# Lecture 5

- The divide-n-conquer strategy.

The divide-n-conquer strategy subdivides a problem in to sub-problems whose sizes are typically non-constant functions of the original size. We then solve those sub-problems and compose those solutions for a solution to the original problem. Binary search, which we discussed in Lecture 2, is a good example of the deployment of this strategy. In binary search, given a problem of size $n$, we are left with one sub-problem of size $n/2$, which is a linear-function in $n$.

Following is CLRS's take on what the strategy is. As it mentions, merge sort, is a(nother) good example that adopts this strategy. We reproduce the merge sort algorithm from CLRS as well. After, we reproduce the chapter from CLRS on how to analyze algorithms that involve recursion, which algorithms based on divide-n-conquer often do. Then, we reproduce Quick sort, and in particular discuss a very useful subroutine it employes, partition. We then discuss so-called order statistics, e.g., the median, and how partition turns out to be powerful, provided it is parameterized well.

In this section, we examine an alternative design approach, known as "divide-and-conquer." We shall use divide-and-conquer to design a sorting algorithm whose worst-case running time is much less than that of insertion sort. One advantage of divide-and-conquer algorithms is that their running times are often easily determined using techniques that will be introduced in Chapter 4.

### 2.3.1 The divide-and-conquer approach

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a ***divide-and-conquer*** approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

**Divide** the problem into a number of subproblems.

**Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

**Combine** the solutions to the subproblems into the solution for the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

**Divide:** Divide the $n$-element sequence to be sorted into two subsequences of $n/2$ elements each.

**Conquer:** Sort the two subsequences recursively using merge sort.

**Combine:** Merge the two sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. To perform the merging, we use an auxiliary procedure MERGE$(A, p, q, r)$, where $A$ is an array and $p$, $q$, and $r$ are indices numbering elements of the array such that $p \le q < r$. The procedure assumes that the subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are in sorted order. It ***merges*** them to form a single sorted subarray that replaces the current subarray $A[p\,..\,r]$.

Our MERGE procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the number of elements being merged, and it works as follows. Returning to our card-playing

motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile. Computationally, each basic step takes constant time, since we are checking just two top cards. Since we perform at most $n$ basic steps, merging takes $\Theta(n)$ time.

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. The idea is to put on the bottom of each pile a ***sentinel*** card, which contains a special value that we use to simplify our code. Here, we use $\infty$ as the sentinel value, so that whenever a card with $\infty$ is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly $r - p + 1$ cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

```
MERGE(A, p, q, r)
 1  n_1 ← q − p + 1
 2  n_2 ← r − q
 3  create arrays L[1 .. n_1 + 1] and R[1 .. n_2 + 1]
 4  for i ← 1 to n_1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n_2
 7       do R[j] ← A[q + j]
 8  L[n_1 + 1] ← ∞
 9  R[n_2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14            then A[k] ← L[i]
15                 i ← i + 1
16            else A[k] ← R[j]
17                 j ← j + 1
```

In detail, the MERGE procedure works as follows. Line 1 computes the length $n_1$ of the subarray $A[p..q]$, and line 2 computes the length $n_2$ of the subarray $A[q + 1..r]$. We create arrays $L$ and $R$ ("left" and "right"), of lengths $n_1 + 1$ and $n_2 + 1$, respectively, in line 3. The **for** loop of lines 4–5 copies the subar-

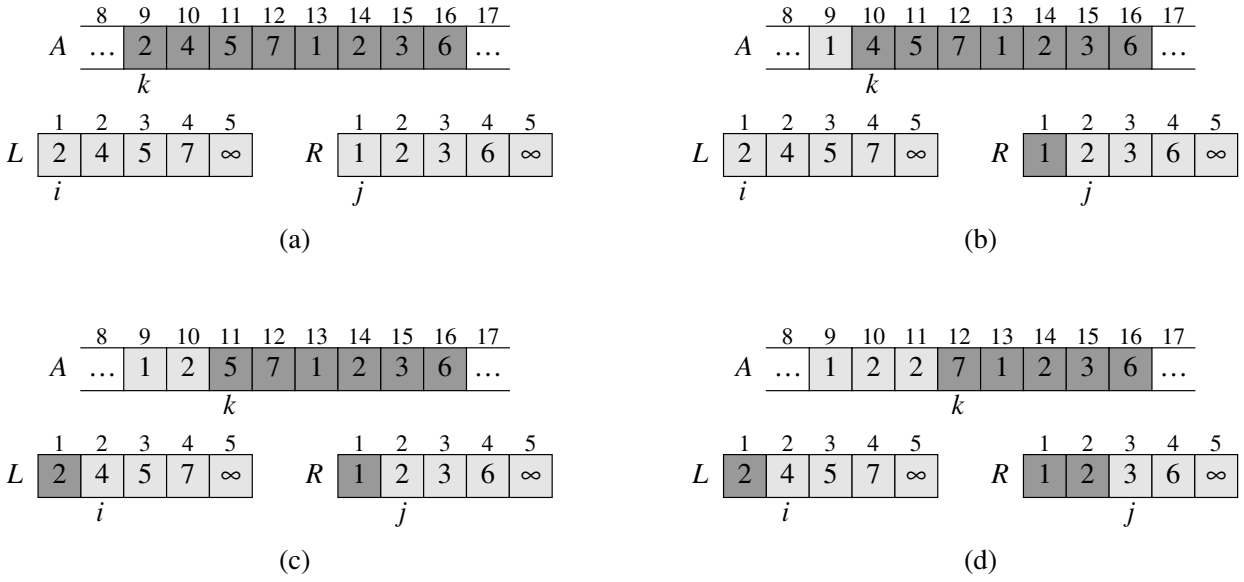


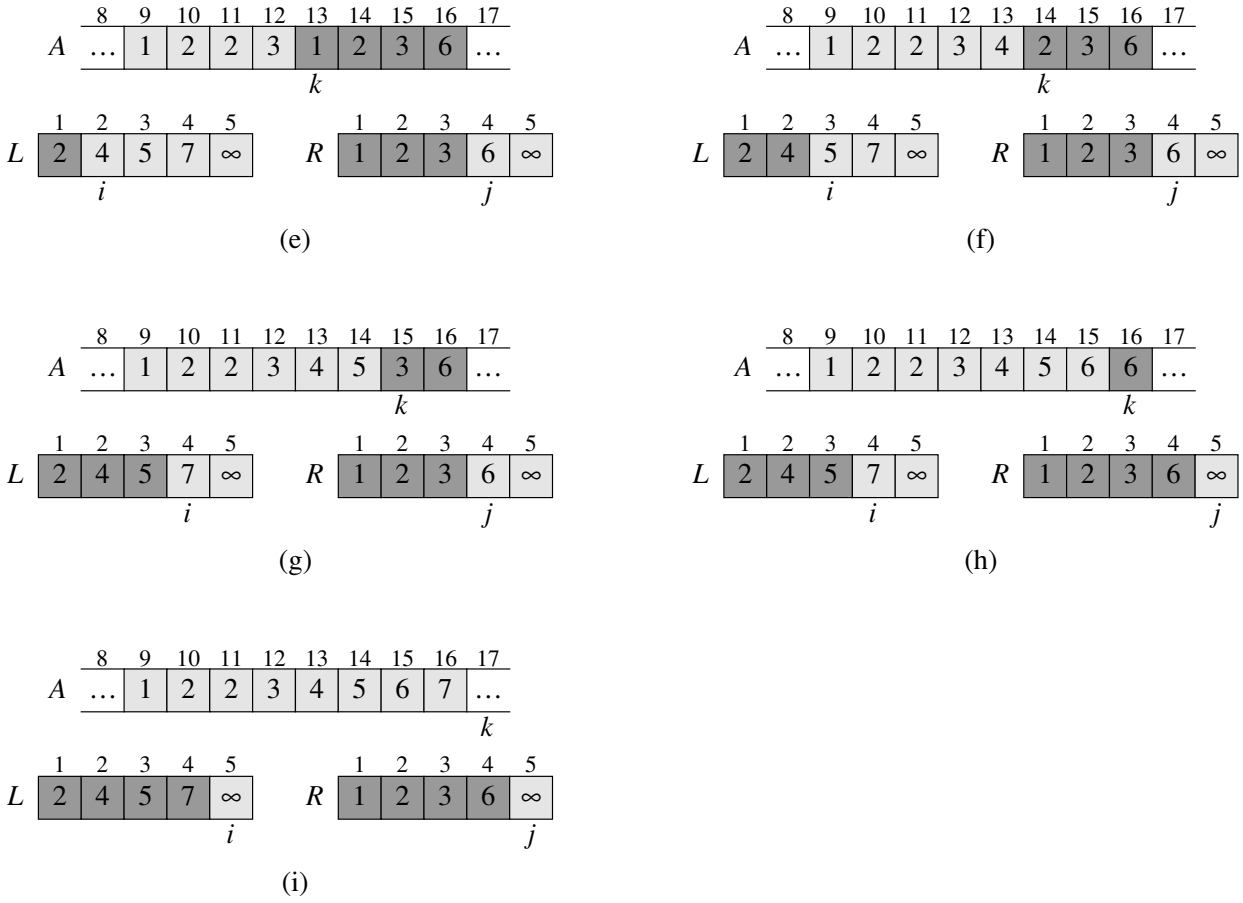

**Figure 2.3** The operation of lines 10–17 in the call MERGE($A, 9, 12, 16$), when the subarray $A[9\,..\,16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6\rangle$. After copying and inserting sentinels, the array $L$ contains $\langle 2, 4, 5, 7, \infty\rangle$, and the array $R$ contains $\langle 1, 2, 3, 6, \infty\rangle$. Lightly shaded positions in $A$ contain their final values, and lightly shaded positions in $L$ and $R$ contain values that have yet to be copied back into $A$. Taken together, the lightly shaded positions always comprise the values originally in $A[9\,..\,16]$, along with the two sentinels. Heavily shaded positions in $A$ contain values that will be copied over, and heavily shaded positions in $L$ and $R$ contain values that have already been copied back into $A$. **(a)–(h)** The arrays $A$, $L$, and $R$, and their respective indices $k$, $i$, and $j$ prior to each iteration of the loop of lines 12–17. **(i)** The arrays and indices at termination. At this point, the subarray in $A[9\,..\,16]$ is sorted, and the two sentinels in $L$ and $R$ are the only two elements in these arrays that have not been copied into $A$.

ray $A[p\,..\,q]$ into $L[1\,..\,n_1]$, and the **for** loop of lines 6–7 copies the subarray $A[q+1\,..\,r]$ into $R[1\,..\,n_2]$. Lines 8–9 put the sentinels at the ends of the arrays $L$ and $R$. Lines 10–17, illustrated in Figure 2.3, perform the $r - p + 1$ basic steps by maintaining the following loop invariant:

> At the start of each iteration of the **for** loop of lines 12–17, the subarray $A[p\,..\,k-1]$ contains the $k - p$ smallest elements of $L[1\,..\,n_1+1]$ and $R[1\,..\,n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into $A$.

We must show that this loop invariant holds prior to the first iteration of the **for** loop of lines 12–17, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

**Initialization:** Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p\,..\,k-1]$ is empty. This empty subarray contains the $k - p = 0$




smallest elements of $L$ and $R$, and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into $A$.

**Maintenance:** To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied back into $A$. Because $A[p \,..\, k-1]$ contains the $k - p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p \,..\, k]$ will contain the $k - p + 1$ smallest elements. Incrementing $k$ (in the **for** loop update) and $i$ (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

**Termination:** At termination, $k = r + 1$. By the loop invariant, the subarray $A[p \,..\, k-1]$, which is $A[p \,..\, r]$, contains the $k - p = r - p + 1$ smallest elements of $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$, in sorted order. The arrays $L$ and $R$ together contain $n_1 + n_2 + 2 = r - p + 3$ elements. All but the two largest have been copied back into $A$, and these two largest elements are the sentinels.

