \Theta\left(n^p\cdot\Theta(n^{1-p})\right) && \text{(because } 1-p \text{ is a positive constant)} \\
&= \Theta(n) && \text{(by Problem 3-5(d))} .
\end{aligned}$$

Although the Akra-Bazzi method is more general than the master theorem, it requires calculus and sometimes a bit more reasoning. You also must ensure that your driving function satisfies the polynomial-growth condition if you want to ignore floors and ceilings, although that's rarely a problem. When it applies, the master method is much simpler to use, but only when subproblem sizes are more or less equal. They are both good tools for your algorithmic toolkit.

Exercises

★ ***4.7-1***

Consider an Akra-Bazzi recurrence $T(n)$ on the reals as given in recurrence (4.22), and define $T'(n)$ as

$$T'(n) = cf(n) + \sum_{i=1}^{k} a_i T'(n/b_i) ,$$

where $c > 0$ is constant. Prove that whatever the implicit initial conditions for $T(n)$ might be, there exist initial conditions for $T'(n)$ such that $T'(n) = cT(n)$ for all $n > 0$. Conclude that we can drop the asymptotics on a driving function in any Akra-Bazzi recurrence without affecting its asymptotic solution.

4.7-2

Show that $f(n) = n^2$ satisfies the polynomial-growth condition but that $f(n) = 2^n$ does not.

4.7-3

Let $f(n)$ be a function that satisfies the polynomial-growth condition. Prove that $f(n)$ is asymptotically positive, that is, there exists a constant $n_0 \geq 0$ such that $f(n) \geq 0$ for all $n \geq n_0$.

★ ***4.7-4***
Give an example of a function $f(n)$ that does not satisfy the polynomial-growth condition but for which $f(\Theta(n)) = \Theta(f(n))$.

4.7-5
Use the Akra-Bazzi method to solve the following recurrences.

a. $T(n) = T(n/2) + T(n/3) + T(n/6) + n \lg n$.

b. $T(n) = 3T(n/3) + 8T(n/4) + n^2/\lg n$.

c. $T(n) = (2/3)T(n/3) + (1/3)T(2n/3) + \lg n$.

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

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

★ ***4.7-6***
Use the Akra-Bazzi method to prove the continuous master theorem.

Problems

4-1 Recurrence examples
Give asymptotically tight upper and lower bounds for $T(n)$ in each of the following algorithmic recurrences. Justify your answers.

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

b. $T(n) = T(8n/11) + n$.

c. $T(n) = 16T(n/4) + n^2$.

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

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

f. $T(n) = 7T(n/2) + n^2 \lg n$.

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

h. $T(n) = T(n-2) + n^2$.

4-2 Parameter-passing costs

Throughout this book, we assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three parameter-passing strategies:

1. Arrays are passed by pointer. Time $= \Theta(1)$.
2. Arrays are passed by copying. Time $= \Theta(N)$, where N is the size of the array.
3. Arrays are passed by copying only the subrange that might be accessed by the called procedure. Time $= \Theta(n)$ if the subarray contains n elements.

Consider the following three algorithms:

a. The recursive binary-search algorithm for finding a number in a sorted array (see Exercise 2.3-6).

b. The MERGE-SORT procedure from Section 2.3.1.

c. The MATRIX-MULTIPLY-RECURSIVE procedure from Section 4.1.

Give nine recurrences $T_{a1}(N, n), T_{a2}(N, n), \ldots, T_{c3}(N, n)$ for the worst-case running times of each of the three algorithms above when arrays and matrices are passed using each of the three parameter-passing strategies above. Solve your recurrences, giving tight asymptotic bounds.

4-3 Solving recurrences with a change of variables

Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one you have seen before. Let's solve the recurrence

$$T(n) = 2T\left(\sqrt{n}\right) + \Theta(\lg n) \tag{4.25}$$

by using the change-of-variables method.

a. Define $m = \lg n$ and $S(m) = T(2^m)$. Rewrite recurrence (4.25) in terms of m and $S(m)$.

b. Solve your recurrence for $S(m)$.

c. Use your solution for $S(m)$ to conclude that $T(n) = \Theta(\lg n \lg\lg n)$.

d. Sketch the recursion tree for recurrence (4.25), and use it to explain intuitively why the solution is $T(n) = \Theta(\lg n \lg\lg n)$.

Solve the following recurrences by changing variables:

e. $T(n) = 2T(\sqrt{n}) + \Theta(1)$.

f. $T(n) = 3T(\sqrt[3]{n}) + \Theta(n)$.

