## Section 7.1 – Description of quicksort

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

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 19 | 9 | 5 | 12 | 8 | 7 | 4 | 21 | 2 | 6 | 11 |
| 13 | 19 | 9 | 5 | 12 | 8 | 7 | 4 | 21 | 2 | 6 | 11 |
| 13 | 19 | 9 | 5 | 12 | 8 | 7 | 4 | 21 | 2 | 6 | 11 |
| 9 | 19 | 13 | 5 | 12 | 8 | 7 | 4 | 21 | 2 | 6 | 11 |
| 9 | 5 | 13 | 19 | 12 | 8 | 7 | 4 | 21 | 2 | 6 | 11 |
| 9 | 5 | 13 | 19 | 12 | 8 | 7 | 4 | 21 | 2 | 6 | 11 |
| 9 | 5 | 8 | 19 | 12 | 13 | 7 | 4 | 21 | 2 | 6 | 11 |
| 9 | 5 | 8 | 7 | 12 | 13 | 19 | 4 | 21 | 2 | 6 | 11 |
| 9 | 5 | 8 | 7 | 4 | 13 | 19 | 12 | 21 | 2 | 6 | 11 |
| 9 | 5 | 8 | 7 | 4 | 13 | 19 | 12 | 21 | 2 | 6 | 11 |
| 9 | 5 | 8 | 7 | 4 | 2 | 19 | 12 | 21 | 13 | 6 | 11 |
| 9 | 5 | 8 | 7 | 4 | 2 | 6 | 12 | 21 | 13 | 19 | 11 |
| 9 | 5 | 8 | 7 | 4 | 2 | 6 | 11 | 21 | 13 | 19 | 12 |

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 $q = \lfloor (p + r)/2 \rfloor$ when all elements in the array $A[p, \ldots, r]$ have the same value.

It will return $q = r$. We can update PARTITION to split elements that are equal to the pivot on both sides as follows:

(a) Count the number of elements $y$ such that $y = x$ and set this value to $c$;

(b) Subtract the final pivot index by $\lfloor c/2 \rfloor$.

The updated pseucode is stated below.

```
   Partition-Improved(A, p, r)
1  | x = A[r]
2  | i = j = p − 1
3  | for k = p to r − 1 do
4  | |  if A[k] ≤ x then
5  | |  |  j = j + 1
6  | |  |  exchange A[j] with A[k]
7  | |  |  if A[j] < x then
8  | |  |  |  i = i + 1
9  | |  |  |  exchange A[i] with A[j]
10 | exchange A[j + 1] with A[r]
11 | q = ⌊((i+1)+(j+1))/2⌋
12 | return q
```

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

The **for** loop of lines 3–6 iterates $n - 1$ times and each iteration does a constant amount of work. Thus, it is $O(n)$.

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

We just need to update the condition

$$A[j] \leq x,$$

to

$$A[j] \geq x.$$

## Section 7.2 – Performance of quicksort

7.2-1 Use the substitution method to prove 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.

Our guess is

$$T(n) \leq cn^2 - dn \ \forall\, n \geq n_0,$$

where $c$, $d$, and $n_0$ are positive constants. Substituting into the recurrence, yields

$$\begin{aligned} T(n) &\leq c(n-1)^2 - d(n-1) + en \\ &= cn^2 - 2cn + c - d(n-1) + en \quad (c = 1, d = 2e) \\ &\leq cn^2, \end{aligned}$$

where the last step holds as long as $n_0 \geq 2$.

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

As discussed in (7.1-2), when all elements are the same, $q$ will always be equal to $r$, which gives the worst-case split. Thus, QUICKSORT as implemented in Section 7.1, will run in $\Theta(n^2)$ in this case.

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.

The pivot index $q$ will always be 1, which gives a 0 to $n-1$ split. The recurrence will be $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.

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.

Lets assume that each item is out of order by no more than $k$ positions. Note that in the above scenario, $k$ usually can be bounded by a constant. In this case, INSERTION-SORT runs in $O(kn)$ (it will make at most $k$ swaps for each item of the array), which is close to linear for small $k$. On the other hand, *most* splits given by the PARTITION procedure will be no better than a $k-1$ to $n-k$ split. Assuming that it always give an $k-1$ to $n-k$ split, the recurrence of QUICKSORT will be $T(n) = T(k) + T(n-k) + \Theta(n)$, which is close to quadratic for small $k$.

7.2-5 Suppose that the splits at every level of quicksort are in the proportion $1-\alpha$ to $\alpha$, 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.)

