

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, \dots, 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   $i \leftarrow \text{RANDOM}(p, r)$ 
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  $q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r)$ 
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

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

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, \dots, z_n , with z_i being the i th smallest

element. We also define the set $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$ 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

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

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

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

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

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

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 $q = 0, 1, \dots, n - 1$ 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: