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 $A[p \mathinner{\ldotp\ldotp} r]$.

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

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

Combine: Since the subarrays are sorted in place, no work is needed to combine them: the entire array $A[p \mathinner{\ldotp\ldotp} r]$ 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 $A[p \mathinner{.\,.} r]$ 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 $A[p \mathinner{.\,.} r]$. 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.
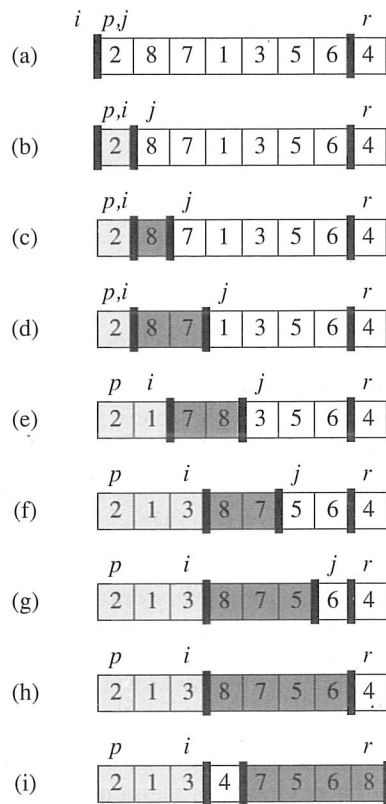
Figure 7.1 The operation of PARTITION on a sample array. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, 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
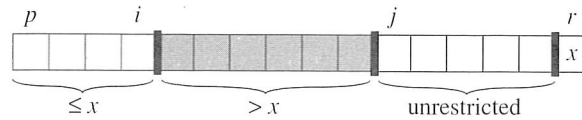
Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p \ldots r]$. The values in $A[p \ldots i]$ are all less than or equal to x, the values in $A[i + 1 \ldots j - 1]$ are all greater than x, and $A[r] = x$. The values in $A[j \ldots r - 1]$ can take on any values.

$A[j] > x$; 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 $A[j] \leq x$; i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Because of the swap, we now have that $A[i] \leq x$, 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 $A[p \ldots r]$ is $\Theta(n)$, where $n = r - p + 1$ (see Exercise 7.1-3).

Exercises

7.1-1
Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$.

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

7.1-3
Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.
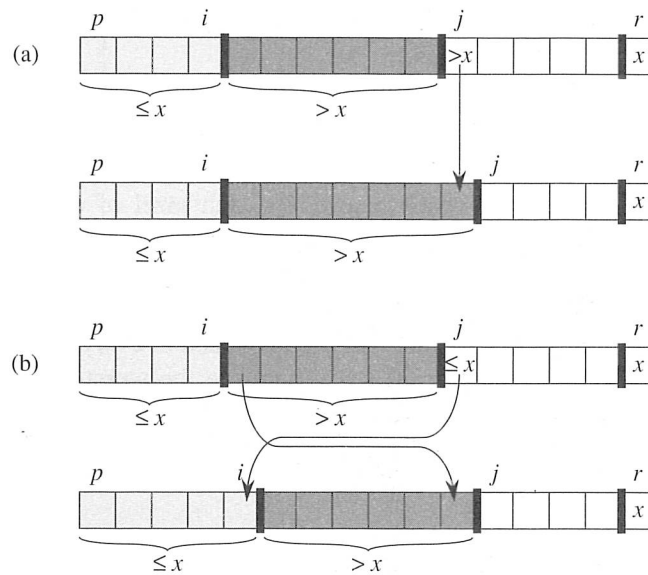
Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \leq x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

7.1-4
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, 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

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

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

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

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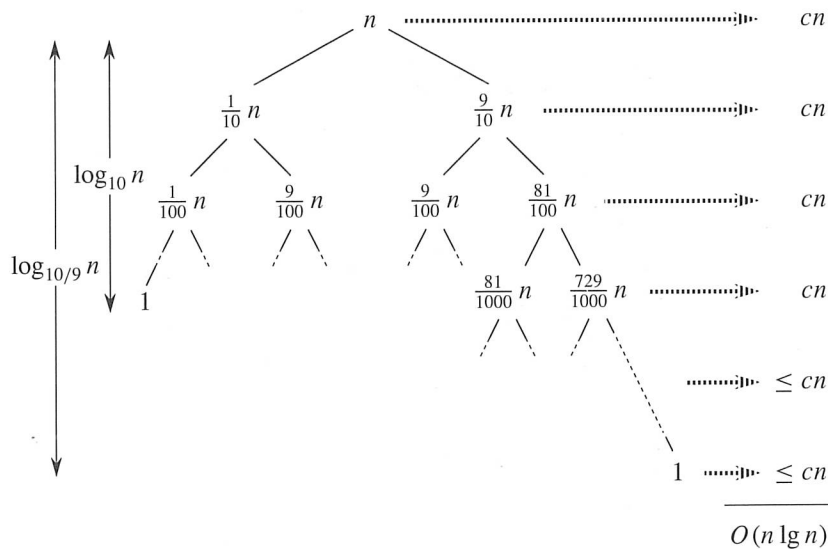
Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

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
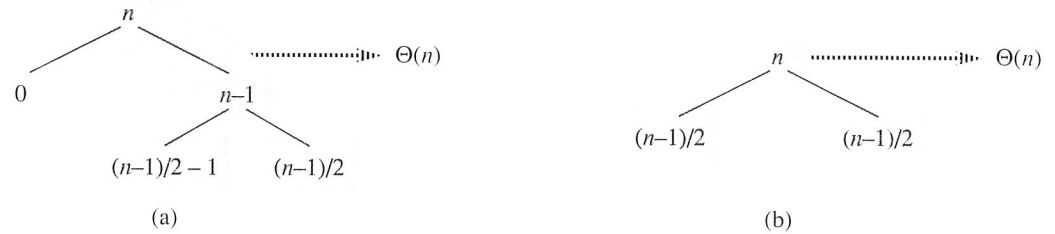
Figure 7.5 (a) Two levels of a recursion tree for quicksort. The partitioning at the root costs n and produces a "bad" split: two subarrays of sizes 0 and $n - 1$. The partitioning of the subarray of size $n - 1$ costs $n - 1$ and produces a "good" split: subarrays of size $(n - 1)/2 - 1$ and $(n - 1)/2$. (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.

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.)

7.2-6 ⋆
Argue that for any constant $0 < \alpha \leq 1/2$, the probability is approximately $1 - 2\alpha$ that on a random input array, PARTITION produces a split more balanced than $1 - \alpha$ to α.

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 $i \leftarrow$ RANDOM(p, r)
2 exchange $A[r] \leftrightarrow A[i]$
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** $q \leftarrow$ 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.

Exercises

7.3-1

Why do we analyze the average-case performance of a randomized algorithm and not its worst-case performance?

7.3-2

During the running of the procedure RANDOMIZED-QUICKSORT, how many calls are made to the random-number generator RANDOM in the worst case? How about in the best case? Give your answer in terms of Θ-notation.

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

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

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 $T(n) \leq cn^2$ for some constant c. Substituting this guess into recurrence (7.1), we obtain

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

The expression $q^2 + (n - q - 1)^2$ achieves a maximum over the parameter's range $0 \leq q \leq n - 1$ 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 $\max_{0 \leq q \leq n-1}(q^2 + (n - q - 1)^2) \leq (n - 1)^2 = n^2 - 2n + 1$. Continuing with our bounding of $T(n)$, we obtain

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

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 ith 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

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

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

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

It remains to compute $\Pr\{z_i \text{ is compared to } z_j\}$. Our analysis assumes that each pivot is chosen randomly and independently.

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, and because pivots are chosen randomly and independently, the probability that any given element is the first one chosen as a pivot is $1/(j - i + 1)$. Thus, we have

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

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

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

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

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

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

Exercises

7.4-1
Show that in the recurrence

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

$T(n) = \Omega(n^2)$.

7.4-2
Show that quicksort's best-case running time is $\Omega(n \lg n)$.

7.4-3
Show that $q^2 + (n - q - 1)^2$ achieves a maximum over $q = 0, 1, \ldots, n - 1$ when $q = 0$ or $q = n - 1$.

7.4-4
Show that RANDOMIZED-QUICKSORT's expected running time is $\Omega(n \lg n)$.

7.4-5
The running time of quicksort can be improved in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. When quicksort is called on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. Argue that this sorting algorithm runs in $O(nk + n \lg(n/k))$ expected time. How should k be picked, both in theory and in practice?