To see that the MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$, observe that each of lines 1–3 and 8–11 takes constant time, the **for** loops of

lines 4–7 take $\Theta(n_1 + n_2) = \Theta(n)$ time,[6] and there are $n$ iterations of the **for** loop of lines 12–17, each of which takes constant time.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT$(A, p, r)$ sorts the elements in the subarray $A[p..r]$. If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index $q$ that partitions $A[p..r]$ into two subarrays: $A[p..q]$, containing $\lceil n/2 \rceil$ elements, and $A[q+1..r]$, containing $\lfloor n/2 \rfloor$ elements.[7]

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

To sort the entire sequence $A = \langle A[1], A[2], \ldots, A[n] \rangle$, we make the initial call MERGE-SORT$(A, 1, length[A])$, where once again $length[A] = n$. Figure 2.4 illustrates the operation of the procedure bottom-up when $n$ is a power of 2. The algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length $n/2$ are merged to form the final sorted sequence of length $n$.

### 2.3.2 Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call to itself, its running time can often be described by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size $n$ in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

A recurrence for the running time of a divide-and-conquer algorithm is based on the three steps of the basic paradigm. As before, we let $T(n)$ be the running time on a problem of size $n$. If the problem size is small enough, say $n \leq c$

[6] We shall see in Chapter 3 how to formally interpret equations containing $\Theta$-notation.

[7] The expression $\lceil x \rceil$ denotes the least integer greater than or equal to $x$, and $\lfloor x \rfloor$ denotes the greatest integer less than or equal to $x$. These notations are defined in Chapter 3. The easiest way to verify that setting $q$ to $\lfloor (p+r)/2 \rfloor$ yields subarrays $A[p..q]$ and $A[q+1..r]$ of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively, is to examine the four cases that arise depending on whether each of $p$ and $r$ is odd or even.

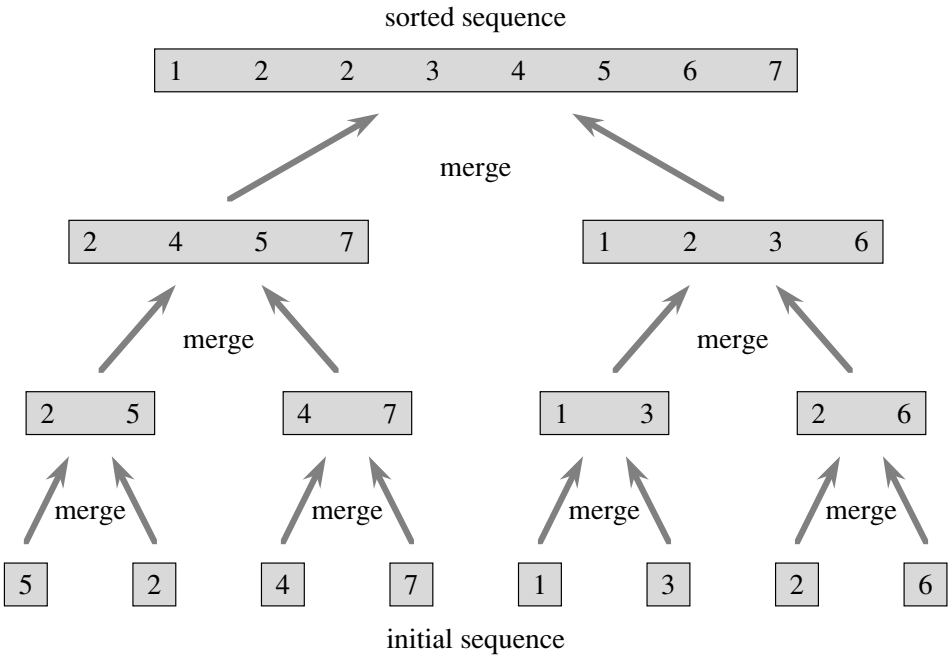



**Figure 2.4** The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

for some constant $c$, the straightforward solution takes constant time, which we write as $\Theta(1)$. Suppose that our division of the problem yields $a$ subproblems, each of which is $1/b$ the size of the original. (For merge sort, both $a$ and $b$ are 2, but we shall see many divide-and-conquer algorithms in which $a \neq b$.) If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \text{ ,} \\ aT(n/b) + D(n) + C(n) & \text{otherwise .} \end{cases}$$

In Chapter 4, we shall see how to solve common recurrences of this form.

### Analysis of merge sort

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of 2. Each divide step then yields two subsequences of size exactly $n/2$. In Chapter 4, we shall see that this assumption does not affect the order of growth of the solution to the recurrence.

We reason as follows to set up the recurrence for $T(n)$, the worst-case running time of merge sort on $n$ numbers. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

**Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

**Conquer:** We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

**Combine:** We have already noted that the MERGE procedure on an $n$-element subarray takes time $\Theta(n)$, so $C(n) = \Theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of $n$, that is, $\Theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases} \tag{2.1}$$

In Chapter 4, we shall see the "master theorem," which we can use to show that $T(n)$ is $\Theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$. Because the logarithm function grows more slowly than any linear function, for large enough inputs, merge sort, with its $\Theta(n \lg n)$ running time, outperforms insertion sort, whose running time is $\Theta(n^2)$, in the worst case.

We do not need the master theorem to intuitively understand why the solution to the recurrence (2.1) is $T(n) = \Theta(n \lg n)$. Let us rewrite recurrence (2.1) as

$$T(n) = \begin{cases} c & \text{if } n = 1 , \\ 2T(n/2) + cn & \text{if } n > 1 , \end{cases} \tag{2.2}$$

where the constant $c$ represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps.[8]

Figure 2.5 shows how we can solve the recurrence (2.2). For convenience, we assume that $n$ is an exact power of 2. Part (a) of the figure shows $T(n)$, which in part (b) has been expanded into an equivalent tree representing the recurrence. The $cn$ term is the root (the cost at the top level of recursion), and the two subtrees

[8] It is unlikely that the same constant exactly represents both the time to solve problems of size 1 and the time per array element of the divide and combine steps. We can get around this problem by letting $c$ be the larger of these times and understanding that our recurrence gives an upper bound on the running time, or by letting $c$ be the lesser of these times and understanding that our recurrence gives a lower bound on the running time. Both bounds will be on the order of $n \lg n$ and, taken together, give a $\Theta(n \lg n)$ running time.

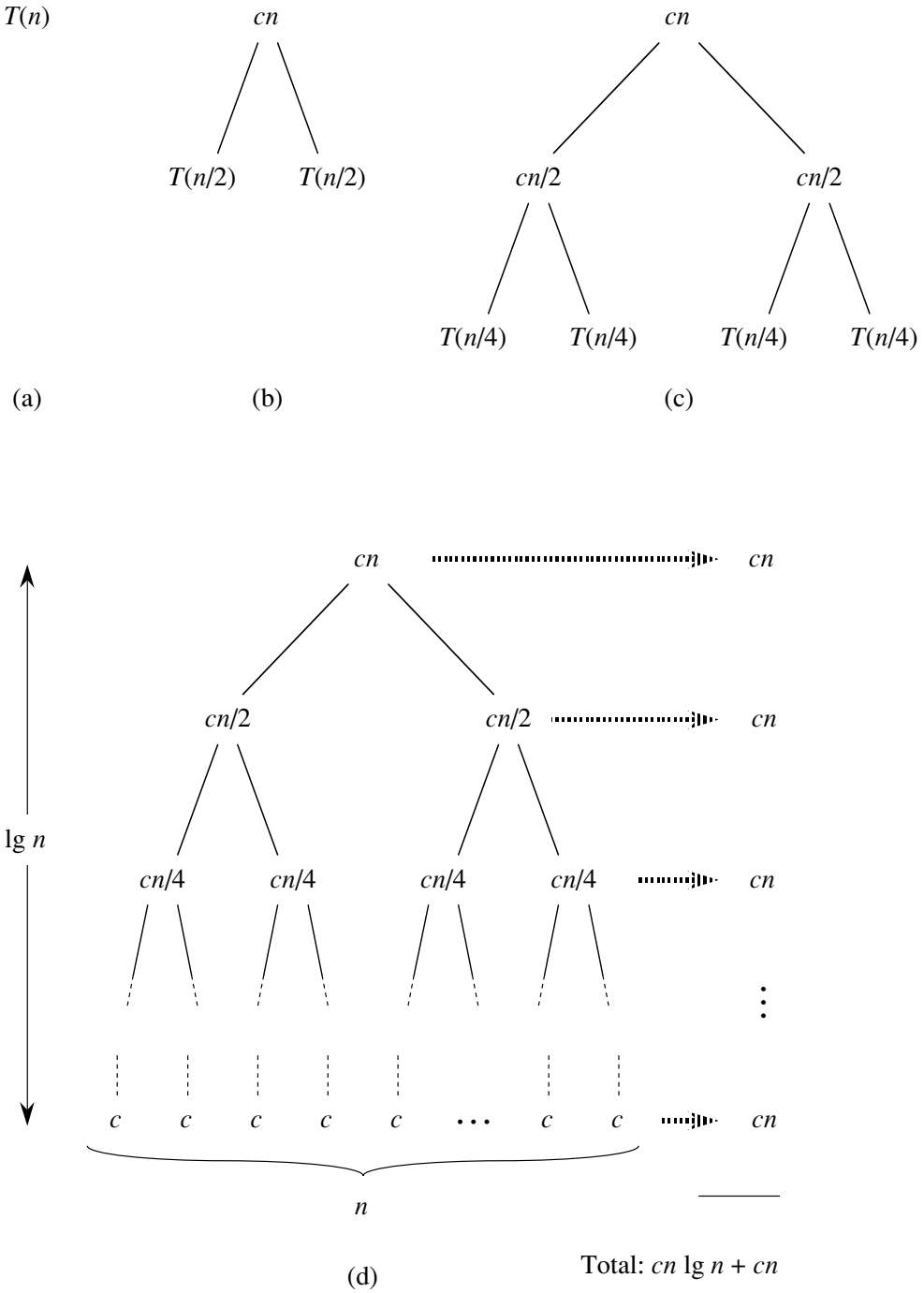



**Figure 2.5** The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of $cn$. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

of the root are the two smaller recurrences $T(n/2)$. Part (c) shows this process carried one step further by expanding $T(n/2)$. The cost for each of the two subnodes at the second level of recursion is $cn/2$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1, each with a cost of $c$. Part (d) shows the resulting tree.

Next, we add the costs across each level of the tree. The top level has total cost $cn$, the next level down has total cost $c(n/2) + c(n/2) = cn$, the level after that has total cost $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$, and so on. In general, the level $i$ below the top has $2^i$ nodes, each contributing a cost of $c(n/2^i)$, so that the $i$th level below the top has total cost $2^i\, c(n/2^i) = cn$. At the bottom level, there are $n$ nodes, each contributing a cost of $c$, for a total cost of $cn$.

The total number of levels of the "recursion tree" in Figure 2.5 is $\lg n + 1$. This fact is easily seen by an informal inductive argument. The base case occurs when $n = 1$, in which case there is only one level. Since $\lg 1 = 0$, we have that $\lg n + 1$ gives the correct number of levels. Now assume as an inductive hypothesis that the number of levels of a recursion tree for $2^i$ nodes is $\lg 2^i + 1 = i + 1$ (since for any value of $i$, we have that $\lg 2^i = i$). Because we are assuming that the original input size is a power of 2, the next input size to consider is $2^{i+1}$. A tree with $2^{i+1}$ nodes has one more level than a tree of $2^i$ nodes, and so the total number of levels is $(i + 1) + 1 = \lg 2^{i+1} + 1$.

To compute the total cost represented by the recurrence (2.2), we simply add up the costs of all the levels. There are $\lg n + 1$ levels, each costing $cn$, for a total cost of $cn(\lg n + 1) = cn \lg n + cn$. Ignoring the low-order term and the constant $c$ gives the desired result of $\Theta(n \lg n)$.

# 4 Recurrences

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

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

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

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

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

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

### Technicalities

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

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

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

$$T(n) = 2T(n/2) + \Theta(n) , \tag{4.3}$$

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

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

## 4.1 The substitution method

The substitution method for solving recurrences entails two steps:

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

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

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

$$T(n) = 2T(\lfloor n/2 \rfloor) + n , \tag{4.4}$$

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

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

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

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

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

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

### Making a good guess

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

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

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

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

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

### Subtleties

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

Consider the recurrence

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

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

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

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

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

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

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

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

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

### Avoiding pitfalls

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

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

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

### Changing variables

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

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

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

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

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

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

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

## 4.2 The recursion-tree method

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

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

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

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

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

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

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

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

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

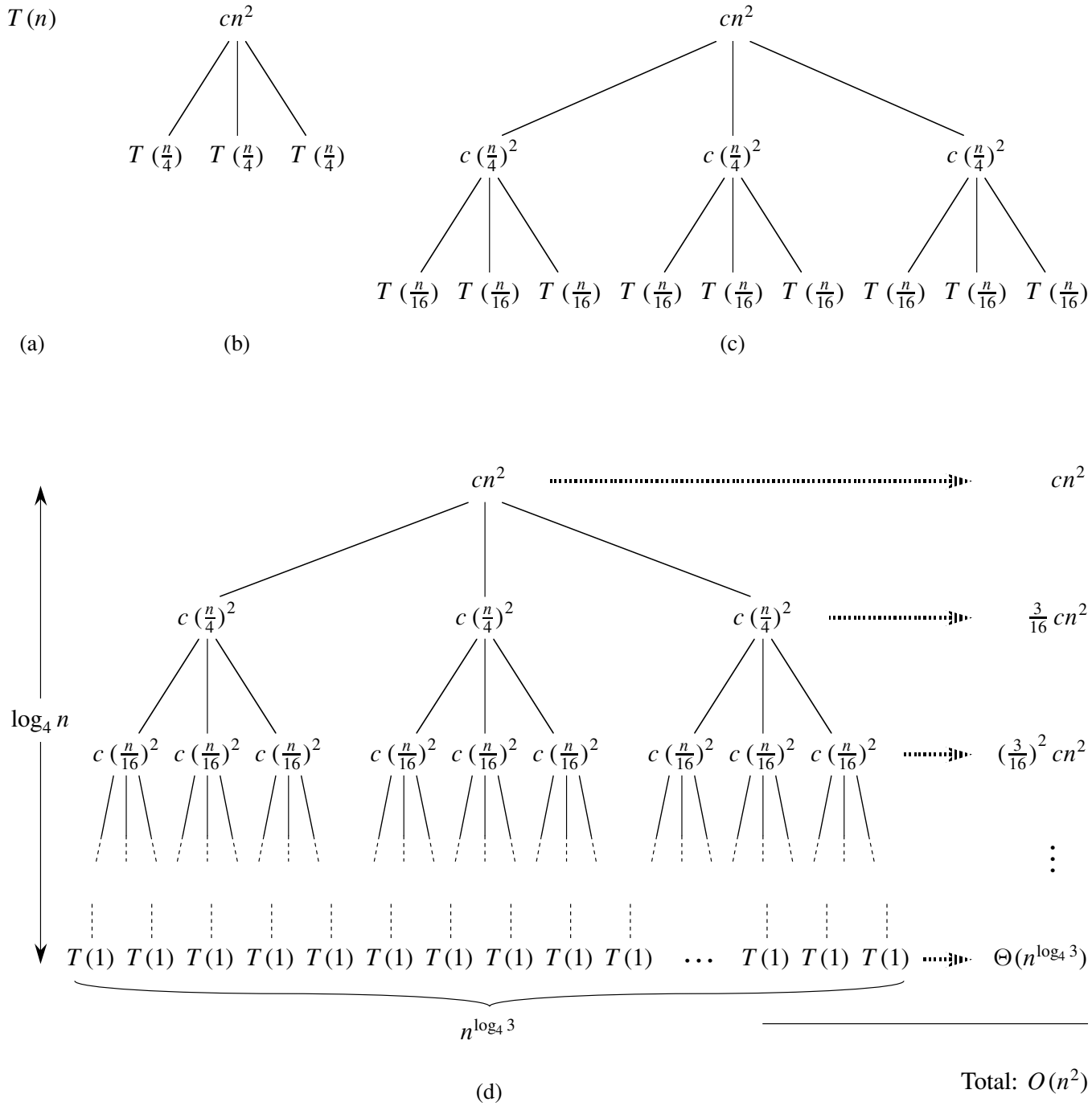



**Figure 4.1** The construction of a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

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

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

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

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

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

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

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

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

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

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

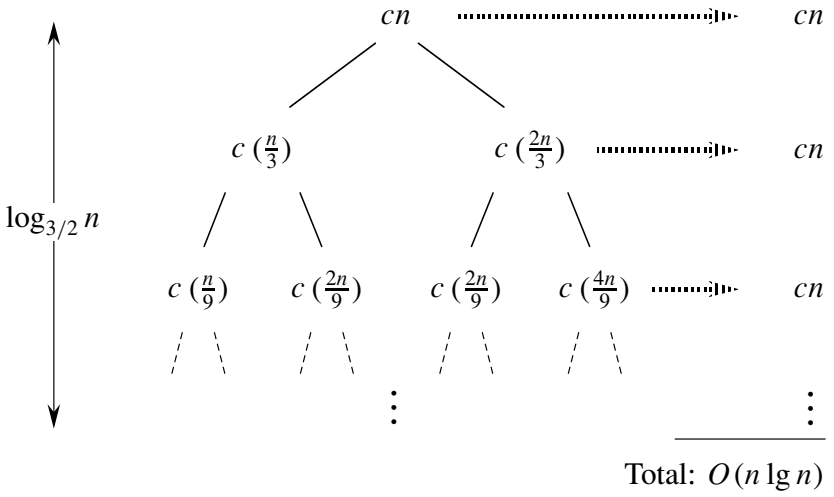



**Figure 4.2** A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

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

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

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

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

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

## 4.3 The master method

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

$$T(n) = aT(n/b) + f(n) , \tag{4.5}$$

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

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

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

### The master theorem

The master method depends on the following theorem.

***Theorem 4.1 (Master theorem)***
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

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

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

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

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

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

### Using the master method

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

As a first example, consider

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

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

Now consider

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

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

For the recurrence

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

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

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

The master method does not apply to the recurrence

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

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

## ⋆ 4.4 Proof of the master theorem

This section contains a proof of the master theorem (Theorem 4.1). The proof need not be understood in order to apply the theorem.

The proof is in two parts. The first part analyzes the "master" recurrence (4.5), under the simplifying assumption that $T(n)$ is defined only on exact powers of $b > 1$, that is, for $n = 1, b, b^2, \ldots$. This part gives all the intuition needed to understand why the master theorem is true. The second part shows how the analysis can be extended to all positive integers $n$ and is merely mathematical technique applied to the problem of handling floors and ceilings.

In this section, we shall sometimes abuse our asymptotic notation slightly by using it to describe the behavior of functions that are defined only over exact powers of $b$. Recall that the definitions of asymptotic notations require that bounds be proved for all sufficiently large numbers, not just those that are powers of $b$. Since we could make new asymptotic notations that apply to the set $\{b^i : i = 0, 1, \ldots\}$, instead of the nonnegative integers, this abuse is minor.

Nevertheless, we must always be on guard when we are using asymptotic notation over a limited domain so that we do not draw improper conclusions. For example, proving that $T(n) = O(n)$ when $n$ is an exact power of 2 does not guarantee that $T(n) = O(n)$. The function $T(n)$ could be defined as

$$T(n) = \begin{cases} n & \text{if } n = 1, 2, 4, 8, \ldots, \\ n^2 & \text{otherwise}, \end{cases}$$

in which case the best upper bound that can be proved is $T(n) = O(n^2)$. Because of this sort of drastic consequence, we shall never use asymptotic notation over a limited domain without making it absolutely clear from the context that we are doing so.

### 4.4.1 The proof for exact powers

The first part of the proof of the master theorem analyzes the recurrence (4.5)

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

for the master method, under the assumption that $n$ is an exact power of $b > 1$, where $b$ need not be an integer. The analysis is broken into three lemmas. The first reduces the problem of solving the master recurrence to the problem of evaluating an expression that contains a summation. The second determines bounds on this summation. The third lemma puts the first two together to prove a version of the master theorem for the case in which $n$ is an exact power of $b$.

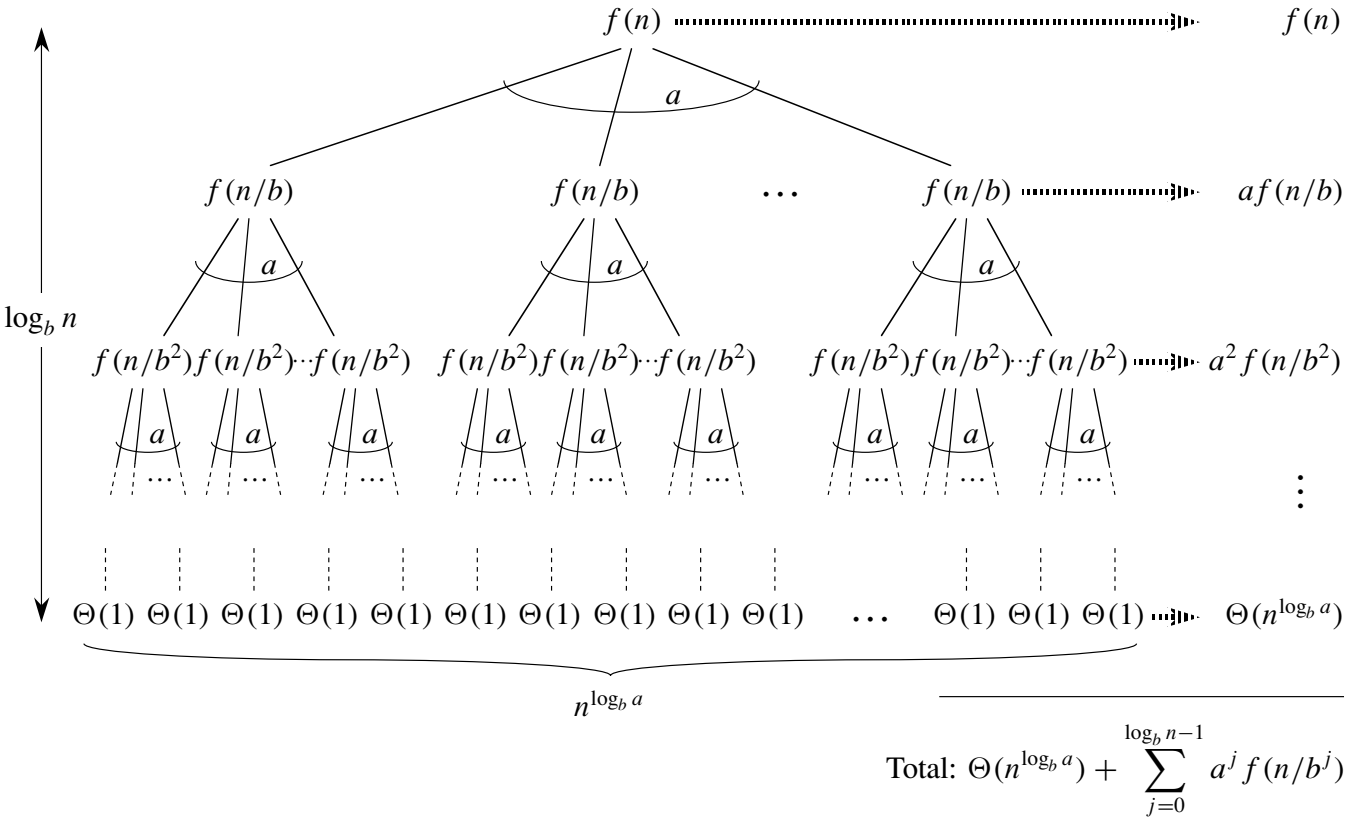



**Figure 4.3** The recursion tree generated by $T(n) = aT(n/b) + f(n)$. The tree is a complete $a$-ary tree with $n^{\log_b a}$ leaves and height $\log_b n$. The cost of each level is shown at the right, and their sum is given in equation (4.6).