Note that

$$\alpha \leq \frac{1}{2} \leq 1 - \alpha,$$

which implies $\alpha n \leq (1-\alpha)n$. Thus, the minimum depth occurs on the path from which the problem size is always divided by $1/\alpha$. This depth is the number of divisions of $n$ by $(1/\alpha)$ until reaching a value less than of equal to one, which is

$$\log_{1/\alpha} n = \frac{\lg n}{\lg(1/\alpha)} = \frac{\lg n}{-\lg \alpha} = -\frac{\lg n}{\lg \alpha}.$$

The maximum depth occurs on the path from which the problem size is always divided by $1/(1-\alpha)$. This depth is the number of divisions of $n$ by $1/(1-\alpha)$ until reaching a value less than or equal to one, which is

$$\log_{1/(1-\alpha)} n = \frac{\lg n}{\lg(1/(1-\alpha))} = \frac{\lg n}{-\lg(1-\alpha)} = -\frac{\lg n}{\lg(1-\alpha)}.$$

7.2-6 ($\star$) 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 $\alpha$.

Note that $\alpha$ denotes the proportion of the smallest split. Since the input array is random, the possible proportions for the smallest split forms a uniform probability distribution, such that

$$\Pr\left\{\left[0, \frac{1}{2}\right]\right\} = 1.$$

Thus, the probability of getting a more balanced split is

$$\begin{aligned}\Pr\left\{\left(\alpha, \frac{1}{2}\right]\right\} &= \Pr\left\{\left[\alpha, \frac{1}{2}\right]\right\} \\ &= \frac{1/2 - \alpha}{1/2 - 0} \\ &= \frac{1/2}{1/2} - \frac{\alpha}{1/2} \\ &= 1 - 2\alpha.\end{aligned}$$

## Section 7.3 – A randomized version of quicksort

7.3-1 Why do we analyze the expected running time of a randomized algorithm and not its worst-case running time?

> We can analyze the worst-case. However, due to the randomization, it is not very useful since we can not associate a specific input to a specific running time. On the other hand, we can calculate the expected running time, which takes into account all the possible inputs.

7.3-2 When Randomized-Quicksort runs, 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 $\Theta$-notation.

> First note that counting the number of calls to Random is the same as counting number of calls to Partition.
>
> The worst-case occurs when Partition always gives an $(n-1)$-to-0 split. Note that after the first $n-1$ pivots are selected, the remaining subarray will contain a single element. Since Partition is only called on subarrays of size greater than one, in the worst-case the number of calls to Partition is $n - 1 = \Theta(n)$.
>
> As for the best case, consider the array $A = [1, 2, 3]$. If the element 2 is the first to be selected as a pivot, the subarrays [1] and [3] will not be passed to Partition (both of them has size one) and the number of calls to Partition in this case is 1. In general, in the best-case the number of calls to Partition is $\lfloor n/2 \rfloor = \Theta(n)$.

## Section 7.4 – Analysis of quicksort

7.4-1 Show that in the recurrence

$$T(n) = \max_{0 \le q \le n-1}(T(q) + T(n-q-1)) + \Theta(n),$$
$$T(n) = \Omega(n^2).$$

We guess that $T(n) \ge cn^2$ for some constant $c$. Substituting into the recurrence, yields

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

The expression $q^2 + (n-q-1)^2$ achieves a maximum at $q = 0$ (proof on (7.4-3)). Thus, we have

$$\max_{0 \le q \le n-1}(q^2 + (n-q-1)^2) = (n-1)^2,$$

which give us the bound

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

since we pick the constant $c$ small enough so that the $\Theta(n)$ term dominates the $c(2n-1)$ term, which implies

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

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

Let $T(n)$ be the best-case time of QUICKSORT on an input of size $n$. We have the recurrence

$$T(n) = \min_{0 \le q \le n-1}(T(q) + T(n-q-1)) + \Theta(n).$$

We guess $T(n) \ge cn \lg n$ for some constant $c$. Substituting into the recurrence yields

