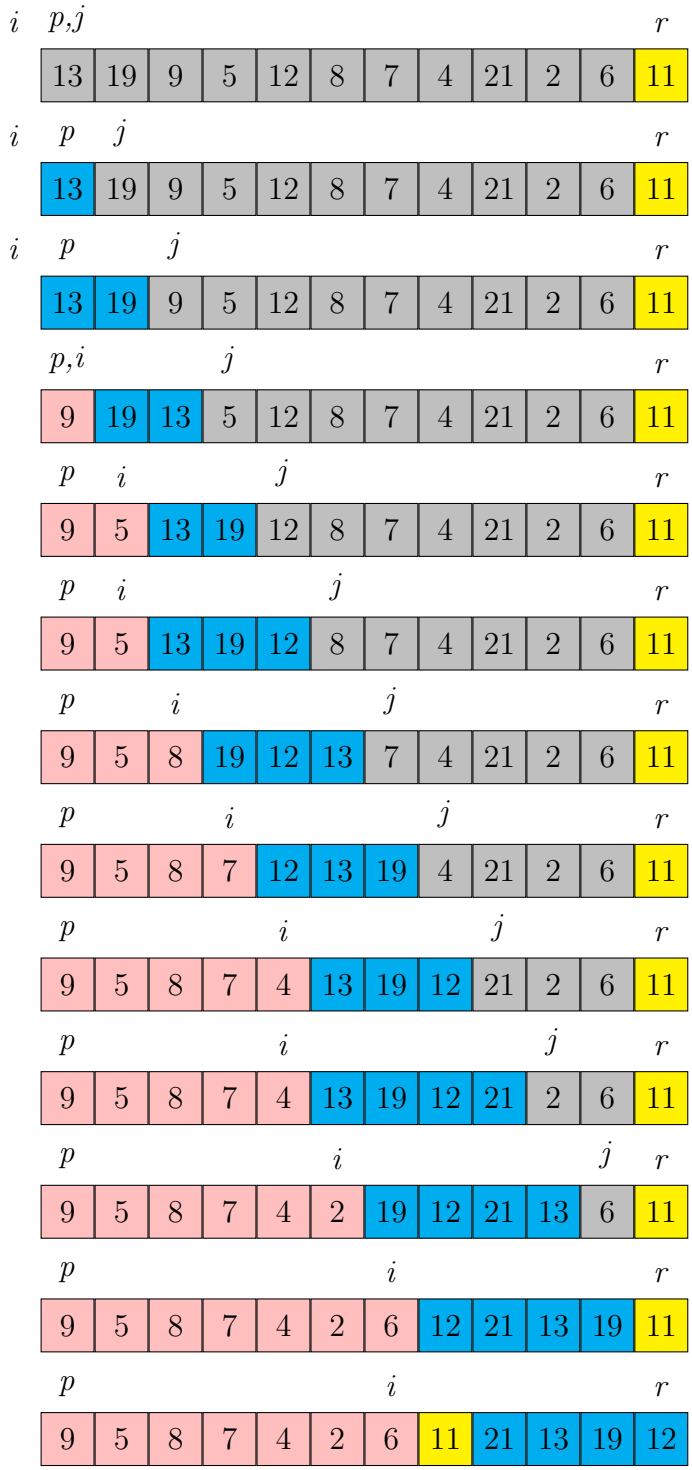
Exercises in Section 7.1

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

Solution



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

   Solution If all elements in the subarray $A[p:r]$ have the same value, the condition of the **if** statement will be always true. Therefore, i increases by $r-p$ as there are $r-p$ iterations in total. Hence, after the **for** loop, $i = (p-1) + (r-p) = r-1$, and the returned value is $q = i+1 = r$. Below is an amended program that returns $\lfloor (p+r)/2 \rfloor$ when all elements in the subarray $A[p:r]$ have the same value:

Algorithm 1 PARTITION'(A, p, r)

```
1: x = A[r]
2: i = p − 1
3: for j = p to r − 1 do
4:    if A[j] ≠ A[j + 1] then
5:       break
6:    end if
7:    return ⌊(p + r)/2⌋
8: end for
9: for j = p to r − 1 do
10:   if A[j] ≤ x then
11:      i = i + 1
12:      exchange A[i] with A[j]
13:   end if
14: end for
15: exchange A[i + 1] with A[r]
16: return i + 1
```

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

   Solution All lines except the **for** loop take constant time, and there are $n-1 = O(n)$ iterations in total. Therefore, the running time of PARTITION is $O(n)$. In the **for** loop, at least a comparison of $A[j]$ and x is made, which guarantees the tightness of this upper bound. Hence, the running time of PARTITION is $\Theta(n)$.