7.4-6 ⋆
Consider modifying the PARTITION procedure by randomly picking three elements from array A and partitioning about their median (the middle value of the three elements). Approximate the probability of getting at worst an α-to-$(1 - \alpha)$ split, as a function of α in the range $0 < \alpha < 1$.

Problems

7-1 *Hoare partition correctness*
The version of PARTITION given in this chapter is not the original partitioning algorithm. Here is the original partition algorithm, which is due to C. A. R. Hoare:

HOARE-PARTITION(A, p, r)

```
 1   x ← A[p]
 2   i ← p − 1
 3   j ← r + 1
 4   while TRUE
 5        do repeat j ← j − 1
 6              until A[j] ≤ x
 7           repeat i ← i + 1
 8              until A[i] ≥ x
 9           if i < j
10              then exchange A[i] ↔ A[j]
11              else return j
```

a. Demonstrate the operation of HOARE-PARTITION on the array $A = \langle 13, 19, 9,$ $5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, showing the values of the array and auxiliary values after each iteration of the **for** loop in lines 4–11.

The next three questions ask you to give a careful argument that the procedure HOARE-PARTITION is correct. Prove the following:

b. The indices i and j are such that we never access an element of A outside the subarray $A[p .. r]$.

c. When HOARE-PARTITION terminates, it returns a value j such that $p \leq j < r$.

d. Every element of $A[p .. j]$ is less than or equal to every element of $A[j+1 .. r]$ when HOARE-PARTITION terminates.

The PARTITION procedure in Section 7.1 separates the pivot value (originally in $A[r]$) from the two partitions it forms. The HOARE-PARTITION procedure, on the other hand, always places the pivot value (originally in $A[p]$) into one of the two partitions $A[p .. j]$ and $A[j + 1 .. r]$. Since $p \leq j < r$, this split is always nontrivial.

e. Rewrite the QUICKSORT procedure to use HOARE-PARTITION.

7-2   *Alternative quicksort analysis*

An alternative analysis of the running time of randomized quicksort focuses on the expected running time of each individual recursive call to QUICKSORT, rather than on the number of comparisons performed.

a. Argue that, given an array of size n, the probability that any particular element is chosen as the pivot is $1/n$. Use this to define indicator random variables $X_i = I\{i\text{th smallest element is chosen as the pivot}\}$. What is $E[X_i]$?

b. Let $T(n)$ be a random variable denoting the running time of quicksort on an array of size n. Argue that

$$E[T(n)] = E\left[\sum_{q=1}^{n} X_q \left(T(q-1) + T(n-q) + \Theta(n)\right)\right] . \tag{7.5}$$

c. Show that equation (7.5) can be rewritten as

$$E[T(n)] = \frac{2}{n} \sum_{q=2}^{n-1} E[T(q)] + \Theta(n) . \tag{7.6}$$

d. Show that

$$\sum_{k=2}^{n-1} k \lg k \le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 . \tag{7.7}$$

(*Hint:* Split the summation into two parts, one for $k = 2, 3, \ldots, \lceil n/2 \rceil - 1$ and one for $k = \lceil n/2 \rceil, \ldots, n - 1$.)

e. Using the bound from equation (7.7), show that the recurrence in equation (7.6) has the solution $E[T(n)] = \Theta(n \lg n)$. (*Hint:* Show, by substitution, that $E[T(n)] \le an \lg n$ for sufficiently large n and for some positive constant a.)

7-3 Stooge sort

Professors Howard, Fine, and Howard have proposed the following "elegant" sorting algorithm:

STOOGE-SORT(A, i, j)

```
1   if A[i] > A[j]
2       then exchange A[i] ↔ A[j]
3   if i + 1 ≥ j
4       then return
5   k ← ⌊(j − i + 1)/3⌋            ▷ Round down.
6   STOOGE-SORT(A, i, j − k)       ▷ First two-thirds.
7   STOOGE-SORT(A, i + k, j)       ▷ Last two-thirds.
8   STOOGE-SORT(A, i, j − k)       ▷ First two-thirds again.
```

a. Argue that, if $n = length[A]$, then STOOGE-SORT($A, 1, length[A]$) correctly sorts the input array $A[1 .. n]$.

b. Give a recurrence for the worst-case running time of STOOGE-SORT and a tight asymptotic (Θ-notation) bound on the worst-case running time.

c. Compare the worst-case running time of STOOGE-SORT with that of insertion sort, merge sort, heapsort, and quicksort. Do the professors deserve tenure?