***Lemma 4.2***

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

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

where $i$ is a positive integer. Then

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \ . \tag{4.6}$$

***Proof*** We use the recursion tree in Figure 4.3. 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 with cost $f(n/b^2)$, and thus there are $a^2$ nodes that are distance 2 from the root. In general, there are $a^j$ nodes

that are distance $j$ from the root, and each has cost $f(n/b^j)$. The cost of each leaf is $T(1) = \Theta(1)$, and each leaf is at depth $\log_b n$, since $n/b^{\log_b n} = 1$. There are $a^{\log_b n} = n^{\log_b a}$ leaves in the tree.

We can obtain equation (4.6) by summing the costs of each level of the tree, as shown in the figure. The cost for a level $j$ of internal nodes is $a^j f(n/b^j)$, and so the total of all internal node levels is

$$\sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \ .$$

In the underlying divide-and-conquer algorithm, this sum represents the costs of dividing problems into subproblems and then recombining the subproblems. The cost of all the leaves, which is the cost of doing all $n^{\log_b a}$ subproblems of size 1, is $\Theta(n^{\log_b a})$. ■

In terms of the recursion tree, the three cases of the master theorem correspond to cases in which the total cost of the tree is (1) dominated by the costs in the leaves, (2) evenly distributed across the levels of the tree, or (3) dominated by the cost of the root.

The summation in equation (4.6) 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 $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a nonnegative function defined on exact powers of $b$. A function $g(n)$ defined over exact powers of $b$ by

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \tag{4.7}$$

can then be bounded asymptotically for exact powers of $b$ as follows.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $g(n) = O(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $g(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $af(n/b) \leq cf(n)$ for some constant $c < 1$ and for all $n \geq b$, 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 (4.7) yields

$$g(n) = O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right) . \tag{4.8}$$

We bound the summation within the $O$-notation by factoring out terms and simplifying, which leaves an increasing geometric series:

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} &= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} (b^\epsilon)^j \\
&= n^{\log_b a-\epsilon} \left(\frac{b^{\epsilon \log_b n}-1}{b^\epsilon-1}\right) \\
&= n^{\log_b a-\epsilon} \left(\frac{n^\epsilon-1}{b^\epsilon-1}\right) .
\end{aligned}$$

Since $b$ and $\epsilon$ are constants, we can rewrite the last expression as $n^{\log_b a-\epsilon} O(n^\epsilon) = O(n^{\log_b a})$. Substituting this expression for the summation in equation (4.8) yields

$$g(n) = O(n^{\log_b a}) ,$$

and case 1 is proved.

Under the assumption that $f(n) = \Theta(n^{\log_b a})$ for case 2, we have that $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$. Substituting into equation (4.7) yields

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right) . \tag{4.9}$$

We bound the summation within the $\Theta$ as in case 1, but this time we do not obtain a geometric series. Instead, we discover that every term of the summation is the same:

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1 \\
&= n^{\log_b a} \log_b n .
\end{aligned}$$

Substituting this expression for the summation in equation (4.9) yields

$$\begin{aligned}
g(n) &= \Theta(n^{\log_b a} \log_b n) \\
&= \Theta(n^{\log_b a} \lg n) ,
\end{aligned}$$

and case 2 is proved.

Case 3 is proved similarly. Since $f(n)$ appears in the definition (4.7) of $g(n)$ and all terms of $g(n)$ are nonnegative, we can conclude that $g(n) = \Omega(f(n))$ for exact powers of $b$. Under our assumption that $af(n/b) \le cf(n)$ for some constant $c < 1$ and all $n \ge b$, we have $f(n/b) \le (c/a)f(n)$. Iterating $j$ times, we have $f(n/b^j) \le (c/a)^j f(n)$ or, equivalently, $a^j f(n/b^j) \le c^j f(n)$. Substituting into equation (4.7) and simplifying yields a geometric series, but unlike the series in case 1, this one has decreasing terms:

$$\begin{aligned} g(n) &= \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \\ &\le \sum_{j=0}^{\log_b n - 1} c^j f(n) \\ &\le f(n) \sum_{j=0}^{\infty} c^j \\ &= f(n) \left( \frac{1}{1-c} \right) \\ &= O(f(n)) , \end{aligned}$$

since $c$ is constant. Thus, we can conclude that $g(n) = \Theta(f(n))$ for exact powers of $b$. Case 3 is proved, which completes the proof of the lemma. ■

We can now prove a version of the master theorem for the case in which $n$ is an exact power of $b$.

***Lemma 4.4***

Let $a \ge 1$ and $b > 1$ be constants, and let $f(n)$ be a nonnegative function defined on exact powers of $b$. Define $T(n)$ on exact powers of $b$ by the recurrence

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

where $i$ is a positive integer. Then $T(n)$ can be bounded asymptotically for exact powers of $b$ as follows.

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

***Proof*** We use the bounds in Lemma 4.3 to evaluate the summation (4.6) from Lemma 4.2. For case 1, we have

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}) , \end{aligned}$$

and for case 2,

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) \\ &= \Theta(n^{\log_b a} \lg n) . \end{aligned}$$

For case 3,

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(f(n)) \\ &= \Theta(f(n)) , \end{aligned}$$

because $f(n) = \Omega(n^{\log_b a+\epsilon})$. ■

### 4.4.2 Floors and ceilings

To complete the proof of the master theorem, we must now extend our analysis to the situation in which floors and ceilings are used in the master recurrence, so that the recurrence is defined for all integers, not just exact powers of $b$. Obtaining a lower bound on

$$T(n) = aT(\lceil n/b \rceil) + f(n) \tag{4.10}$$

and an upper bound on

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \tag{4.11}$$

is routine, since the bound $\lceil n/b \rceil \geq n/b$ can be pushed through in the first case to yield the desired result, and the bound $\lfloor n/b \rfloor \leq n/b$ can be pushed through in the second case. Lower bounding the recurrence (4.11) requires much the same technique as upper bounding the recurrence (4.10), so we shall present only this latter bound.

We modify the recursion tree of Figure 4.3 to produce the recursion tree in Figure 4.4. As we go down in the recursion tree, we obtain a sequence of recursive invocations on the arguments

$$\begin{aligned} &n , \\ &\lceil n/b \rceil , \\ &\lceil \lceil n/b \rceil /b \rceil , \\ &\lceil \lceil \lceil n/b \rceil /b \rceil /b \rceil , \\ &\qquad \vdots \end{aligned}$$

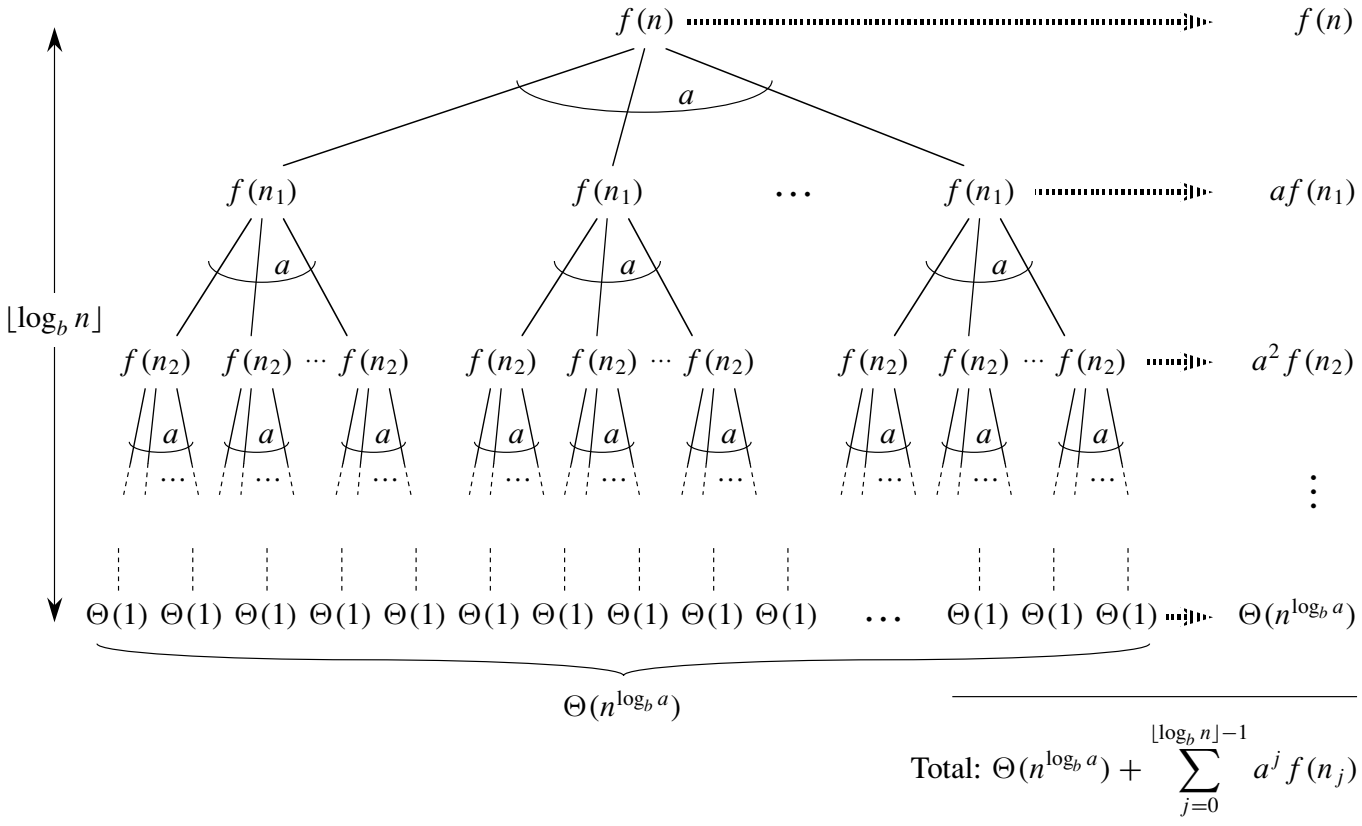



**Figure 4.4** The recursion tree generated by $T(n) = aT(\lceil n/b \rceil) + f(n)$. The recursive argument $n_j$ is given by equation (4.12).