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

For simplicify, assume that $n$ is odd. The expression $cq \lg q + (n-q-1)\lg(n-q-1)$ achieves a minimum when

$$q = n - q - 1,$$

which implies

$$q = \frac{n-1}{2}.$$

Thus, we have

$$\begin{aligned} T(n) &\ge c\left(\frac{n-1}{2}\right)\lg\left(\frac{n-1}{2}\right) + c\left(n - \frac{n-1}{2} - 1\right)\lg\left(n - \frac{n-1}{2} - 1\right) + \Theta(n) \\ &= c(n-1)\lg\left(\frac{n-1}{2}\right) + \Theta(n) \\ &= c(n-1)\lg(n-1) - c(n-1) + \Theta(n) \\ &= cn\lg(n-1) - c\lg(n-1) - c(n-1) + \Theta(n) \\ &\ge cn\lg\left(\frac{n}{2}\right) - c\lg(n-1) - c(n-1) + \Theta(n) \qquad (n \ge 2) \\ &= cn\lg n - cn - c\lg(n-1) - c(n-1) + \Theta(n) \\ &= cn\lg n - c(2n + \lg(n-1) - 1) + \Theta(n) \\ &\ge cn\lg n, \end{aligned}$$

since we pick the constant $c$ small enough so that the $\Theta(n)$ term dominates the $c(2n + \lg(n-1) - 1)$ term, which implies

$$T(n) = \Omega(n \lg n).$$

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

Let $f(q) = q^2 + (n - q - 1)^2$. We have

$$f'(q) = 2q + 2(n - q - 1) \cdot (-1) = 4q - 2n + 2,$$

and

$$f''(q) = 4.$$

Since the second derivative is positive, $f(q)$ achieves a maximum over $0, 1, \ldots, n - 1$ at either endpoint. But we have

$$f(0) = 0^2 + (n - 1)^2 = (n - 1)^2 + (n - (n - 1) - 1)^2 = f(n - 1),$$

which implies that both endpoints are maximum.

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

Combining equations (7.2) and (7.3), we get

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \\
&= \sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{k=1}^{n-i} \frac{2}{k+1} + \sum_{i=\lfloor n/2 \rfloor + 1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&\geq \sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&\geq \sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{k=1}^{n/2} \frac{2}{k+1} \\
&\geq \sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{k=1}^{n/2} \frac{1}{k} && \text{(since } k \geq 1\text{)} \\
&= \left\lfloor \frac{n}{2} \right\rfloor \cdot \left( \lg \left( \frac{n}{2} \right) + O(1) \right) && \text{(approx. of harmonic number)} \\
&= \Omega(n \lg n).
\end{aligned}$$

7.4-5 We can improve the running time of quicksort in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. Upon calling quicksort 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 we pick $k$, both in theory and in practice?

Lets first analyze the modified Quicksort. As in the standard Quicksort, it is easy to see that the worst-case of this modified version is still $O(n^2)$. As for the expected time, we can use a similar argument to the one used on Section 7.2, in which we saw that any split of constant proportionality on Quicksort yields a recursion tree of depth $\Theta(\lg n)$. Assume that Partition on this modified Quicksort always give a 99-to-1 split. The height $h$ of the recursion tree would be

$$\frac{n}{(100/99)^h} = k \to h = \log_{100/99} \frac{n}{k} \to h = \Theta\left( \lg \frac{n}{k} \right).$$

Since each recursion level has cost at most $cn$, the expected total cost of this modified Quicksort is $O(n \lg \frac{n}{k})$. As for the cost of the Insertion-Sort, note after running the modified Quicksort, every element will be out of order by at most $k$ positions. Thus, each iteration of the outer loop of Insertion-Sort will make at most $k$ swaps, which gives a running time of $O(nk)$. Finally, the cost of the whole algorithm is

$$O(nk) + O\left( n \lg \left( \frac{n}{k} \right) \right) = O\left( nk + n \lg \left( \frac{n}{k} \right) \right).$$

7.4-6 ($\star$) 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 $\alpha$-to-$(1-\alpha)$ split, as a function of $\alpha$ in the range $0 < \alpha < 1$.

First assume $0 < \alpha \leq 1/2$. There four ways to get a split worse than $\alpha$-to-$(1-\alpha)$:

(a) The index of exactly two elements are smaller than $\alpha n$

(b) The index of exactly two elements are greater than $n - \alpha n$.

(c) The index of all three elements are smaller than $\alpha n$.

(d) The index of all three elements are greater than $n - \alpha n$.