4-4 More recurrence examples

Give asymptotically tight upper and lower bounds for $T(n)$ in each of the following recurrences. Justify your answers.

a. $T(n) = 5T(n/3) + n \lg n$.

b. $T(n) = 3T(n/3) + n/\lg n$.

c. $T(n) = 8T(n/2) + n^3\sqrt{n}$.

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

e. $T(n) = 2T(n/2) + n/\lg n$.

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

g. $T(n) = T(n-1) + 1/n$.

h. $T(n) = T(n-1) + \lg n$.

i. $T(n) = T(n-2) + 1/\lg n$.

j. $T(n) = \sqrt{n}\,T(\sqrt{n}) + n$.

4-5 Fibonacci numbers

This problem develops properties of the Fibonacci numbers, which are defined by recurrence (3.31) on page 69. We'll explore the technique of generating functions to solve the Fibonacci recurrence. Define the ***generating function*** (or ***formal power series***) $\mathcal{F}$ as

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \cdots ,\end{aligned}$$

where F_i is the ith Fibonacci number.

a. Show that $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

b. Show that

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1-z-z^2} \\ &= \frac{z}{(1-\phi z)(1-\widehat{\phi} z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\widehat{\phi} z}\right) ,\end{aligned}$$

where ϕ is the golden ratio, and $\widehat{\phi}$ is its conjugate (see page 69).

c. Show that

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \widehat{\phi}^i) z^i .$$

You may use without proof the generating-function version of equation (A.7) on page 1142, $\sum_{k=0}^{\infty} x^k = 1/(1-x)$. Because this equation involves a generating function, x is a formal variable, not a real-valued variable, so that you don't have to worry about convergence of the summation or about the requirement in equation (A.7) that $|x| < 1$, which doesn't make sense here.

d. Use part (c) to prove that $F_i = \phi^i/\sqrt{5}$ for $i > 0$, rounded to the nearest integer. (*Hint:* Observe that $\left|\widehat{\phi}\right| < 1$.)

e. Prove that $F_{i+2} \geq \phi^i$ for $i \geq 0$.

4-6 Chip testing

Professor Diogenes has n supposedly identical integrated-circuit chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the professor cannot trust the answer of a bad chip. Thus, the four possible outcomes of a test are as follows:

Chip A says	Chip B says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

a. Show that if at least $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

Now you will design an algorithm to identify which chips are good and which are bad, assuming that more than $n/2$ of the chips are good. First, you will determine how to identify one good chip.

b. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size. That is, show how to use $\lfloor n/2 \rfloor$ pairwise tests to obtain a set with at most $\lceil n/2 \rceil$ chips that still has the property that more than half of the chips are good.

c. Show how to apply the solution to part (b) recursively to identify one good chip. Give and solve the recurrence that describes the number of tests needed to identify one good chip.

You have now determined how to identify one good chip.

d. Show how to identify all the good chips with an additional $\Theta(n)$ pairwise tests.

4-7 Monge arrays

An $m \times n$ array A of real numbers is a ***Monge array*** if for all i, j, k, and l such that $1 \le i < k \le m$ and $1 \le j < l \le n$, we have

$$A[i, j] + A[k, l] \le A[i, l] + A[k, j] .$$

In other words, whenever we pick two rows and two columns of a Monge array and consider the four elements at the intersections of the rows and the columns, the sum of the upper-left and lower-right elements is less than or equal to the sum of the lower-left and upper-right elements. For example, the following array is Monge:

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

a. Prove that an array is Monge if and only if for all $i = 1, 2, ..., m - 1$ and $j = 1, 2, ..., n - 1$, we have

$$A[i, j] + A[i + 1, j + 1] \le A[i, j + 1] + A[i + 1, j] .$$

(*Hint:* For the "if" part, use induction separately on rows and columns.)

b. The following array is not Monge. Change one element in order to make it Monge. (*Hint:* Use part (a).)

37 23 22 32
21 6 7 10
53 34 30 31
32 13 9 6
43 21 15 8

c. Let $f(i)$ be the index of the column containing the leftmost minimum element of row i. Prove that $f(1) \leq f(2) \leq \cdots \leq f(m)$ for any $m \times n$ Monge array.

d. Here is a description of a divide-and-conquer algorithm that computes the leftmost minimum element in each row of an $m \times n$ Monge array A:

> Construct a submatrix A' of A consisting of the even-numbered rows of A. Recursively determine the leftmost minimum for each row of A'. Then compute the leftmost minimum in the odd-numbered rows of A.

Explain how to compute the leftmost minimum in the odd-numbered rows of A (given that the leftmost minimum of the even-numbered rows is known) in $O(m + n)$ time.

e. Write the recurrence for the running time of the algorithm in part (d). Show that its solution is $O(m + n \log m)$.

Chapter notes

Divide-and-conquer as a technique for designing algorithms dates back at least to 1962 in an article by Karatsuba and Ofman [242], but it might have been used well before then. According to Heideman, Johnson, and Burrus [211], C. F. Gauss devised the first fast Fourier transform algorithm in 1805, and Gauss's formulation breaks the problem into smaller subproblems whose solutions are combined.

Strassen's algorithm [424] caused much excitement when it appeared in 1969. Before then, few imagined the possibility of an algorithm asymptotically faster than the basic MATRIX-MULTIPLY procedure. Shortly thereafter, S. Winograd reduced the number of submatrix additions from 18 to 15 while still using seven submatrix multiplications. This improvement, which Winograd apparently never published (and which is frequently miscited in the literature), may enhance the practicality of the method, but it does not affect its asymptotic performance. Probert [368] described Winograd's algorithm and showed that with seven multiplications, 15 additions is the minimum possible.

Strassen's $\Theta(n^{\lg 7}) = O(n^{2.81})$ bound for matrix multiplication held until 1987, when Coppersmith and Winograd [103] made a significant advance, improving the

bound to $O(n^{2.376})$ time with a mathematically sophisticated but wildly impractical algorithm based on tensor products. It took approximately 25 years before the asymptotic upper bound was again improved. In 2012 Vassilevska Williams [445] improved it to $O(n^{2.37287})$, and two years later Le Gall [278] achieved $O(n^{2.37286})$, both of them using mathematically fascinating but impractical algorithms. The best lower bound to date is just the obvious $\Omega(n^2)$ bound (obvious because any algorithm for matrix multiplication must fill in the n^2 elements of the product matrix).

The performance of MATRIX-MULTIPLY-RECURSIVE can be improved in practice by coarsening the leaves of the recursion. It also exhibits better cache behavior than MATRIX-MULTIPLY, although MATRIX-MULTIPLY can be improved by "tiling." Leiserson et al. [293] conducted a performance-engineering study of matrix multiplication in which a parallel and vectorized divide-and-conquer algorithm achieved the highest performance. Strassen's algorithm can be practical for large dense matrices, although large matrices tend to be sparse, and sparse methods can be much faster. When using limited-precision floating-point values, Strassen's algorithm produces larger numerical errors than the $\Theta(n^3)$ algorithms do, although Higham [215] demonstrated that Strassen's algorithm is amply accurate for some applications.

Recurrences were studied as early as 1202 by Leonardo Bonacci [66], also known as Fibonacci, for whom the Fibonacci numbers are named, although Indian mathematicians had discovered Fibonacci numbers centuries before. The French mathematician De Moivre [108] introduced the method of generating functions with which he studied Fibonacci numbers (see Problem 4-5). Knuth [259] and Liu [302] are good resources for learning the method of generating functions.

Aho, Hopcroft, and Ullman [5, 6] offered one of the first general methods for solving recurrences arising from the analysis of divide-and-conquer algorithms. The master method was adapted from Bentley, Haken, and Saxe [52]. The Akra-Bazzi method is due (unsurprisingly) to Akra and Bazzi [13]. Divide-and-conquer recurrences have been studied by many researchers, including Campbell [79], Graham, Knuth, and Patashnik [199], Kuszmaul and Leiserson [274], Leighton [287], Purdom and Brown [371], Roura [389], Verma [447], and Yap [462].

The issue of floors and ceilings in divide-and-conquer recurrences, including a theorem similar to Theorem 4.5, was studied by Leighton [287]. Leighton proposed a version of the polynomial-growth condition. Campbell [79] removed several limitations in Leighton's statement of it and showed that there were polynomially bounded functions that do not satisfy Leighton's condition. Campbell also carefully studied many other technical issues, including the well-definedness of divide-and-conquer recurrences. Kuszmaul and Leiserson [274] provided a proof of Theorem 4.5 that does not involve calculus or other higher math. Both Campbell and Leighton explored the perturbations of arguments beyond simple floors and ceilings.