Let us denote the $j$th element in the sequence by $n_j$, where

$$n_j = \begin{cases} n & \text{if } j = 0\,, \\ \lceil n_{j-1}/b \rceil & \text{if } j > 0\,. \end{cases} \tag{4.12}$$

Our first goal is to determine the depth $k$ such that $n_k$ is a constant. Using the inequality $\lceil x \rceil \le x + 1$, we obtain

$$\begin{aligned} n_0 &\le n\,, \\ n_1 &\le \frac{n}{b} + 1\,, \\ n_2 &\le \frac{n}{b^2} + \frac{1}{b} + 1\,, \\ n_3 &\le \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1\,, \\ &\vdots \end{aligned}$$

In general,

$$\begin{aligned} n_j &\le \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} \\ &< \frac{n}{b^j} + \sum_{i=0}^{\infty} \frac{1}{b^i} \\ &= \frac{n}{b^j} + \frac{b}{b-1} \, . \end{aligned}$$

Letting $j = \lfloor \log_b n \rfloor$, we obtain

$$\begin{aligned} n_{\lfloor \log_b n \rfloor} &< \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-1} \\ &\le \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} \\ &= \frac{n}{n/b} + \frac{b}{b-1} \\ &= b + \frac{b}{b-1} \\ &= O(1) \, , \end{aligned}$$

and thus we see that at depth $\lfloor \log_b n \rfloor$, the problem size is at most a constant.

From Figure 4.4, we see that

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \, , \tag{4.13}$$

which is much the same as equation (4.6), except that $n$ is an arbitrary integer and not restricted to be an exact power of $b$.

We can now evaluate the summation

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \tag{4.14}$$

from (4.13) in a manner analogous to the proof of Lemma 4.3. Beginning with case 3, if $af(\lceil n/b \rceil) \le cf(n)$ for $n > b + b/(b-1)$, where $c < 1$ is a constant, then it follows that $a^j f(n_j) \le c^j f(n)$. Therefore, the sum in equation (4.14) can be evaluated just as in Lemma 4.3. For case 2, we have $f(n) = \Theta(n^{\log_b a})$. If we can show that $f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$, then the proof for case 2 of Lemma 4.3 will go through. Observe that $j \le \lfloor \log_b n \rfloor$ implies $b^j/n \le 1$. The bound $f(n) = O(n^{\log_b a})$ implies that there exists a constant $c > 0$ such that for all

sufficiently large $n_j$,

$$\begin{aligned} f(n_j) &\leq c\left(\frac{n}{b^j} + \frac{b}{b-1}\right)^{\log_b a} \\ &= c\left(\frac{n}{b^j}\left(1 + \frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \\ &= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \left(\frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \\ &\leq c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b}{b-1}\right)^{\log_b a} \\ &= O\left(\frac{n^{\log_b a}}{a^j}\right), \end{aligned}$$

since $c(1 + b/(b-1))^{\log_b a}$ is a constant. Thus, case 2 is proved. The proof of case 1 is almost identical. The key is to prove the bound $f(n_j) = O(n^{\log_b a - \epsilon})$, which is similar to the corresponding proof of case 2, though the algebra is more intricate.

We have now proved the upper bounds in the master theorem for all integers $n$. The proof of the lower bounds is similar.

# 7 Quicksort

Quicksort is a sorting algorithm whose worst-case running time is $\Theta(n^2)$ on an input array of $n$ numbers. In spite of this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average: its expected running time is $\Theta(n \lg n)$, and the constant factors hidden in the $\Theta(n \lg n)$ notation are quite small. It also has the advantage of sorting in place (see page 16), and it works well even in virtual memory environments.

Section 7.1 describes the algorithm and an important subroutine used by quicksort for partitioning. Because the behavior of quicksort is complex, we start with an intuitive discussion of its performance in Section 7.2 and postpone its precise analysis to the end of the chapter. Section 7.3 presents a version of quicksort that uses random sampling. This algorithm has a good average-case running time, and no particular input elicits its worst-case behavior. The randomized algorithm is analyzed in Section 7.4, where it is shown to run in $\Theta(n^2)$ time in the worst case and in $O(n \lg n)$ time on average.

## 7.1 Description of quicksort

Quicksort, like merge sort, is based on the divide-and-conquer paradigm introduced in Section 2.3.1. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p \,..\, r]$.

**Divide:** Partition (rearrange) the array $A[p \,..\, r]$ into two (possibly empty) subarrays $A[p \,..\, q-1]$ and $A[q+1 \,..\, r]$ such that each element of $A[p \,..\, q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1 \,..\, r]$. Compute the index $q$ as part of this partitioning procedure.

**Conquer:** Sort the two subarrays $A[p \,..\, q-1]$ and $A[q+1 \,..\, r]$ by recursive calls to quicksort.

**Combine:** Since the subarrays are sorted in place, no work is needed to combine them: the entire array $A[p \,..\, r]$ is now sorted.

The following procedure implements quicksort.

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

To sort an entire array $A$, the initial call is QUICKSORT$(A, 1, length[A])$.

### Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p\,..\,r]$ in place.

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

Figure 7.1 shows the operation of PARTITION on an 8-element array. PARTITION always selects an element $x = A[r]$ as a ***pivot*** element around which to partition the subarray $A[p\,..\,r]$. As the procedure runs, the array is partitioned into four (possibly empty) regions. At the start of each iteration of the **for** loop in lines 3–6, each region satisfies certain properties, which we can state as a loop invariant:

> At the beginning of each iteration of the loop of lines 3–6, for any array index $k$,
>
> 1. If $p \le k \le i$, then $A[k] \le x$.
> 2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
> 3. If $k = r$, then $A[k] = x$.

Figure 7.2 summarizes this structure. The indices between $j$ and $r - 1$ are not covered by any of the three cases, and the values in these entries have no particular relationship to the pivot $x$.

We need to show that this loop invariant is true prior to the first iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

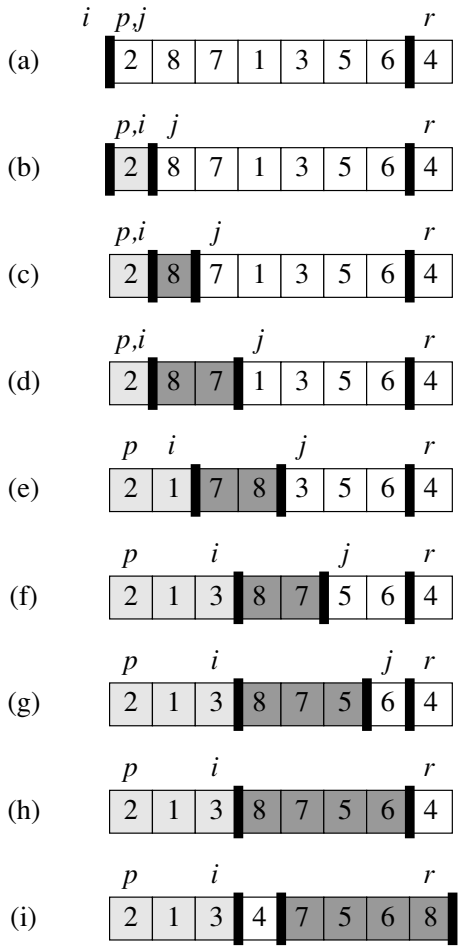



**Figure 7.1** The operation of PARTITION on a sample array. Lightly shaded array elements are all in the first partition with values no greater than $x$. Heavily shaded elements are in the second partition with values greater than $x$. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 7 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6 and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

**Initialization:** Prior to the first iteration of the loop, $i = p - 1$, and $j = p$. There are no values between $p$ and $i$, and no values between $i + 1$ and $j - 1$, so the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

**Maintenance:** As Figure 7.3 shows, there are two cases to consider, depending on the outcome of the test in line 4. Figure 7.3(a) shows what happens when

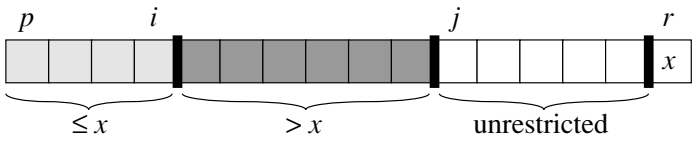



**Figure 7.2** The four regions maintained by the procedure PARTITION on a subarray $A[p\,..\,r]$. The values in $A[p\,..\,i]$ are all less than or equal to $x$, the values in $A[i+1\,..\,j-1]$ are all greater than $x$, and $A[r] = x$. The values in $A[j\,..\,r-1]$ can take on any values.

$A[j] > x$; the only action in the loop is to increment $j$. After $j$ is incremented, condition 2 holds for $A[j-1]$ and all other entries remain unchanged. Figure 7.3(b) shows what happens when $A[j] \le x$; $i$ is incremented, $A[i]$ and $A[j]$ are swapped, and then $j$ is incremented. Because of the swap, we now have that $A[i] \le x$, and condition 1 is satisfied. Similarly, we also have that $A[j-1] > x$, since the item that was swapped into $A[j-1]$ is, by the loop invariant, greater than $x$.

**Termination:** At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to $x$, those greater than $x$, and a singleton set containing $x$.

The final two lines of PARTITION move the pivot element into its place in the middle of the array by swapping it with the leftmost element that is greater than $x$. The output of PARTITION now satisfies the specifications given for the divide step.

The running time of PARTITION on the subarray $A[p\,..\,r]$ is $\Theta(n)$, where $n = r - p + 1$ (see Exercise 7.1-3).

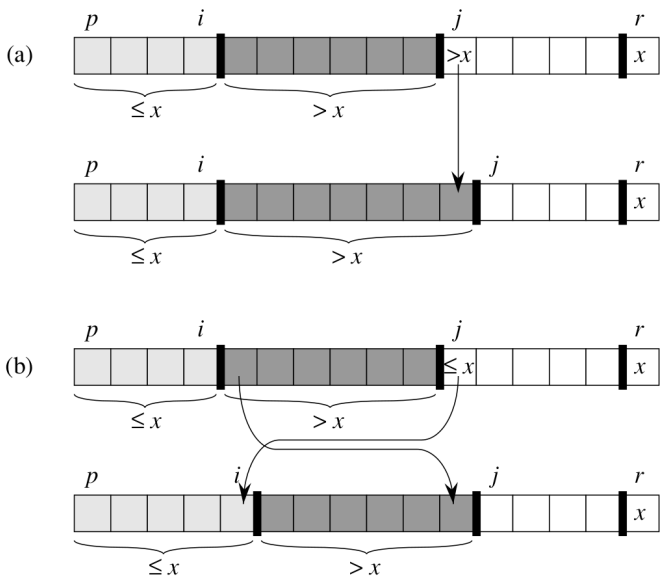



**Figure 7.3** The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment $j$, which maintains the loop invariant. **(b)** If $A[j] \leq x$, index $i$ is incremented, $A[i]$ and $A[j]$ are swapped, and then $j$ is incremented. Again, the loop invariant is maintained.

## 7.2 Performance of quicksort

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, and this in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort. In this section, we shall informally investigate how quicksort performs under the assumptions of balanced versus unbalanced partitioning.

### Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with $n - 1$ elements and one with 0 elements. (This claim is proved in Section 7.4.1.) Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs $\Theta(n)$ time. Since the recursive call

on an array of size 0 just returns, $T(0) = \Theta(1)$, and the recurrence for the running time is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \ . \end{aligned}$$

Intuitively, if we sum the costs incurred at each level of the recursion, we get an arithmetic series (equation (A.2)), which evaluates to $\Theta(n^2)$. Indeed, it is straightforward to use the substitution method to prove that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$. (See Exercise 7.2-1.)

Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$. Therefore the worst-case running time of quicksort is no better than that of insertion sort. Moreover, the $\Theta(n^2)$ running time occurs when the input array is already completely sorted—a common situation in which insertion sort runs in $O(n)$ time.

