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 $A[j] \le x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

7.1-2
What value of q does PARTITION return when all elements in the array $A[p \mathinner{..} r]$ have the same value? Modify PARTITION so that $q = \lfloor (p + r)/2 \rfloor$ when all elements in the array $A[p \mathinner{..} r]$ have the same value.

7.1-3
Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

7.1-4
How would you modify QUICKSORT to sort into nonincreasing order?

7.2 Performance of quicksort

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which 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. (We prove this claim 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

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

where we tolerate the sloppiness from ignoring the floor and ceiling and from subtracting 1. By case 2 of the master theorem (Theorem 4.1), this recurrence has the solution $T(n) = \Theta(n \lg n)$. By equally balancing the two sides of the partition at every level of the recursion, we get 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 understand-
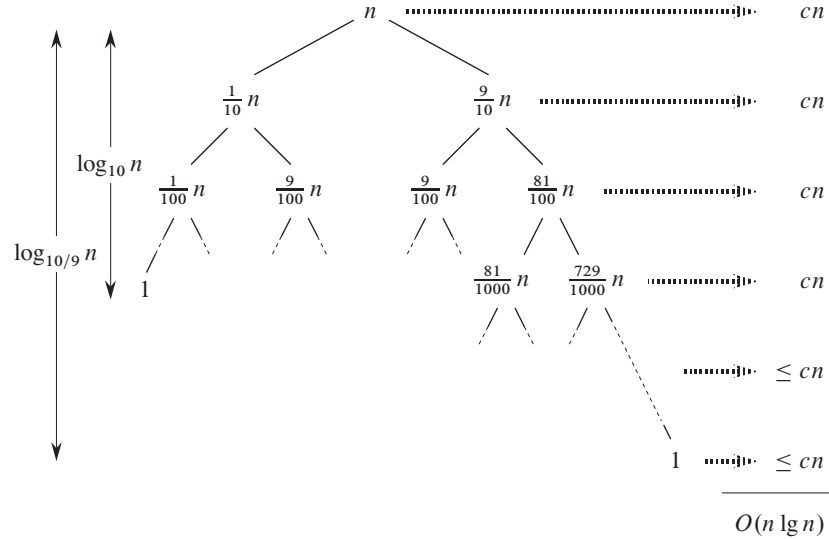
Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

ing 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

$$T(n) = T(9n/10) + T(n/10) + cn \ ,$$

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 the recursion reaches a boundary condition 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 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. Indeed, even a 99-to-1 split yields an $O(n \lg n)$ running time. In fact, 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.
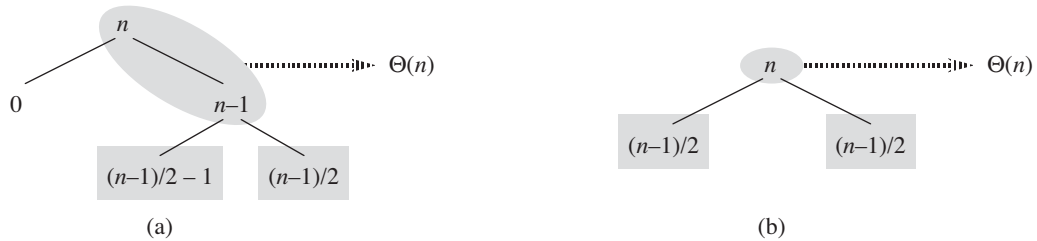
(a)   (b)

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).

Intuition for the average case

To develop a clear notion of the randomized behavior of quicksort, we must make
an assumption about how frequently we expect to encounter the various inputs.
The behavior of quicksort depends on 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, the partitioning is highly un-
likely to happen in the same way at every level, as our informal analysis has as-
sumed. 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
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,
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 $n - 1$ undergoes best-case
partitioning 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 sub-arrays 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 expected running time of a randomized version of quicksort in Section 7.4.2.

Exercises

7.2-1
Use the substitution method to prove that the recurrence $T(n) = T(n - 1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$, as claimed at the beginning of Section 7.2.

7.2-2
What is the running time of QUICKSORT when all elements of array A have the same value?

7.2-3
Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

7.2-4
Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure INSERTION-SORT would tend to beat the procedure QUICKSORT on this problem.

7.2-5
Suppose that the splits at every level of quicksort are in the proportion $1 - \alpha$ to α, where $0 < \alpha \leq 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $-\lg n / \lg \alpha$ and the maximum depth is approximately $-\lg n / \lg(1 - \alpha)$. (Don't worry about integer round-off.)