Since we want an approximation, assume that we can repeat the same element. The probability of cases (a) and (b) is

$$\Pr\{(a)\} = \Pr\{(b)\} = 3 \cdot \left( \frac{\alpha n}{n} \cdot \frac{\alpha n}{n} \cdot \frac{(1-\alpha)n}{n} \right) = 3\alpha^2 - 3\alpha^3,$$

in which the multiplication on the left is needed since there are $\binom{3}{1} = 3$ ways to pick one of three elements outside the desired range. The probability of cases (c) and (d) is

$$\Pr\{(c)\} = \Pr\{(d)\} = \frac{\alpha n}{n} \cdot \frac{\alpha n}{n} \cdot \frac{\alpha n}{n} = \alpha^3.$$

Thus, the probability of getting a split worse than $\alpha$-to-$(1-\alpha)$ is

$$\begin{aligned} 1 - \Pr\{(a) + (b) + (c) + (d)\} &= 1 - (\Pr\{(a)\} + \Pr\{(b)\} + \Pr\{(c)\} + \Pr\{(d)\}) \\ &= 1 - \left((3\alpha^2 - 3\alpha^3) + (3\alpha^2 - 3\alpha^3) + \alpha^3 + \alpha^3\right) \\ &= 1 - (6\alpha^2 - 4\alpha^3) \\ &= 1 - 6\alpha^2 + 4\alpha^3. \end{aligned}$$

The proof is similar for $1/2 \leq \alpha < 1$ and the result is the same.

## 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 do
5   |   | repeat
6   |   |   | j = j − 1
7   |   | until A[j] ≤ x
8   |   | repeat
9   |   |   | i = i + 1
10  |   | until A[i] ≥ x
11  |   | if i < j then
12  |   |   | exchange A[i] with A[j]
13  |   | else
14  |   |   | 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 **while** loop in lines 4–13.

The next three questions ask you to give a careful argument that the procedure HOARE-PARTITION is correct. Assuming that the subarray $A[p, \ldots, r]$ contains at least two elements, prove the following:

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

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

**d.** Every element of $A[p, \ldots, j]$ is less than or equal to every element of $A[j + 1, \ldots, r]$ 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 $A[p, \ldots, j]$ and $A[j + 1, \ldots, r]$. Since $p \le j < r$, this split is always nontrivial.

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

(a) The operation is illustrated below:

| $i$ | $x$ | | | | | | | | | | | $j$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 13 | 19 | 9 | 5 | 12 | 8 | 7 | 4 | 11 | 2 | 6 | 21 |
| | $x, i$ | | | | | | | | | | $j$ | |
| | 13 | 19 | 9 | 5 | 12 | 8 | 7 | 4 | 11 | 2 | 6 | 21 |
| | | $i$ | | | | | | | | $j$ | $x$ | |
| | 6 | 19 | 9 | 5 | 12 | 8 | 7 | 4 | 11 | 2 | 13 | 21 |
| | | | | | | | | | $j$ | $i$ | $x$ | |
| | 6 | 2 | 9 | 5 | 12 | 8 | 7 | 4 | 11 | 19 | 13 | 21 |

(b) Consider the following ***loop invariant***:

> At the beginning of each iteration of the **while** loop of lines 4-13, $i < j$ and the subarray $A[p, \ldots, j - 1]$ contains at least one element that is lower than or equal to $A[x]$. Similarly, the subarray $A[i + 1, \ldots, r]$ contains at least one element that is greater than or equal to $A[x]$.

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.

- **Initialization.** Prior to the **while** loop of lines 4-13, $i = p-1$ and $j = r-1$. Since $A[i+1, \ldots, r] = A[p, \ldots, j-1] = A[p, \ldots, r]$, the element $A[x]$ is present in both subarrays. Thus, the loop invariant is valid before the loop.
- **Maintenance.** From the loop invariant, the **for** loops of lines 5-7 and 8-10 will both stop on valid indices $i$ and $j$. The loop only goes to the next iteration if $i < j$. In that case, the elements $A[i]$ and $A[j]$ are exchanged, which ensures the loop invariant for the next iteration.
- **Termination.** At termination, the **for** loops of lines 5-7 and 8-10 will stop on valid indices $i$ and $j$ such that $i \ge j$ and the loop terminates before going to the next iteration.

The above loop invariant ensures that the for loops of lines 5-7 and 8-10 will never make $j < p$ or $i > r$, which implies that the HOARE-PARTITION procedure always access elements within the subarray $A[p, \ldots, r]$.

(c) From item (b), we have the lower bound $j \geq p$. As for the upper bound of $j$, note that:

i. If $A[r] > x$, the condition on line 7 will be false at least one time, which implies that line 6 will be executed at least twice. Since the initial value of $j$ is $r + 1$, in this case we have $j < r$.

ii. If $A[r] \leq x$, $A[r]$ will be exchanged with $A[p]$ in the first iteration of the **while** loop and line 6 will be executed for the second time in the next iteration. Thus, in this case we also have $j < r$.