7-4   Stack depth for quicksort

The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After the call to PARTITION, the left subarray is recursively sorted and then the right subarray is recursively sorted. The second recursive call in QUICKSORT is not really necessary; it can be avoided by using an iterative control structure. This technique, called *tail recursion*, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion.

QUICKSORT$'(A, p, r)$

```
1   while p < r
2        do ▷ Partition and sort left subarray.
3             q ← PARTITION(A, p, r)
4             QUICKSORT'(A, p, q − 1)
5             p ← q + 1
```

a. Argue that QUICKSORT$'(A, 1, length[A])$ correctly sorts the array A.

Compilers usually execute recursive procedures by using a *stack* that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. When a procedure is invoked, its information is *pushed* onto the stack; when it terminates, its information is *popped*. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The *stack depth* is the maximum amount of stack space used at any time during a computation.

b. Describe a scenario in which the stack depth of QUICKSORT$'$ is $\Theta(n)$ on an n-element input array.

c. Modify the code for QUICKSORT$'$ so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

7-5   Median-of-3 partition

One way to improve the RANDOMIZED-QUICKSORT procedure is to partition around a pivot that is chosen more carefully than by picking a random element from the subarray. One common approach is the *median-of-3* method: choose the pivot as the median (middle element) of a set of 3 elements randomly selected from the subarray. (See Exercise 7.4-6.) For this problem, let us assume that the elements in the input array $A[1 .. n]$ are distinct and that $n \geq 3$. We denote the

sorted output array by $A'[1 \mathinner{.\,.} n]$. Using the median-of-3 method to choose the pivot element x, define $p_i = \Pr\{x = A'[i]\}$.

a. Give an exact formula for p_i as a function of n and i for $i = 2, 3, \ldots, n - 1$. (Note that $p_1 = p_n = 0$.)

b. By what amount have we increased the likelihood of choosing the pivot as $x = A'[\lfloor (n + 1)/2 \rfloor]$, the median of $A[1 \mathinner{.\,.} n]$, compared to the ordinary implementation? Assume that $n \to \infty$, and give the limiting ratio of these probabilities.

c. If we define a "good" split to mean choosing the pivot as $x = A'[i]$, where $n/3 \leq i \leq 2n/3$, by what amount have we increased the likelihood of getting a good split compared to the ordinary implementation? (*Hint:* Approximate the sum by an integral.)

d. Argue that in the $\Omega(n \lg n)$ running time of quicksort, the median-of-3 method affects only the constant factor.

7-6   *Fuzzy sorting of intervals*

Consider a sorting problem in which the numbers are not known exactly. Instead, for each number, we know an interval on the real line to which it belongs. That is, we are given n closed intervals of the form $[a_i, b_i]$, where $a_i \leq b_i$. The goal is to *fuzzy-sort* these intervals, i.e., produce a permutation $\langle i_1, i_2, \ldots, i_n \rangle$ of the intervals such that there exist $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \leq c_2 \leq \cdots \leq c_n$.

a. Design an algorithm for fuzzy-sorting n intervals. Your algorithm should have the general structure of an algorithm that quicksorts the left endpoints (the a_i's), but it should take advantage of overlapping intervals to improve the running time. (As the intervals overlap more and more, the problem of fuzzy-sorting the intervals gets easier and easier. Your algorithm should take advantage of such overlapping, to the extent that it exists.)

b. Argue that your algorithm runs in expected time $\Theta(n \lg n)$ in general, but runs in expected time $\Theta(n)$ when all of the intervals overlap (i.e., when there exists a value x such that $x \in [a_i, b_i]$ for all i). Your algorithm should not be checking for this case explicitly; rather, its performance should naturally improve as the amount of overlap increases.

Chapter notes

The quicksort procedure was invented by Hoare [147]; Hoare's version appears in Problem 7-1. The PARTITION procedure given in Section 7.1 is due to N. Lomuto. The analysis in Section 7.4 is due to Avrim Blum. Sedgewick [268] and Bentley [40] provide a good reference on the details of implementation and how they matter.

McIlroy [216] showed how to engineer a "killer adversary" that produces an array on which virtually any implementation of quicksort takes $\Theta(n^2)$ time. If the implementation is randomized, the adversary produces the array after seeing the random choices of the quicksort algorithm.