4. Modify QUICKSORT to sort into monotonically decreasing order.

   Solution The program QUICKSORT remains the same:

Algorithm 2 QUICKSORT(A, p, r)

1: **if** $p < r$ **then**
2: $q = \text{PARTITION}(A, p, r)$
3: $\text{QUICKSORT}(A, p, q-1)$
4: $\text{QUICKSORT}(A, q+1, r)$
5: **end if**

The program PARTITION is modified in order to sort the array in decreasing order:

Algorithm 3 PARTITION'(A, p, r)

1: $x = A[r]$
2: $i = p - 1$
3: **for** $j = p$ to $r - 1$ **do**
4: **if** $A[j] \geq x$ **then**
5: $i = i + 1$
6: exchange $A[i]$ with $A[j]$
7: **end if**
8: **end for**
9: exchange $A[i+1]$ with $A[r]$
10: **return** $i + 1$

Exercises in Section 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.
   Solution Assume $T(k) = O(k^2)$ for $k < n$, therefore $T(n-1) \le c(n-1)^2$. We have

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

   for sufficiently large c. So, $T(n) = O(n^2)$. On the other hand, assume $T(k) = \Omega(k^2)$ for $k < n$, therefore $T(n-1) \ge d(n-1)^2$. We have

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

   for sufficiently small d. Hence, $T(n) = \Theta(n^2)$.

2. What is the running time of QUICKSORT when all elements of array A have the same value?
   Solution If all elements of array A have the same value, in Exercise 7.2-2 we have shown that each call of PARTITION(A, p, r) produces one subproblem with $r - p$ elements and one with 0 elements, that is, the worst-case. Hence, the running time of QUICKSORT is the worst-case running time, which is $\Theta(n^2)$.

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.
   Solution If the array A contains distinct elements and is sorted in decreasing order, in each call of PARTITION(A, p, r), $A[r]$ is strictly smaller than any other elements. Therefore, neither there will be any swaps of elements nor $i = 0$ will increase, and thus $q = i + 1 = 1$ is returned, and it produces one subproblem with $r - p$ elements and one with 0 elements, which is also the worst-case. Hence, the running time of QUICKSORT is $\Theta(n^2)$.

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. Explain persuasively why the procedure INSERTION-SORT might tend to beat the procedure QUICKSORT on this problem.

Solution We will count the approximate numbers of assignments needed to sort an almost-sorted array A with INSERTION-SORT and QUICKSORT. Note that the variables c and d appear below may depend on n, but with small coefficient.

First, consider INSERTION-SORT (Recall the program in Section 2.1). In each of the $n-1$ iterations of the **for** loop, there are at least 3 assignments, together with a **while** loop with 2 assignments per iteration. Since A is almost sorted, the **while** loop is not likely to iterate many times, say c time in average, where c should be small. Thus, there are $(3+2c)(n-1)$ assignments.

Then, consider QUICKSORT. In each PARTITION(A, p, r), there are at least 5 assignments (including the 3 iterations for the exchange of two elements), together with almost $4(r-p)$ assignments, because $x = A[r]$ is almost the largest element by assumption. We say there are $5+4(r-p-d)$ assignments, where d is small. It produces one subproblem with $r-p-d$ elements and one with d elements. We ignore the latter one with d elements because it is small. Therefore, the total number of assignments is approximately

$$\begin{aligned}
&(5+4(n-1-d))+(5+4(n-1-2d))+\cdots+(5+4(1)) \\
&= 5\frac{n-2}{d}+4\frac{((n-d)(\frac{n-2}{d}))}{2} \\
&= \left(\frac{n-2}{d}\right)(5+2(n-d)).
\end{aligned}$$

By the assumption that c and d are small, and suppose n is sufficiently large, we can safely say

$$\begin{aligned}
\frac{n-2}{d} &> (3+2c), \text{ and} \\
5+2(n-d) &> n-1.
\end{aligned}$$

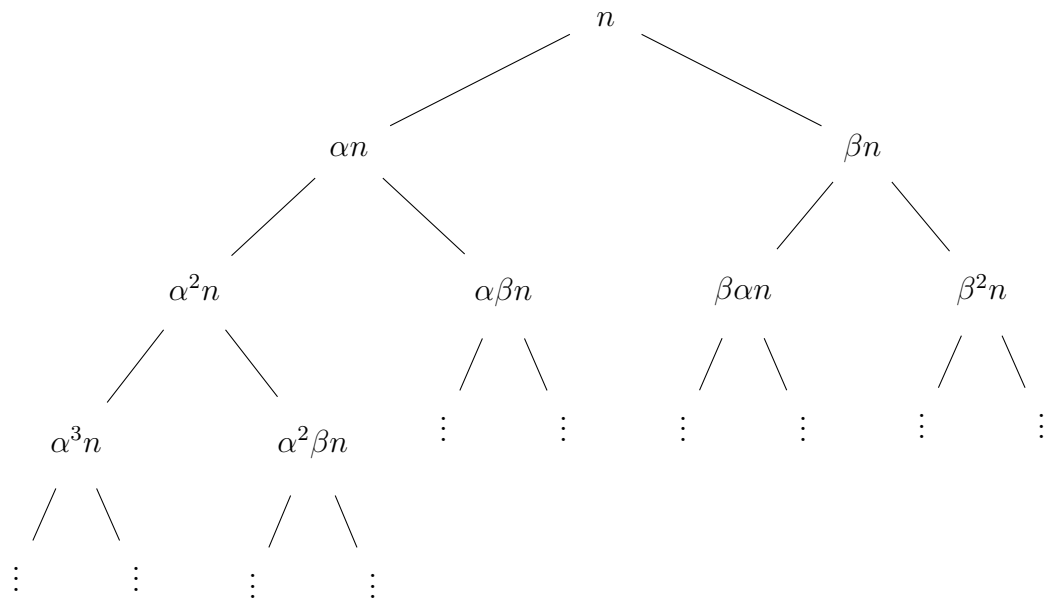
Hence, we have

$$\left(\frac{n-2}{d}\right)(5+2(n-d)) > (3+2c)(n-1),$$

that is, INSERTION-SORT is likely to beat QUICKSORT on this problem by costing less running time.

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

Solution The recursion tree is shown below:



Since $\alpha \leq \beta$, in depth d of the recursion tree, the smallest element is $\alpha^d n$ and the largest element is $\beta^d n$. Therefore, a leaf is of minimum depth when it is of the form $\alpha^d n$, and we need to find the smallest integer d such that $\alpha^d n \leq 1$, that is,

$$d_{min} = \lceil \log_\alpha (1/n) \rceil = \lceil \log_{1/\alpha} n \rceil \approx \log_{1/\alpha} n.$$

On the other hand, a leaf is of maximum depth when it is of the form $\beta^d n$, and we need to find the smallest integer d such that $\beta^d n \leq 1$, that is,

$$d_{max} = \lceil \log_\beta (1/n) \rceil = \lceil \log_{1/\beta} n \rceil \approx \log_{1/\beta} n.$$

6. Consider an array with distinct elements and for which all permutations of the elements are equally likely. Argue that for any constant $0 < \alpha \leq 1/2$, the probability is approximately $1 - 2\alpha$ that PARTITION produces a split at least as balanced as $1 - \alpha$ to α.

Solution In the result array of PARTITION(A, p, r), $A[r]$ is greater than all elements before, and less than all elements after, and the index of $A[r]$ is returned. Thus, the returned value of PARTITION(A, p, r) is q if and only if $A[r]$ is the q-th smallest element of the subarray. Suppose the subarray has n elements, given the assumption that all permutations of the elements in this subarray are equally likely, $A[r]$ has equal probability to be the 1-st to

n-th smallest elements. Hence, PARTITION(A, p, r) produces a split at least as balanced as $1 - \alpha$ to α if and only if $A[r]$ is the αn-th to $(1 - \alpha)n$-th smallest element, that is,

$$\textit{probability} = \frac{(1 - \alpha)n - \alpha n + 1}{(n - 1) + 1} = 1 - 2\alpha + 1/n \approx 1 - 2\alpha.$$

Exercises in Section 7.3

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

   Solution The running time of a randomized algorithm is random and does not depend on the input parameters (e.g. arrays). Therefore, the worst-case running time, where we discuss the choice of the input parameters, is meaningless for randomized algorithms.

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 Θ-notation.

   Solution Note that RANDOM is called once RANDOM-PARTITION is called. There are $\Theta(n)$ leaves in the recursion tree, and the number of nodes should be a constant product of the number of leaves. Thus, in both worst case and best case, RANDOM is called $\Theta(n)$ times.

Exercises in Section 7.4

1. Show that the recurrence $T(n) = \max\{T(q) + T(n-q-1) : 0 \le q \le n-1\} + \Theta(n)$ has a lower bound of $T(n) = \Omega(n^2)$.

   Solution Assume $T(k) \ge ck^2$ for $k < n$. By substitution, we have

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

   where the maximum is given by

$$\begin{aligned} q^2 + (n-q-1)^2 &= q^2 + (n-1)^2 - 2q(n-1) + q^2 \\ &= (n-1)^2 + 2q(q-(n-1)) \\ &\le (n-1)^2. \end{aligned}$$

   We have to check this upper bound is tight. Take $q = n-1$, it is easy to see the term after $(n-1)^2$ vanishes. Therefore, the maximum is indeed $(n-1)^2$. Substitute to the recurrence, we have

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

   for sufficiently small c. Hence, $T(n) = \Omega(n^2)$. □

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

   Solution The recurrence associated to the best case for quicksort is

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

   Assume $T(k) \ge ck \lg k$ for $k < n$. By substitution, we have

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

   We define the function

$$f(x) = x \lg(x) + (n-1-x)\lg(n-1-x).$$

The first derivative of $f(x)$ is

$$\begin{aligned} f'(x) &= \frac{d}{dx}(x \lg x) + \frac{d}{dx}\left((n-1-x)\lg(n-1-x)\right) \\ &= (\lg x + 1) - (\lg(n-1-x)+1) \\ &= \lg\left(\frac{x}{n-1-x}\right), \end{aligned}$$

which evaluates to 0 at $x = (n-1)/2$, where we have

$$f(\frac{n-1}{2}) = (n-1)\lg\left(\frac{n-1}{2}\right).$$

Therefore, the minimum appearing in the recurrence is bounded from below by $f(\frac{n-1}{2}) = (n-1)\lg\left(\frac{n-1}{2}\right)$. Substituting to the recurrence, we get

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

by taking sufficiently small c such that all other terms are dominated by the function hidden in $\Theta(n)$. Hence, we conclude $T(n) = \Omega(n \lg n)$. □

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

$$\begin{aligned} q^2 + (n-q-1)^2 &= q^2 + (n-1)^2 - 2q(n-1) + q^2 \\ &= (n-1)^2 + 2q(q-(n-1)) \\ &\leq (n-1)^2, \end{aligned}$$

where the equality achieves if and only if $q = 0$ or $q = n-1$. □

4. Show that RANDOMIZED-QUICKSORT's expected running time is $\Omega(n \lg n)$.
Solution With exactly the same justification in the proof of *Lemma 7.1*, the running time of RANDOMIZED-QUICKSORT is $\Omega(X)$, where X is the number of elements comparisons performed. Note that we are looking for a lower bound, so we can ignore the time spent

outside of the **for** loop. Then, in the proof of *Theorem 7.4* we have

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

In fact, the harmonic series is not only $O(\lg n)$, but a tight $\Theta(\lg n)$ (see equation (A.9) on page 1142). Therefore, we have

$$E[X] = \sum_{i=1}^{n-1} \Theta(\lg n) = \Theta(n \lg n).$$

Hence, the expected running time of RANDOMIZED-QUICKSORT is $\Omega(E[X]) = \Omega(n \lg n)$. □

5. Coarsening the recursion, as we did in Problem 2-1 for merge sort, is a common way to improve the running time of quicksort in practice. We modify the base case of the recursion so that if the array has fewer than k elements, the subarray is sorted by insertion sort, rather than by continued recursive calls to quicksort. Argue that the randomized version of this sorting algorithm runs in $O(nk + n \lg(n/k))$ expected time. How should you pick k, both in the theory and in practice?

   Solution As the leaves of the recursion tree, there are expected to be $O(n/k)$ calls of insertion sort, with $O(k^2)$ running time for each. So, the total running time of all calls of insertion sort is $O(O(n/k) \cdot O(k^2)) = O(nk)$.

   Again, the recursion tree have an expected depth of $O(\lg(n/k))$, with running time $O(n)$ on each level. Therefore, the running time for all internal nodes (that is, other than the leaves) is $O(O\lg(n/k) \cdot n) = O(n \lg(n-k))$. Finally, we conclude the expected running time of this sorting algorithm is $O(nk + n \lg(n/k))$.

   Suppose the function hidden in the O-notation is $T(n,k) = \alpha nk + \beta n \lg(n/k)$, we have

$$\frac{\partial}{\partial k} T(n,k) = \alpha n - \beta \frac{n}{k \ln 2},$$

   which equals 0 when $k = \frac{\beta}{\alpha \ln 2}$. So, in this case, we should pick $k = \lfloor \frac{\beta}{\alpha \ln 2} \rfloor$ or $\lceil \frac{\beta}{\alpha \ln 2} \rceil$.

   In practice, the case could be more complicated. For example, the function hidden in the O-notation possibly has extra asymptotically smaller terms, and the restriction on the size of n can also affect the choice of k. One should consider multiple factors before making the appropriate choice of k.

6. Consider modifying the PARTITION procedure by randomly picking three elements from subarray $A[p : r]$ and partitioning about their median (the middle value of the three elements). Approximate the probability of getting worst than an α-to-$(1-\alpha)$ split, as a function of α in the range $0 < \alpha < 1/2$.

Solution For simplicity, we assume the elements in A are $1, 2, \ldots, n$, and one element can be taken multiple times which does not affect the result. We want to find the probability of getting worse than an α-to-$(1-\alpha)$ split, that is, the median locates at $[1, \alpha]$ or $[1-\alpha, 1]$. Since these two intervals are symmetric, we need to find one side only. Note that the median is in $[1, \alpha]$ if and only if at least two out of three chosen elements are in $[1, \alpha]$. The probability that exactly two elements are in $[1, \alpha]$ is

$$\binom{3}{2}\alpha^2(1-\alpha) = 3\alpha^2 - 3\alpha^3,$$

and the probability that all three elements are in $[1, \alpha]$ is α^3. Therefore, the probability that the median is in $[1, \alpha]$ is

$$(3\alpha^2 - 3\alpha^3) + \alpha^3 = 3\alpha^2 - 2\alpha^3.$$

Symmetric to $[1-\alpha, 1]$, the final answer is $6\alpha^2 - 4\alpha^3$.

Problems in Chapter 7

1. **Hoare partition correctness**

   The version of PARTITION given in this chapter is not the original partitioning algorithm. Here is the original partitioning algorithm, which is due to C.A.R. Hoare.

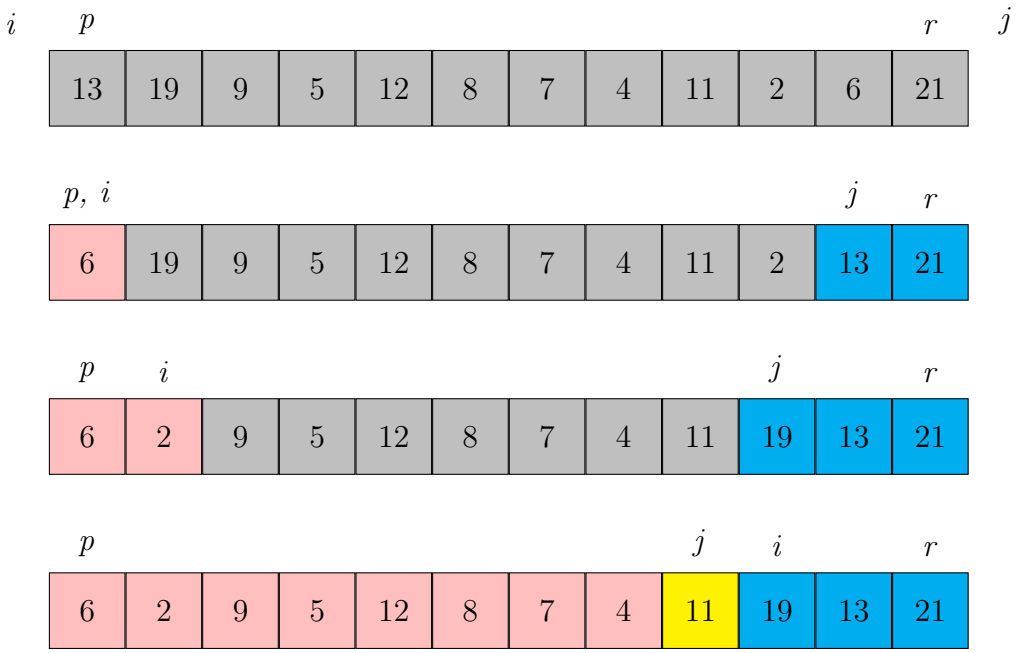
Algorithm 4 HOARE-PARTITION(A, p, r)

```
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
15:   end if
16: end while
```

   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 the indices i and j after each iteration of the **while** loop in lines 4-13.

   Solution In the **while** loop in lines 4-13, the program looks for the rightmost element that is smaller than or equal to x, and the leftmost element that is greater than or equal to x, and swap them if they are not the same element. With this procedure, the entries that i passes by are all smaller than x, and the entries that j passes by are all greater than x.

   In this example, $x = A[p] = 13$. Below shows the procedure of HOARE-PARTITION in sorting the array A:



Finally, $j = 9$ is returned, and the array A is separated into two subarrays $\langle 6, 2, 9, 5, 12, 8, 7, 4\rangle$ and $\langle 19, 13, 21\rangle$.

b. Describe how the PARTITION procedure in Section 7.1 differs from HOARE-PARTITION when all elements in $A[p : r]$ are equal. Describe a practical advantage of HOARE-PARTITION over PARTITION for use in quicksort.

Solution In Exercise 7.1-2, we have shown that if all elements in the subarray $A[p : r]$ are equal, the program PARTITION(A, p, r) returns r. In HOARE-PARTITION(A, p, r), however, if all elements are equal, the program will swap the first and the last element, swap the second and last second element, ..., until i and j meet. The final value of j, which is returned, is almost the middle of $A[p, r]$ (in fact, it is $\lfloor (p + r)/2 \rfloor$).

In practice, especially when there are repeated values in the array, HOARE-PARTITION tends to divide the subarray in a more balanced way, reducing the average running time.

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

c. The indices i and j are such that the procedure never accesses an element of A outside the subarray $A[p : r]$.

Solution In the first iteration, since $x = A[p]$, the final value of i is p, and the final value of j is at least p and at most r. Now, suppose at the end of an iteration we have $p \le i \le r$ and $p \le j \le r$. If the loop is not terminated and we proceed to the next iteration, it is easy to see that $i < j$ and $A[i] \le A[j]$. With these relations, in the next iteration, the new position of i is at most the old position of j, and the new position of j is at least the old position of i, again yielding $p \le i \le r$ and $p \le j \le r$. □

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

Solution It suffices to show $j \neq r$. If the **for** loop terminates in the first iteration, then $j = i = p \neq r$. Otherwise, there are at least 2 iterations, and j decreases by at least 2, that is, the final value of j is less than or equal to $(r+1) - 2 = r - 1 < r$. □

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

Solution As discussed in a., all elements in $A[p : i-1]$ are smaller than or equal to x, and all elements in $A[j+1 : r]$ are greater than or equal to x. When HOARE-PARTITION terminates, we have $i \geq j$. The proof is almost done except possibly $i = j$ and we need to show $A[j] \leq x$. This is also obvious because $i = j$ implies $A[i] = A[j] = x$. □

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 $p \leq j < r$, neither partition is empty.

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

Solution

Algorithm 5 QUICKSORT(A, p, r)

```
1: if p ≠ r then
2:     q = HOARE-PARTITION(A, p, r)
3:     QUICKSORT(A, p, q)
4:     QUICKSORT(A, q + 1, r)
5: end if
```

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. This problem examines what happens when they are not.

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

Solution Since all elements in the array are equal, the randomized array has no difference. The associated recurrence is

$$T(n) = T(n-1) + \Theta(n)$$

with solution $T(n) = \Theta(n^2)$.

b. The PARTITION procedure returns an index 1 such that each element of $A[p : q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1 : r]$ is greater than $A[q]$.

Modify the PARTITION procedure to produce a procedure PARTITION'(A, p, r), which permutes the elements of $A[q : r]$ and returns two indices q and t, where $p \le q \le t \le r$, such that

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

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

Solution

Algorithm 6 PARTITION'(A, p, r)

```
1:  x = A[r]
2:  i = p − 1
3:  for j = p to r − 1 do
4:      if A[j] ≤ x then
5:          i = i + 1
6:          exchange A[i] with A[j]
7:      end if
8:  end for
9:  exchange A[i + 1] with A[r]
10: t = i + 1
11: while i ≥ p and A[i] == A[t] do
12:     i = i − 1
13: end while
14: k = p
15: while k < i do
16:     if A[k] == A[t] then
17:         exchange A[k] with A[i]
18:         while i ≥ p and A[i] == A[t] do
19:             i = i − 1
20:         end while
21:     end if
22:     k = k + 1
23: end while
24: return i + 1, t
```

Let $n = r - p + 1$ be the size of the subarray, and note that $\Theta(n) = \Theta(r-p)$. In each of the **while** loop starting at line 15, k decreases by 1, so there are $O(n)$ iterations in total. The **while** loop starting at line 18 looks terrible, but since i decreases by 1 in

each iteration, the total number of iterations of this nested **while** loop over the whole program is $O(n)$. Hence, the running time of PARTITION'(A, p, r) is $O(n)$.

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 where elements are not known to be equal to each other.

Solution

Algorithm 7 RANDOMIZED-PARTITION'(A, p, r)

1: $i =$ RANDOM(p, r)
2: exchange $A[p]$ with $A[i]$
3: **return** PARTITION'(A, p, r)

Algorithm 8 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'$(A, t + 1, r)$
5: **end if**

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

Solution The analysis are almost the same. Since $q \leq t$, we removed all elements that have the same value of the pivot instead of only the pivot itself, thus the subarray $A[p : r]$ is divided into two even smaller subarrays using QUICKSORT' than using QUICKSORT. Therefore, all analysis in Section 7.4.2 are still valid in order to find an upper bound of the running time of QUICKSORT'.

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. As in the analysis of Section 7.4.2, assume that the values of the elements are distinct.

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 probability to define indicator random variable $X_i = I\{i\text{-th smallest element is chosen as the pivot}\}$. What is $E[X_i]$?

Solution In line 1 of RANDOMIZED-PARTITION'(A, p, r), the pivot is randomly determined from p to r with equal probability $1/n$. For each $p \leq i \leq r$, define the indicator

$X_i = I\{i\text{-th smallest element is chosen as the pivot}\}$, we have

$$E[X_i] = P(i\text{-th smallest element is chosen as the pivot}) = 1/n.$$

□

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

$$E[T(n)] = E\left[\sum_{q=1}^{n} X_q \left(T(q-1) + T(n-q) + \Theta(n)\right)\right]. \tag{7.2}$$

Solution For each $1 \le q \le n$, if $A[q]$ is chosen as the pivot, the recurrence will be

$$T(n) = T(q-1) + T(n-q) + \Theta(n).$$

Therefore, $T(n)$ is the summation of the product of X_q and the associated recurrence:

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

Take expectation on each side, we have

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

□

c. Show how to rewrite equation 7.2 as

$$E[T(n)] = \frac{2}{n}\sum_{q=1}^{n-1} E[T(q)] + \Theta(n). \tag{7.3}$$

Solution By the linearity of expectation, and note that for a particular q, X_q and $T(q-1) + T(n-q) + \Theta(n)$ are independent. We have

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

By the trick of change of index, we have

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

Hence,

$$\begin{aligned}RHS &= \frac{2}{n}\sum_{q=1}^{n} E\left[T(q-1)\right] + \Theta(n) \\ &= \frac{2}{n}\sum_{q=0}^{n-1} E\left[T(q)\right] + \Theta(n) \\ &= \frac{2}{n}\sum_{q=1}^{n-1} E\left[T(q)\right] + \Theta(n),\end{aligned}$$

where the last equality holds because $T(0)$ takes constant time $\Theta(1)$, and the resulting term $(2/n)\Theta(1) = \Theta(n^{-1})$ is dominated by the $\Theta(n)$ additive term. □

d. Show that

$$\sum_{q=1}^{n-1} q \lg q \leq \frac{n^2}{2}\lg n - \frac{n^2}{8} \tag{7.4}$$

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

Solution

$$\begin{aligned}\sum_{q=1}^{n-1} q \lg q &\leq \sum_{q=1}^{\lceil n/2 \rceil - 1} q \lg q + \sum_{q=\lceil n/2 \rceil}^{n-1} q \lg q \\ &\leq \sum_{q=1}^{\lceil n/2 \rceil - 1} q \lg (n/2) + \sum_{q=\lceil n/2 \rceil}^{n-1} q \lg n \\ &= \sum_{q=1}^{\lceil n/2 \rceil - 1} q \lg (n) - \sum_{q=1}^{\lceil n/2 \rceil - 1} q + \sum_{q=\lceil n/2 \rceil}^{n-1} q \lg n \\ &= \sum_{q=1}^{n-1} q \lg (n) - \sum_{q=1}^{\lceil n/2 \rceil - 1} q.\end{aligned}$$

The left summation evaluates

$$\sum_{q=1}^{n-1} q \lg (n) = \frac{n(n-1)}{2}\lg n = \frac{n^2}{2}\lg n - \frac{n}{2}\lg n,$$

and the right summation evaluates

$$\sum_{q=1}^{\lceil n/2\rceil-1} q = \frac{\lceil n/2\rceil(\lceil n/2\rceil-1)}{2} \geq \frac{(n/2)(n/2-1)}{2} = \frac{n^2}{8} - \frac{n}{4}.$$

For $n \geq 2$, we have

$$\frac{n}{2}\lg n \geq \frac{n}{2} \geq \frac{n}{4}.$$

Hence,

$$\begin{aligned}\sum_{q=1}^{n-1} q\lg q &\leq \left(\frac{n^2}{2}\lg n - \frac{n}{2}\lg n\right) - \left(\frac{n^2}{8} - \frac{n}{4}\right) \\ &= \left(\frac{n^2}{2}\lg n - \frac{n^2}{8}\right) - \left(\frac{n}{2}\lg n - \frac{n}{4}\right) \\ &\leq \frac{n^2}{2}\lg n - \frac{n^2}{8}.\end{aligned}$$

□

e. Using the bound from equation 7.4, show that the recurrence in equation 7.3 has the solution $E[T(n)] = O(n\lg n)$. (*Hint:* Show, by substitution, that $E[T(n)] \leq an\lg n$ for sufficiently large n and for some positive constant a.)

Solution Suppose $E[T(n)] \leq an\lg n$. By substitution, we have

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

for sufficiently large a. □

4. **Stooge Sort**

Professors Howard, Fine and Howard have proposed a deceptively simple sorting algorithm, named stooge sort in their honor, appearing on the following page.