These observations give us the bound $p \leq j < r$.

(d) Consider the following ***loop invariant***:

> At the beginning of each iteration of the **while** loop of lines 4-13, every element of the subarray $A[p, \ldots, \min(i, j)]$ is less than or equal to every element of the subarray $A[j + 1, \ldots, r]$.

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.

- **Initialization.** Prior to the **while** loop, $i = p - 1$, $j = r + 1$. Since the subarrays $A[p, \ldots, i]$ and $A[j, \ldots, r]$ are empty, the loop invariant is trivially satisfied.
- **Maintenance.** To see that each iteration maintains the loop invariant, note that the **for** loops of lines 5-7 and 8-10 will decrease $j$ and increase $i$ until find an $A[j] \leq x$ and an $A[i] \geq x$, respectivelly. At this point, the loop invariant (of previous iteration) along with the conditions of lines 7 and 10 ensures that every element of $A[p, \ldots, i - 1]$ is less than or equal to every element of $A[j + 1, \ldots, r]$. The only possible exceptions to the loop invariant in this iteration are the elements $A[i]$ and $A[j]$. Since $A[i] \geq x$ and $A[j] \leq x$, we have $A[i] \geq A[j]$. To go to the next iteration line 11 must be valid and the exchange of $A[i]$ with $A[j]$ at line 12 maintains the loop invariant.
- **Termination.** At termination, the **for** loop of lines 5-7 and 8-10 will stop on indices $i$ and $j$ such that $i \geq j$. Since $\min(i, j) = j$, the loop invariant (of previous iteration) along with the conditions of lines 7 and 10 ensures that every element of $A[p, \ldots, j]$ will be less than or equal to every element of $A[j + 1, \ldots, r]$.

(e) The pseudocode is stated below.

```
Hoare-Quicksort(A, p, r)
1   if p < r then
2       q = Hoare-Partition(A, p, r)
3       Hoare-Partition(A, p, q)
4       Hoare-Partition(A, q + 1, r)
```

7-2 ***Quicksort with equal element values***

The analysis of the expected running time of randomized quicksort in Section 7.4.2 assumes that all element values are distinct. In this problem, we examine what happens when they are not.

**a.** Suppose that all element values are equal. What would be randomized quicksort's running time in this case?

**b.** The PARTITION procedure returns an index $q$ such that each element of $A[p, \ldots, q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1, \ldots, r]$ is greater than $A[q]$. Modify the PARTITION procedure to produce a procedure PARTITION'$(A, p, r)$, which permutes the elements of $A[p, \ldots, r]$ and returns two indices $q$ and $t$, where $p \leq q \leq t \leq r$, such that

- all elements if $A[q, \ldots, t]$ are equal,
- each element of $A[p, \ldots, q-1]$ is less than $A[q]$, and
- each element of $A[r+1, \ldots, r]$ is greater than $A[q]$.

Like PARTITION, your PARTITION' procedure should take $\Theta(r-p)$ time.

**c.** Modify the RANDOMIZED-PARTITION procedure to call PARTITION', and name the new procedure RANDOMIZED-PARTITION'. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT'$(A, p, r)$ that calls RANDOMIZED-PARTITION' and recurses only on partitions of elements not known to be equal to each other.

**d.** Using QUICKSORT', how would you adjust the analysis in Section 7.4.2 to avoid the assumption that all elements are distinct?

(a) In this case, the condition on line 4 of the PARTITION procedure will always be valid and it will always give "bad" splits $((n-1)$-to-0). Thus, the running time will be $\Theta(n^2)$.

(b) This item is similar to the Question 7.1-2. The pseudocode of the modified PARTITION procedure is stated below.

```
Partition'(A, p, r)
1  | x = A[r]
2  | i = j = p − 1
3  | for k = p to r − 1 do
4  | |  if A[k] ≤ x then
5  | |  |  j = j + 1
6  | |  |  exchange A[j] with A[k]
7  | |  |  if A[j] < x then
8  | |  |  |  i = i + 1
9  | |  |  |  exchange A[i] with A[j]
10 | exchange A[j + 1] with A[r]
11 | q = i + 1
12 | t = j + 1
13 | return q, t
```

(c) The pseudocode of the modified RANDOMIZED-PARTITION QUICKSORT procedures are stated below.

```
Randomized-Partition'(A, p, r)
1  | i = Random(p, r)
2  | exchange A[r] with A[i]
3  | return Partition'(A, p, r)

Quicksort'(A, p, r)
1  | if p < r then
2  | |  q, t = Randomized-Partition'(A, p, r)
3  | |  Quicksort'(A, p, q − 1)
4  | |  Quicksort'(A, t + 1, r)
```

(d) We just need to rewrite the sentence

> In general, because we assume that element values are distinct, 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.

as

> Once a pivot $z_q$ is chosen with $z_i \leq z_q \leq z_j$, such that $i \neq q$ and $j \neq q$, we know that $z_i$ and $z_j$ cannot be compared at any subsequent time.

7-3 ***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 RANDOMIZED-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 $X_i = \mathrm{I}\{i\text{th smallest element is chosen as the pivot}\}$. What is $\mathrm{E}[X_i]$?

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

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

**c.** Show that we can rewrite equation (7.5) as

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

**d.** Show that

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

(*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 $\mathrm{E}[T(n)] = \Theta(n\lg n)$. (*Hint:* Show, by substitution, that $\mathrm{E}[T(n)] \le an\lg n$ for sufficiently large $n$ and for some positive constant $a$.)

(a) In each recursive call to RANDOMIZED-QUICKSORT, the pivot is randomly chosen among the $n$ elements of the input subarray. Thus, we have

$$\mathrm{E}[X_i] = \Pr\{X_i = 1\} = \frac{1}{n}.$$

(b) Each call to PARTITION tales $\Theta(n)$ and once the $q$th

(c) Each call to QUICKSORT selects a pivot $q$, such that $1 \le q \le n$, that partitions the array into two subarrays of sizes $q-1$ and $n-q$. The indicator random variable $X_q$ indicates whether the element with index $q$ is selected as the pivot. Since only one element can be chosen as the pivot at a given call to QUICKSORT and the running time of each call is $\Theta(n)$, the running time of QUICKSORT can be written as

$$T(n) = \sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n)),$$

which implies

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

(d)

$$\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\sum_{q=1}^{n} X_q(T(q-q) + T(n-q) + \Theta(n))\right] \\
&= \sum_{q=1}^{n}\mathrm{E}[X_q(T(q-1) + T(n-q) + \Theta(n))] \\
&= \sum_{q=1}^{n}\left(\frac{1}{n}\cdot\mathrm{E}[(T(q-1) + T(n-q) + \Theta(n))]\right) \\
&= \frac{1}{n}\cdot\sum_{q=1}^{n}\mathrm{E}[T(q-1)] + \frac{1}{n}\cdot\sum_{q=1}^{n}\mathrm{E}[T(n-q)] + \frac{1}{n}\cdot\sum_{q=1}^{n}\Theta(n) \\
&= \frac{1}{n}\cdot\sum_{q=2}^{n-1}\mathrm{E}[T(q)] + \frac{1}{n}\cdot\Theta(1) + \frac{1}{n}\cdot\sum_{q=2}^{n-1}\mathrm{E}[T(q)] + \frac{1}{n}\cdot\Theta(1) + \Theta(n) \\
&= \frac{2}{n}\cdot\sum_{q=2}^{n-1}\mathrm{E}[T(q)] + \Theta(n).
\end{aligned}$$

(e) We guess that

$$\mathrm{E}[T(n)] \leq an \lg n,$$

for some constant $a$. Substituting into the recurrence yields

$$\begin{aligned}
\mathrm{E}[T(n)] &\leq \frac{2}{n} \sum_{q=2}^{n-1} (aq \lg q) + \Theta(n) \\
&= \frac{2a}{n} \sum_{q=2}^{n-1} (q \lg q) + \Theta(n) \\
&\leq \frac{2a}{n} \left( \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + \Theta(n) \\
&= an \lg n - a \frac{1}{4} n + \Theta(n) \\
&\leq an \lg n,
\end{aligned}$$

since we pick the constant $a$ large enough so that the $a(1/4)n$ term dominates the $\Theta(n)$ term.

7-4 ***Stack depth for quicksort***

The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in QUICKSORT is not really necessary; we can avoid it 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:

```
Tail-Recursive-Quicksort(A, p, r)
1  while p < r do
2      // Partition and sort left subarray
3      q = Partition(A, p, r)
4      Tail-Recursive-Quicksort(A, p, q - 1)
5      p = q + 1
```

**a.** Argue that TAIL-RECURSIVE-QUICKSORT$(A, 1, A.length)$ 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. Upon calling a procedure, 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 TAIL-RECURSIVE-QUICKSORT'S stack depth is $\Theta(n)$ on an $n$-element array.