### Best-case partitioning

In the most even possible split, PARTITION produces two subproblems, each of size no more than $n/2$, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$. In this case, quicksort runs much faster. The recurrence for the running time is then

$$T(n) \leq 2T(n/2) + \Theta(n) \ ,$$

which by case 2 of the master theorem (Theorem 4.1) has the solution $T(n) = O(n \lg n)$. Thus, the equal balancing of the two sides of the partition at every level of the recursion produces an asymptotically faster algorithm.

### Balanced partitioning

The average-case running time of quicksort is much closer to the best case than to the worst case, as the analyses in Section 7.4 will show. The key to understanding why is to understand how the balance of the partitioning is reflected in the recurrence that describes the running time.

Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the recurrence

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

on the running time of quicksort, where we have explicitly included the constant $c$ hidden in the $\Theta(n)$ term. Figure 7.4 shows the recursion tree for this recurrence. Notice that every level of the tree has cost $cn$, until a boundary condition is reached at depth $\log_{10} n = \Theta(\lg n)$, and then the levels have cost at most $cn$. The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$. The total cost of quicksort is

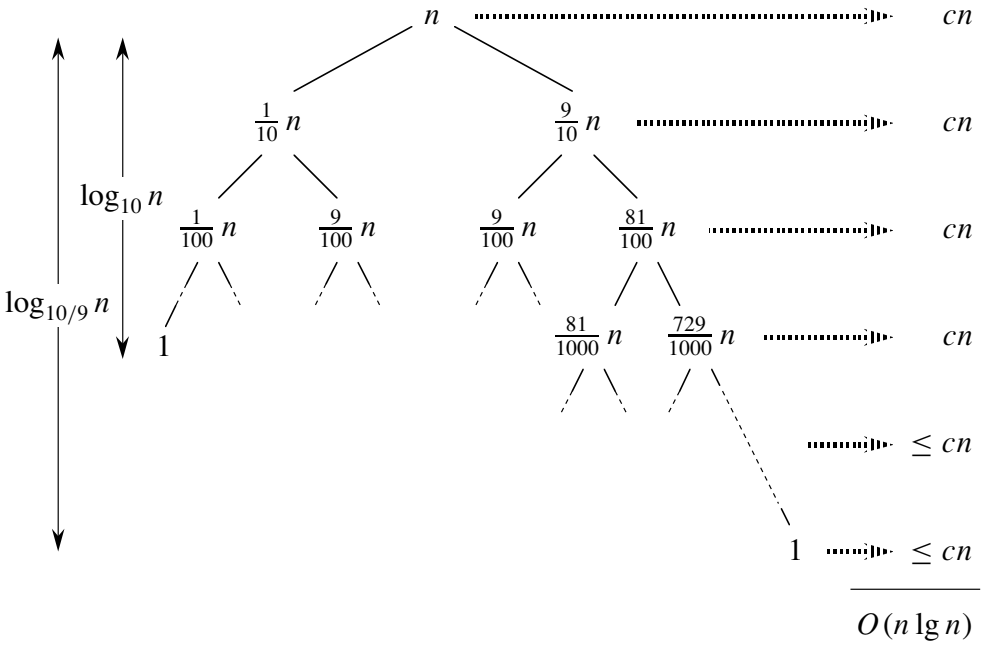



**Figure 7.4** A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant $c$ implicit in the $\Theta(n)$ term.

therefore $O(n \lg n)$. Thus, with a 9-to-1 proportional split at every level of recursion, which intuitively seems quite unbalanced, quicksort runs in $O(n \lg n)$ time—asymptotically the same as if the split were right down the middle. In fact, even a 99-to-1 split yields an $O(n \lg n)$ running time. The reason is that any split of *constant* proportionality yields a recursion tree of depth $\Theta(\lg n)$, where the cost at each level is $O(n)$. The running time is therefore $O(n \lg n)$ whenever the split has constant proportionality.

### Intuition for the average case

To develop a clear notion of the average case for quicksort, we must make an assumption about how frequently we expect to encounter the various inputs. The behavior of quicksort is determined by the relative ordering of the values in the array elements given as the input, and not by the particular values in the array. As in our probabilistic analysis of the hiring problem in Section 5.2, we will assume for now that all permutations of the input numbers are equally likely.

When we run quicksort on a random input array, it is unlikely that the partitioning always happens in the same way at every level, as our informal analysis has assumed. We expect that some of the splits will be reasonably well balanced and that some will be fairly unbalanced. For example, Exercise 7.2-6 asks you to show

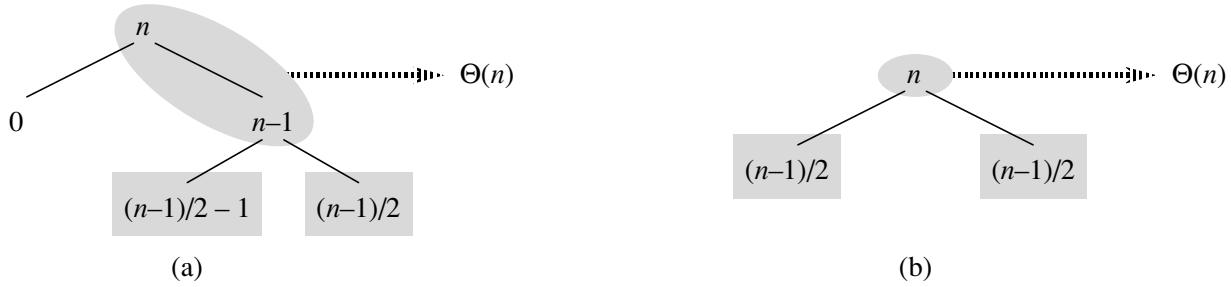



**Figure 7.5** **(a)** Two levels of a recursion tree for quicksort. The partitioning at the root costs $n$ and produces a "bad" split: two subarrays of sizes 0 and $n - 1$. The partitioning of the subarray of size $n - 1$ costs $n - 1$ and produces a "good" split: subarrays of size $(n - 1)/2 - 1$ and $(n - 1)/2$. **(b)** A single level of a recursion tree that is very well balanced. In both parts, the partitioning cost for the subproblems shown with elliptical shading is $\Theta(n)$. Yet the subproblems remaining to be solved in (a), shown with square shading, are no larger than the corresponding subproblems remaining to be solved in (b).

that about 80 percent of the time PARTITION produces a split that is more balanced than 9 to 1, and about 20 percent of the time it produces a split that is less balanced than 9 to 1.

In the average case, PARTITION produces a mix of "good" and "bad" splits. In a recursion tree for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree. Suppose for the sake of intuition, however, that the good and bad splits alternate levels in the tree, and that the good splits are best-case splits and the bad splits are worst-case splits. Figure 7.5(a) shows the splits at two consecutive levels in the recursion tree. At the root of the tree, the cost is $n$ for partitioning, and the subarrays produced have sizes $n - 1$ and 0: the worst case. At the next level, the subarray of size $n - 1$ is best-case partitioned into subarrays of size $(n - 1)/2 - 1$ and $(n - 1)/2$. Let's assume that the boundary-condition cost is 1 for the subarray of size 0.

The combination of the bad split followed by the good split produces three subarrays of sizes 0, $(n - 1)/2 - 1$, and $(n - 1)/2$ at a combined partitioning cost of $\Theta(n) + \Theta(n - 1) = \Theta(n)$. Certainly, this situation is no worse than that in Figure 7.5(b), namely a single level of partitioning that produces two subarrays of size $(n - 1)/2$, at a cost of $\Theta(n)$. Yet this latter situation is balanced! Intuitively, the $\Theta(n - 1)$ cost of the bad split can be absorbed into the $\Theta(n)$ cost of the good split, and the resulting split is good. Thus, the running time of quicksort, when levels alternate between good and bad splits, is like the running time for good splits alone: still $O(n \lg n)$, but with a slightly larger constant hidden by the $O$-notation. We shall give a rigorous analysis of the average case of a randomized version of quicksort in Section 7.4.2.

## 7.3 A randomized version of quicksort

In exploring the average-case behavior of quicksort, we have made an assumption that all permutations of the input numbers are equally likely. In an engineering situation, however, we cannot always expect it to hold. (See Exercise 7.2-4.) As we saw in Section 5.3, we can sometimes add randomization to an algorithm in order to obtain good average-case performance over all inputs. Many people regard the

resulting randomized version of quicksort as the sorting algorithm of choice for large enough inputs.

In Section 5.3, we randomized our algorithm by explicitly permuting the input. We could do so for quicksort also, but a different randomization technique, called ***random sampling***, yields a simpler analysis. Instead of always using $A[r]$ as the pivot, we will use a randomly chosen element from the subarray $A[p\,..\,r]$. We do so by exchanging element $A[r]$ with an element chosen at random from $A[p\,..\,r]$. This modification, in which we randomly sample the range $p, \ldots, r$, ensures that the pivot element $x = A[r]$ is equally likely to be any of the $r - p + 1$ elements in the subarray. Because the pivot element is randomly chosen, we expect the split of the input array to be reasonably well balanced on average.

The changes to PARTITION and QUICKSORT are small. In the new partition procedure, we simply implement the swap before actually partitioning:

```
RANDOMIZED-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  exchange A[r] ↔ A[i]
3  return PARTITION(A, p, r)
```

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2     then q ← RANDOMIZED-PARTITION(A, p, r)
3          RANDOMIZED-QUICKSORT(A, p, q − 1)
4          RANDOMIZED-QUICKSORT(A, q + 1, r)
```

We analyze this algorithm in the next section.

## 7.4 Analysis of quicksort

Section 7.2 gave some intuition for the worst-case behavior of quicksort and for why we expect it to run quickly. In this section, we analyze the behavior of quicksort more rigorously. We begin with a worst-case analysis, which applies to either QUICKSORT or RANDOMIZED-QUICKSORT, and conclude with an average-case analysis of RANDOMIZED-QUICKSORT.

### 7.4.1 Worst-case analysis

We saw in Section 7.2 that a worst-case split at every level of recursion in quicksort produces a $\Theta(n^2)$ running time, which, intuitively, is the worst-case running time of the algorithm. We now prove this assertion.

Using the substitution method (see Section 4.1), we can show that the running time of quicksort is $O(n^2)$. Let $T(n)$ be the worst-case time for the procedure QUICKSORT on an input of size $n$. We have the recurrence

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) , \tag{7.1}$$

where the parameter $q$ ranges from 0 to $n-1$ because the procedure PARTITION produces two subproblems with total size $n-1$. We guess that $T(n) \le cn^2$ for some constant $c$. Substituting this guess into recurrence (7.1), we obtain

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) . \end{aligned}$$

The expression $q^2 + (n-q-1)^2$ achieves a maximum over the parameter's range $0 \le q \le n-1$ at either endpoint, as can be seen since the second derivative of the expression with respect to $q$ is positive (see Exercise 7.4-3). This observation gives us the bound $\max_{0 \le q \le n-1}(q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$. Continuing with our bounding of $T(n)$, we obtain

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

since we can pick the constant $c$ large enough so that the $c(2n-1)$ term dominates the $\Theta(n)$ term. Thus, $T(n) = O(n^2)$. We saw in Section 7.2 a specific case in which quicksort takes $\Omega(n^2)$ time: when partitioning is unbalanced. Alternatively, Exercise 7.4-1 asks you to show that recurrence (7.1) has a solution of $T(n) = \Omega(n^2)$. Thus, the (worst-case) running time of quicksort is $\Theta(n^2)$.

### 7.4.2 Expected running time

We have already given an intuitive argument why the average-case running time of RANDOMIZED-QUICKSORT is $O(n \lg n)$: if, in each level of recursion, the split induced by RANDOMIZED-PARTITION puts any constant fraction of the elements on one side of the partition, then the recursion tree has depth $\Theta(\lg n)$, and $O(n)$ work is performed at each level. Even if we add new levels with the most unbalanced split possible between these levels, the total time remains $O(n \lg n)$. We can analyze the expected running time of RANDOMIZED-QUICKSORT precisely by first understanding how the partitioning procedure operates and then using this understanding to derive an $O(n \lg n)$ bound on the expected running time. This upper bound on the expected running time, combined with the $\Theta(n \lg n)$ best-case bound we saw in Section 7.2, yields a $\Theta(n \lg n)$ expected running time.

#### Running time and comparisons

The running time of QUICKSORT is dominated by the time spent in the PARTITION procedure. Each time the PARTITION procedure is called, a pivot element is selected, and this element is never included in any future recursive calls to QUICKSORT and PARTITION. Thus, there can be at most $n$ calls to PARTITION over the entire execution of the quicksort algorithm. One call to PARTITION takes $O(1)$ time plus an amount of time that is proportional to the number of iterations of the **for** loop in lines 3–6. Each iteration of this **for** loop performs a comparison in line 4, comparing the pivot element to another element of the array $A$. Therefore, if we can count the total number of times that line 4 is executed, we can bound the total time spent in the **for** loop during the entire execution of QUICKSORT.

***Lemma 7.1***
Let $X$ be the number of comparisons performed in line 4 of PARTITION over the entire execution of QUICKSORT on an $n$-element array. Then the running time of QUICKSORT is $O(n + X)$.

***Proof*** By the discussion above, there are $n$ calls to PARTITION, each of which does a constant amount of work and then executes the **for** loop some number of times. Each iteration of the **for** loop executes line 4. ■

Our goal, therefore is to compute $X$, the total number of comparisons performed in all calls to PARTITION. We will not attempt to analyze how many comparisons are made in *each* call to PARTITION. Rather, we will derive an overall bound on the total number of comparisons. To do so, we must understand when the algorithm compares two elements of the array and when it does not. For ease of analysis, we rename the elements of the array $A$ as $z_1, z_2, \ldots, z_n$, with $z_i$ being the $i$th smallest

element. We also define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ to be the set of elements between $z_i$ and $z_j$, inclusive.

When does the algorithm compare $z_i$ and $z_j$? To answer this question, we first observe that each pair of elements is compared at most once. Why? Elements are compared only to the pivot element and, after a particular call of PARTITION finishes, the pivot element used in that call is never again compared to any other elements.

Our analysis uses indicator random variables (see Section 5.2). We define

$$X_{ij} = \mathrm{I}\{z_i \text{ is compared to } z_j\} \ ,$$

where we are considering whether the comparison takes place at any time during the execution of the algorithm, not just during one iteration or one call of PARTITION. Since each pair is compared at most once, we can easily characterize the total number of comparisons performed by the algorithm:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \ .$$

Taking expectations of both sides, and then using linearity of expectation and Lemma 5.1, we obtain

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\} \ . \end{aligned} \tag{7.2}$$

It remains to compute $\Pr\{z_i \text{ is compared to } z_j\}$.

It is useful to think about when two items are *not* compared. Consider an input to quicksort of the numbers 1 through 10 (in any order), and assume that the first pivot element is 7. Then the first call to PARTITION separates the numbers into two sets: $\{1, 2, 3, 4, 5, 6\}$ and $\{8, 9, 10\}$. In doing so, the pivot element 7 is compared to all other elements, but no number from the first set (e.g., 2) is or ever will be compared to any number from the second set (e.g., 9).

In general, once a pivot $x$ is chosen with $z_i < x < z_j$, we know that $z_i$ and $z_j$ cannot be compared at any subsequent time. If, on the other hand, $z_i$ is chosen as a pivot before any other item in $Z_{ij}$, then $z_i$ will be compared to each item in $Z_{ij}$, except for itself. Similarly, if $z_j$ is chosen as a pivot before any other item in $Z_{ij}$, then $z_j$ will be compared to each item in $Z_{ij}$, except for itself. In our example, the

values 7 and 9 are compared because 7 is the first item from $Z_{7,9}$ to be chosen as a pivot. In contrast, 2 and 9 will never be compared because the first pivot element chosen from $Z_{2,9}$ is 7. Thus, $z_i$ and $z_j$ are compared if and only if the first element to be chosen as a pivot from $Z_{ij}$ is either $z_i$ or $z_j$.

We now compute the probability that this event occurs. Prior to the point at which an element from $Z_{ij}$ has been chosen as a pivot, the whole set $Z_{ij}$ is together in the same partition. Therefore, any element of $Z_{ij}$ is equally likely to be the first one chosen as a pivot. Because the set $Z_{ij}$ has $j - i + 1$ elements, the probability that any given element is the first one chosen as a pivot is $1/(j - i + 1)$. Thus, we have

$$\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1} . \qquad (7.3)
\end{aligned}$$

The second line follows because the two events are mutually exclusive. Combining equations (7.2) and (7.3), we get that

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} .$$

We can evaluate this sum using a change of variables ($k = j - i$) and the bound on the harmonic series in equation (A.7):

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) . \qquad (7.4)
\end{aligned}$$

Thus we conclude that, using RANDOMIZED-PARTITION, the expected running time of quicksort is $O(n \lg n)$.

# 9 Medians and Order Statistics

The $i$th ***order statistic*** of a set of $n$ elements is the $i$th smallest element. For example, the ***minimum*** of a set of elements is the first order statistic ($i = 1$), and the ***maximum*** is the $n$th order statistic ($i = n$). A ***median***, informally, is the "halfway point" of the set. When $n$ is odd, the median is unique, occurring at $i = (n+1)/2$. When $n$ is even, there are two medians, occurring at $i = n/2$ and $i = n/2 + 1$. Thus, regardless of the parity of $n$, medians occur at $i = \lfloor (n+1)/2 \rfloor$ (the ***lower median***) and $i = \lceil (n+1)/2 \rceil$ (the ***upper median***). For simplicity in this text, however, we consistently use the phrase "the median" to refer to the lower median.

This chapter addresses the problem of selecting the $i$th order statistic from a set of $n$ distinct numbers. We assume for convenience that the set contains distinct numbers, although virtually everything that we do extends to the situation in which a set contains repeated values. The ***selection problem*** can be specified formally as follows:

**Input:** A set $A$ of $n$ (distinct) numbers and a number $i$, with $1 \leq i \leq n$.

**Output:** The element $x \in A$ that is larger than exactly $i - 1$ other elements of $A$.

The selection problem can be solved in $O(n \lg n)$ time, since we can sort the numbers using heapsort or merge sort and then simply index the $i$th element in the output array. There are faster algorithms, however.

In Section 9.1, we examine the problem of selecting the minimum and maximum of a set of elements. More interesting is the general selection problem, which is investigated in the subsequent two sections. Section 9.2 analyzes a practical algorithm that achieves an $O(n)$ bound on the running time in the average case. Section 9.3 contains an algorithm of more theoretical interest that achieves the $O(n)$ running time in the worst case.

## 9.1 Minimum and maximum

How many comparisons are necessary to determine the minimum of a set of $n$ elements? We can easily obtain an upper bound of $n - 1$ comparisons: examine each element of the set in turn and keep track of the smallest element seen so far. In the following procedure, we assume that the set resides in array $A$, where $length[A] = n$.

```
MINIMUM(A)
1  min ← A[1]
2  for i ← 2 to length[A]
3        do if min > A[i]
4              then min ← A[i]
5  return min
```

Finding the maximum can, of course, be accomplished with $n - 1$ comparisons as well.

Is this the best we can do? Yes, since we can obtain a lower bound of $n - 1$ comparisons for the problem of determining the minimum. Think of any algorithm that determines the minimum as a tournament among the elements. Each comparison is a match in the tournament in which the smaller of the two elements wins. The key observation is that every element except the winner must lose at least one match. Hence, $n - 1$ comparisons are necessary to determine the minimum, and the algorithm MINIMUM is optimal with respect to the number of comparisons performed.

### Simultaneous minimum and maximum

In some applications, we must find both the minimum and the maximum of a set of $n$ elements. For example, a graphics program may need to scale a set of $(x, y)$ data to fit onto a rectangular display screen or other graphical output device. To do so, the program must first determine the minimum and maximum of each coordinate.

It is not difficult to devise an algorithm that can find both the minimum and the maximum of $n$ elements using $\Theta(n)$ comparisons, which is asymptotically optimal. Simply find the minimum and maximum independently, using $n - 1$ comparisons for each, for a total of $2n - 2$ comparisons.

In fact, at most $3\lfloor n/2 \rfloor$ comparisons are sufficient to find both the minimum and the maximum. The strategy is to maintain the minimum and maximum elements seen thus far. Rather than processing each element of the input by comparing it against the current minimum and maximum, at a cost of 2 comparisons per element,

we process elements in pairs. We compare pairs of elements from the input first *with each other*, and then we compare the smaller to the current minimum and the larger to the current maximum, at a cost of 3 comparisons for every 2 elements.

Setting up initial values for the current minimum and maximum depends on whether $n$ is odd or even. If $n$ is odd, we set both the minimum and maximum to the value of the first element, and then we process the rest of the elements in pairs. If $n$ is even, we perform 1 comparison on the first 2 elements to determine the initial values of the minimum and maximum, and then process the rest of the elements in pairs as in the case for odd $n$.

Let us analyze the total number of comparisons. If $n$ is odd, then we perform $3\lfloor n/2 \rfloor$ comparisons. If $n$ is even, we perform 1 initial comparison followed by $3(n-2)/2$ comparisons, for a total of $3n/2 - 2$. Thus, in either case, the total number of comparisons is at most $3\lfloor n/2 \rfloor$.

## 9.2 Selection in expected linear time

The general selection problem appears more difficult than the simple problem of finding a minimum. Yet, surprisingly, the asymptotic running time for both problems is the same: $\Theta(n)$. In this section, we present a divide-and-conquer algorithm for the selection problem. The algorithm RANDOMIZED-SELECT is modeled after the quicksort algorithm of Chapter 7. As in quicksort, the idea is to partition the input array recursively. But unlike quicksort, which recursively processes both sides of the partition, RANDOMIZED-SELECT only works on one side of the partition. This difference shows up in the analysis: whereas quicksort has an expected running time of $\Theta(n \lg n)$, the expected time of RANDOMIZED-SELECT is $\Theta(n)$.

RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION introduced in Section 7.3. Thus, like RANDOMIZED-QUICKSORT, it is a randomized algorithm, since its behavior is determined in part by the output of a random-number

generator. The following code for RANDOMIZED-SELECT returns the $i$th smallest element of the array $A[p \,..\, r]$.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p = r
2     then return A[p]
3  q ← RANDOMIZED-PARTITION(A, p, r)
4  k ← q − p + 1
5  if i = k            ▷ the pivot value is the answer
6     then return A[q]
7  elseif i < k
8     then return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

After RANDOMIZED-PARTITION is executed in line 3 of the algorithm, the array $A[p \,..\, r]$ is partitioned into two (possibly empty) subarrays $A[p \,..\, q-1]$ and $A[q+1 \,..\, r]$ such that each element of $A[p \,..\, q-1]$ is less than or equal to $A[q]$, which in turn is less than each element of $A[q+1 \,..\, r]$. As in quicksort, we will refer to $A[q]$ as the ***pivot*** element. Line 4 of RANDOMIZED-SELECT computes the number $k$ of elements in the subarray $A[p \,..\, q]$, that is, the number of elements in the low side of the partition, plus one for the pivot element. Line 5 then checks whether $A[q]$ is the $i$th smallest element. If it is, then $A[q]$ is returned. Otherwise, the algorithm determines in which of the two subarrays $A[p \,..\, q-1]$ and $A[q+1 \,..\, r]$ the $i$th smallest element lies. If $i < k$, then the desired element lies on the low side of the partition, and it is recursively selected from the subarray in line 8. If $i > k$, however, then the desired element lies on the high side of the partition. Since we already know $k$ values that are smaller than the $i$th smallest element of $A[p \,..\, r]$—namely, the elements of $A[p \,..\, q]$—the desired element is the $(i-k)$th smallest element of $A[q+1 \,..\, r]$, which is found recursively in line 9. The code appears to allow recursive calls to subarrays with 0 elements, but Exercise 9.2-1 asks you to show that this situation cannot happen.

The worst-case running time for RANDOMIZED-SELECT is $\Theta(n^2)$, even to find the minimum, because we could be extremely unlucky and always partition around the largest remaining element, and partitioning takes $\Theta(n)$ time. The algorithm works well in the average case, though, and because it is randomized, no particular input elicits the worst-case behavior.

The time required by RANDOMIZED-SELECT on an input array $A[p \,..\, r]$ of $n$ elements is a random variable that we denote by $T(n)$, and we obtain an upper bound on $\mathrm{E}[T(n)]$ as follows. Procedure RANDOMIZED-PARTITION is equally likely to return any element as the pivot. Therefore, for each $k$ such that $1 \le k \le n$, the subarray $A[p \,..\, q]$ has $k$ elements (all less than or equal to the pivot) with

probability $1/n$. For $k = 1, 2, \ldots, n$, we define indicator random variables $X_k$ where

$$X_k = \mathrm{I}\{\text{the subarray } A[p\,..\,q] \text{ has exactly } k \text{ elements}\} ,$$

and so we have

$$\mathrm{E}[X_k] = 1/n . \tag{9.1}$$

When we call RANDOMIZED-SELECT and choose $A[q]$ as the pivot element, we do not know, a priori, if we will terminate immediately with the correct answer, recurse on the subarray $A[p\,..\,q-1]$, or recurse on the subarray $A[q+1\,..\,r]$. This decision depends on where the $i$th smallest element falls relative to $A[q]$. Assuming that $T(n)$ is monotonically increasing, we can bound the time needed for the recursive call by the time needed for the recursive call on the largest possible input. In other words, we assume, to obtain an upper bound, that the $i$th element is always on the side of the partition with the greater number of elements. For a given call of RANDOMIZED-SELECT, the indicator random variable $X_k$ has the value 1 for exactly one value of $k$, and it is 0 for all other $k$. When $X_k = 1$, the two subarrays on which we might recurse have sizes $k-1$ and $n-k$. Hence, we have the recurrence

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n) . \end{aligned}$$

Taking expected values, we have

$$\begin{aligned} &\mathrm{E}[T(n)] \\ &\le \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \\ &= \sum_{k=1}^{n} \mathrm{E}[X_k \cdot T(\max(k-1, n-k))] + O(n) && \text{(by linearity of expectation)} \\ &= \sum_{k=1}^{n} \mathrm{E}[X_k] \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) && \text{(by equation (C.23))} \\ &= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) && \text{(by equation (9.1))} . \end{aligned}$$

In order to apply equation (C.23), we rely on $X_k$ and $T(\max(k-1, n-k))$ being independent random variables. Exercise 9.2-2 asks you to justify this assertion.

Let us consider the expression $\max(k-1, n-k)$. We have

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil \text{ ,} \\ n-k & \text{if } k \leq \lceil n/2 \rceil \text{ .} \end{cases}$$

If $n$ is even, each term from $T(\lceil n/2 \rceil)$ up to $T(n-1)$ appears exactly twice in the summation, and if $n$ is odd, all these terms appear twice and $T(\lfloor n/2 \rfloor)$ appears once. Thus, we have

$$\mathrm{E}[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}[T(k)] + O(n) \text{ .}$$

We solve the recurrence by substitution. Assume that $T(n) \leq cn$ for some constant $c$ that satisfies the initial conditions of the recurrence. We assume that $T(n) = O(1)$ for $n$ less than some constant; we shall pick this constant later. We also pick a constant $a$ such that the function described by the $O(n)$ term above (which describes the non-recursive component of the running time of the algorithm) is bounded from above by $an$ for all $n > 0$. Using this inductive hypothesis, we have

$$\begin{aligned}
\mathrm{E}[T(n)] &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n} \left( \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an \\
&= \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) + an \\
&\leq \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + an \\
&= \frac{2c}{n} \left( \frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) + an \\
&= \frac{c}{n} \left( \frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an \\
&= c \left( \frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an \\
&\leq \frac{3cn}{4} + \frac{c}{2} + an \\
&= cn - \left( \frac{cn}{4} - \frac{c}{2} - an \right) \text{ .}
\end{aligned}$$

In order to complete the proof, we need to show that for sufficiently large $n$, this last expression is at most $cn$ or, equivalently, that $cn/4 - c/2 - an \geq 0$. If we add $c/2$ to both sides and factor out $n$, we get $n(c/4 - a) \geq c/2$. As long as we

choose the constant $c$ so that $c/4 - a > 0$, i.e., $c > 4a$, we can divide both sides by $c/4 - a$, giving

$$n \geq \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a} .$$

Thus, if we assume that $T(n) = O(1)$ for $n < 2c/(c-4a)$, we have $T(n) = O(n)$. We conclude that any order statistic, and in particular the median, can be determined on average in linear time.

## 9.3 Selection in worst-case linear time

We now examine a selection algorithm whose running time is $O(n)$ in the worst case. Like RANDOMIZED-SELECT, the algorithm SELECT finds the desired element by recursively partitioning the input array. The idea behind the algorithm, however, is to *guarantee* a good split when the array is partitioned. SELECT uses the deterministic partitioning algorithm PARTITION from quicksort (see Section 7.1), modified to take the element to partition around as an input parameter.

The SELECT algorithm determines the $i$th smallest of an input array of $n > 1$ elements by executing the following steps. (If $n = 1$, then SELECT merely returns its only input value as the $i$th smallest.)

1. Divide the $n$ elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining $n$ mod 5 elements.

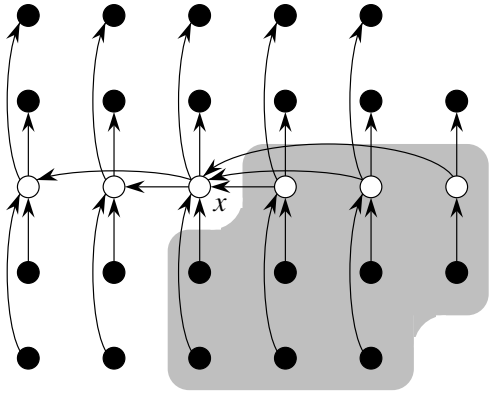



**Figure 9.1** Analysis of the algorithm SELECT. The $n$ elements are represented by small circles, and each group occupies a column. The medians of the groups are whitened, and the median-of-medians $x$ is labeled. (When finding the median of an even number of elements, we use the lower median.) Arrows are drawn from larger elements to smaller, from which it can be seen that 3 out of every full group of 5 elements to the right of $x$ are greater than $x$, and 3 out of every group of 5 elements to the left of $x$ are less than $x$. The elements greater than $x$ are shown on a shaded background.

2. Find the median of each of the $\lceil n/5 \rceil$ groups by first insertion sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
3. Use SELECT recursively to find the median $x$ of the $\lceil n/5 \rceil$ medians found in step 2. (If there are an even number of medians, then by our convention, $x$ is the lower median.)
4. Partition the input array around the median-of-medians $x$ using the modified version of PARTITION. Let $k$ be one more than the number of elements on the low side of the partition, so that $x$ is the $k$th smallest element and there are $n-k$ elements on the high side of the partition.
5. If $i = k$, then return $x$. Otherwise, use SELECT recursively to find the $i$th smallest element on the low side if $i < k$, or the $(i-k)$th smallest element on the high side if $i > k$.

To analyze the running time of SELECT, we first determine a lower bound on the number of elements that are greater than the partitioning element $x$. Figure 9.1 is helpful in visualizing this bookkeeping. At least half of the medians found in step 2 are greater than[1] the median-of-medians $x$. Thus, at least half of the $\lceil n/5 \rceil$ groups

[1] Because of our assumption that the numbers are distinct, we can say "greater than" and "less than" without being concerned about equality.

contribute 3 elements that are greater than $x$, except for the one group that has fewer than 5 elements if 5 does not divide $n$ exactly, and the one group containing $x$ itself. Discounting these two groups, it follows that the number of elements greater than $x$ is at least

$$3\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil-2\right) \;\geq\; \frac{3n}{10}-6\,.$$

Similarly, the number of elements that are less than $x$ is at least $3n/10-6$. Thus, in the worst case, SELECT is called recursively on at most $7n/10+6$ elements in step 5.

We can now develop a recurrence for the worst-case running time $T(n)$ of the algorithm SELECT. Steps 1, 2, and 4 take $O(n)$ time. (Step 2 consists of $O(n)$ calls of insertion sort on sets of size $O(1)$.) Step 3 takes time $T(\lceil n/5\rceil)$, and step 5 takes time at most $T(7n/10+6)$, assuming that $T$ is monotonically increasing. We make the assumption, which seems unmotivated at first, that any input of 140 or fewer elements requires $O(1)$ time; the origin of the magic constant 140 will be clear shortly. We can therefore obtain the recurrence

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \leq 140\,, \\ T(\lceil n/5\rceil)+T(7n/10+6)+O(n) & \text{if } n > 140\,. \end{cases}$$

We show that the running time is linear by substitution. More specifically, we will show that $T(n) \leq cn$ for some suitably large constant $c$ and all $n > 0$. We begin by assuming that $T(n) \leq cn$ for some suitably large constant $c$ and all $n \leq 140$; this assumption holds if $c$ is large enough. We also pick a constant $a$ such that the function described by the $O(n)$ term above (which describes the non-recursive component of the running time of the algorithm) is bounded above by $an$ for all $n > 0$. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

$$\begin{aligned} T(n) &\leq c\lceil n/5\rceil + c(7n/10+6)+an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an)\,, \end{aligned}$$

which is at most $cn$ if

$$-cn/10 + 7c + an \leq 0\,. \tag{9.2}$$

Inequality (9.2) is equivalent to the inequality $c \geq 10a(n/(n-70))$ when $n > 70$. Because we assume that $n \geq 140$, we have $n/(n-70) \leq 2$, and so choosing $c \geq 20a$ will satisfy inequality (9.2). (Note that there is nothing special about the constant 140; we could replace it by any integer strictly greater than 70 and then choose $c$ accordingly.) The worst-case running time of SELECT is therefore linear.

As in a comparison sort (see Section 8.1), SELECT and RANDOMIZED-SELECT determine information about the relative order of elements only by comparing elements. Recall from Chapter 8 that sorting requires $\Omega(n \lg n)$ time in the comparison model, even on average (see Problem 8-1). The linear-time sorting algorithms in Chapter 8 make assumptions about the input. In contrast, the linear-time selection algorithms in this chapter do not require any assumptions about the input. They are not subject to the $\Omega(n \lg n)$ lower bound because they manage to solve the selection problem without sorting.

Thus, the running time is linear because these algorithms do not sort; the linear-time behavior is not a result of assumptions about the input, as was the case for the sorting algorithms in Chapter 8. Sorting requires $\Omega(n \lg n)$ time in the comparison model, even on average (see Problem 8-1), and thus the method of sorting and indexing presented in the introduction to this chapter is asymptotically inefficient.