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

(a) After the PARTITION call, the algorithm calls itself with arguments $A, p, q-1$, sets $p = q+1$, and repeat the same operations. Since the only difference to the next iteration is the new value of $p$, the loop is similar as calling itself with arguments $A, q+1, r$. Thus, TAIL-RECURSIVE-QUICKSORT produces the same result of QUICKSORT.

(b) If the PARTITION procedure always select the largest element of the array as the pivot, the left subarray will always have size $n-1$, and the stack depth will be $\Theta(n)$.

(c) To reduce the maximum stack depth, we should always give to the tail recursion the larger of the two subproblems. The updated pseudocode is stated below.

```
Tail-Recursive-Quicksort-Improved(A, p, r)
1  while p < r do
2      // Partition and sort left subarray
3      q = Partition(A, p, r)
4      if q < (p + r)/2 then
5          Tail-Recursive-Quicksort(A, p, q - 1)
6          p = q + 1
7      else
8          Tail-Recursive-Quicksort(A, q + 1, r)
9          r = q - 1
```

Each recursive call reduces the problem size by at least half. Thus, the stack depth is $O(\lg n)$.

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 $A[1,\ldots,n]$ are distinct and that $n \geq 3$. We denote the sorted output array by $A'[1,\ldots,n]$. 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 $A[1,\ldots,n]$, compared with 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 with 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.

(a) Note that the number of 3-permutations on a set of $n$ elements is

$$\frac{n!}{(n-3)!} = n(n-1)(n-2).$$

To choose the element $i$ as the pivot, one element needs to be within the first $i-1$ positions of the array, one elements needs to be within the last $n-i$ positions of the array, and one element needs to be the $i$th element. Also note that each combination of elements that chooses the element $i$ as the pivot has $3! = 6$ permutations; hence 6 ways to be selected. Thus, we have

$$p_i = \frac{3! \cdot 1 \cdot (i-1) \cdot (n-i)}{n!/(n-3)!} = \frac{6(i-1)(n-i)}{n(n-1)(n-2)}.$$

(b) We have

$$\begin{aligned}
p_{\lfloor (n+1)/2 \rfloor} &= \frac{6 \cdot \left(\left\lfloor \frac{n+1}{2} \right\rfloor - 1\right)\left(n - \left\lfloor \frac{n+1}{2} \right\rfloor\right)}{n(n-1)(n-2)} \\
&\leq \frac{6 \cdot \left(\frac{n+1}{2} - 1\right)\left(n - \frac{n+1}{2}\right)}{n(n-1)(n-2)} \\
&= \frac{6 \cdot \left(\frac{n-1}{2}\right)\left(\frac{n-1}{2}\right)}{n(n-1)(n-2)} \\
&= \frac{3}{2} \cdot \frac{(n-1)(n-1)}{n(n-1)(n-2)} \\
&= \frac{3}{2} \cdot \frac{(n-1)}{n(n-2)}.
\end{aligned}$$

Then, we have the ratio

$$\lim_{n\to\infty} \frac{\frac{3}{2} \cdot \frac{n-1}{n(n-2)}}{\frac{1}{n}} = \lim_{n\to\infty} \frac{3}{2} \frac{n(n-1)}{n(n-2)} = \frac{3}{2} = 1.5.$$

(c) To get a "good" split with the median-of-3 method, the pivot can not be within the first $\lfloor n/3 \rfloor$ elements or within the last $\lceil n/3 \rceil$ elements. Thus, we have

$$\Pr\{\text{good split with median-of-3}\} = \sum_{i=\lceil n/3 \rceil}^{\lfloor 2n/3 \rfloor} p_i \approx \sum_{i=n/3}^{2n/3} p_i = \sum_{i=n/3}^{2n/3} \frac{6(i-1)(n-i)}{n(n-1)(n-2)} = \frac{6}{n(n-1)(n-2)} \cdot \sum_{i=n/3}^{2n/3} (i-1)(n-i).$$

Note that

$$\begin{aligned}
\sum_{x=n/3}^{2n/3} (x-1)(n-x) &\approx \int_{n/3}^{2n/3} (x-1)(n-x)dx \\
&= \int_{n/3}^{2n/3} (nx - x^2 - n + x)dx \\
&= -\frac{1}{3}x^3 + \frac{1}{2}x^2(n+1) - xn \Big|_{n/3}^{2n/3} \\
&= \frac{13}{162}n^3 - \frac{1}{6}n^2,
\end{aligned}$$

which implies

$$\Pr\{\text{good split with median-of-3}\} \approx \frac{6}{n(n-1)(n-2)} \left(\frac{13}{162}n^3 - \frac{1}{6}n^2\right) = \frac{\frac{13}{27}n^3 - n^2}{n(n-1)(n-2)}.$$

Then, we have the ratio

$$\frac{\Pr\{\text{good split with median-of-3}\}}{\Pr\{\text{good split with one pivot}\}} = \lim_{n\to\infty} \frac{\frac{13}{27}n^3 - n^2}{\frac{n(n-1)(n-2)}{\frac{1}{3}}} = \lim_{n\to\infty} \frac{\frac{13}{27}n^3 - n^2}{\frac{n^3 - n^2 - 2n}{\frac{1}{3}}} = \lim_{n\to\infty} \frac{\frac{13}{27}}{\frac{1}{3}} \approx 1.44.$$

(d) The only difference is on the choice of the pivot. However, even if the middle element is always chosen as the pivot (which is the best case), the height of the recursion tree will be $\Theta(\lg n)$. Since each recursion level takes $\Theta(n)$, the running time is still $\Omega(n \lg n)$.

7-6 ***Fuzzy sorting of intervals***

Consider a sorting problem in which we do not know the numbers 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 $a_i \leq b_i$. We wish to ***fuzzy-sort*** these intervals, i.e., to produce a permutation $\langle i_1, i_2, \ldots, i_n \rangle$ of the intervals such that for $j = 1, 2, \ldots, n$, there exist $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \leq c_2 \leq \cdots \leq c_n$.

**a.** Design a randomized algorithm for fuzzy-sorting $n$ intervals. Your algorithm should have the general structure of an algorithm that quicksorts the left endpoints (the $a_i$ values), 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 becomes progressively 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 shoukd not be checking for this case explicitly; rather, its performance should naturally improve as the amount of overlap increases.

(a) Note that any subset of intervals that share a common point are already sorted. Using this notion, we can sort an array of fuzzy intervals with an algorithm similar to quicksort, but with a customized partition procedure that returns two indices $q$ and $t$, where $p \leq q \leq t \leq r$, such that

- There exist $x$ such that $x \in [a_i, b_i]$ for all $q \leq i \leq t$. That is, any permutation of the subarray $A[q, \ldots, t]$ is sorted;
- For all $j < q$, there exist $c_j \in [a_j, b_j]$ such that $c_j < b_i$ for all $q \leq i \leq t$. That is, every interval of $A[p, \ldots, q-1]$ can stay before every interval of $A[q, \ldots, t]$ in the sorted array;
- For all $j > t$, there exist $c_j \in [a_j, b_j]$ such that $c_j > a_i$ for all $q \leq i \leq t$. That is, every interval of $A[t+1, \ldots, r]$ can stay after every interval of $A[q, \ldots, t]$ in the sorted array.

The pseudocode is stated below.

```
Fuzzy-Partition(A, p, r)
 1  a = A[r].a
 2  b = A[r].b
 3  i = j = p − 1
 4  for k = p to r − 1 do
 5      // A[k] can be placed before the pivot
 6      if A[k].a ≤ b then
 7          j = j + 1
 8          exchange A[j] with A[k]
 9          // intervals (A[k] and pivot) do not overlap
10          if A[j].b < a then
11              i = i + 1
12              exchange A[i] with A[j]
13          // intervals (A[k] and pivot) overlap
14          else
15              if A[k].a > a then
16                  a = A[k].a
17              if A[k].b < b then
18                  b = A[k].b
19  exchange A[j + 1] with A[r]
20  q = i + 1
21  t = j + 1
22  return q, t
```

```
Fuzzy-Randomized-Partition(A, p, r)
1  i = Random(p, r)
2  exchange A[r] with A[i]
3  return Fuzzy-Partition(A, p, r)
```

```
Fuzzy-Sort(A, p, r)
1  if p < r then
2      q, t = Fuzzy-Randomized-Partition(A, p, r)
3      Fuzzy-Sort(A, p, q − 1)
4      Fuzzy-Sort(A, t + 1, r)
```

(b) When all intervals share a common point, the array is already sorted. In this case, there will be only one call to FUZZY-PARTITION, which will return $q = p$ and $t = r$. Thus, the algorithm will run in $\Theta(n)$.

The general case is a little tricky to proof.