Algorithm 9 STOOGE-SORT(A, p, r)

```
1: if A[p] > A[r] then
2:     exchange A[p] with A[r]
3: end if
4: if p + 1 < r then
5:     k = ⌊(r − p + 1)/3⌋
6:     STOOGE-SORT(A, p, r − k)
7:     STOOGE-SORT(A, p + k, r)
8:     STOOGE-SORT(A, p, r − k)
9: end if
```

a. Argue that the call STOOGE-SORT$(A, 1, n)$ correctly sorts the array $A[1 : n]$.

Solution The base cases are where $p+1 \geq r$, or equivalently the subarray has less than or equal to 2 elements. If $p+1 < r$, consider the three recursive calls of STOOGE-SORT. Let A_i denote the i-th third of A, and we say $A_i \leq A_j$ if all elements in A_i is smaller than or equal to any element in A_j. The first call sorts the first two-thirds of the subarray. Therefore, after the first call, we have

$$A_1 \leq A_2.$$

The second call sorts the last two-thirds. Note that any element in A_3 that was smaller than some element in A_1 must be sent to A_2 during the second call, because they must appear before all elements that were originally in A_2. Therefore, after the second call, we have

$$A_2 \leq A_3 \quad \text{and} \quad A_1 \leq A_3.$$

The last call sorts the first two thirds again and does not change A_3. Therefore, elements in A_3 are still larger than elements both in A_1 and A_2. Hence, after the three calls of STOOGE-SORT, we have the relation

$$A_1 \leq A_2 \leq A_3,$$

that is, the subarray is indeed sorted. □

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

Solution In fact, the recurrence associated to STOOGE-SORT is the same for any case. Once STOOGE-SORT is called on an array of size n, it calls 3 times of STOOGE-SORT on subarrays of size $2n/3$ plus a constant running time. Therefore, the recurrence is

$$T(n) = 3T(2n/3) + \Theta(1).$$

By the master theorem, the solution is $T(n) = \Theta(n^{\log_{3/2} 3}) \approx \Theta(n^{2.7095})$.

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?

Solution The worst-case running time of merge sort and heapsort (See Exercise 6.4-4) are $\Theta(n \lg n)$, and the worst-case running time of insertion sort and quicksort are $\Theta(n^2)$. None of them is worse than the scary running time $\Theta(n^{2.7095})$ of STOOGE-SORT. These professors may find difficulty in career, but Xiamen University Malaysia always welcomes you to join.

5. **Stack depth for quicksort**

The QUICKSORT procedure of Section 7.1 makes two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the low side of the partition and then it recursively sorts the high side of the partition. The second recursive call in QUICKSORT is not really necessary, because the procedure can instead use an iterative control structure. This transformation technique, called ***tail-recursion elimination***, is provided automatically by good compilers. Applying tail-recursion elimination transforms QUICKSORT into the TRE-QUICKSORT procedure.

Algorithm 10 TRE-QUICKSORT(A, p, r)

```
1: while p < r do
2:     q = PARTITION(A, p, r)
3:     TRE-QUICKSORT(A, p, q − 1)
4:     p = q + 1
5: end while
```

a. Argue that TRE-QUICKSORT$(A, 1, n)$ correctly sorts the array $A[1 : n]$.

Solution The base case where $p \geq r$ is trivial. Now assume the correctness of TRE-QUICKSORT in the recursion calls, we claim that in each iteration of the **while** loop, the value of q is strictly larger than that in the previous iteration, and $A[p_0 : q]$, where p_0 denote the original value of p before the **while** loop, is sorted at the end of this iteration.

In the first iteration, q is larger than or equal to all elements in $A[p_0 : q - 1]$, together with TRE-QUICKSORT$(A, p, q - 1)$ guarantees the subarray $A[p_0 : q]$ to be sorted. Suppose $A[p_0 : q']$, where q' is the value of q in the last iteration, is sorted. By line 4 we have $p = q' + 1$. Thus, the new value $q \geq p > q'$, as desired. Moreover, TRE-QUICKSORT$(A, p, q-1)$ succeeds in sorting the subarray $A[q'+1, q]$ with similar reason. Note that any element in $A[q'+1, q]$ is larger than all elements in $A[p_0 : q']$ as guaranteed by the partition function, we will have $A[p_0 : q]$ sorted, and the proposition is proved.

This **while** loop will finally stop because q increases by at least 1 in each iteration, and p is defined 1 larger than q. Therefore, at the termination when $p \geq r$, the latest value q is at least $r-1$ and at least $A[p_0 : r-1]$ is sorted. Similarly, if $A[p_0 : r-1]$ is the case, $A[r]$ must be the largest element by the property of the partition function, and of course, $A[p_0 : r]$ is sorted. □

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 called, its information is ***pushed*** onto the stack, and 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 stake space used at any time during a computation.

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

Solution Consider the case that all elements in A has the same value. Therefore, the **while** loop only has one iteration, and therefore calls TRE-QUICKSORT once on the subarray $A[1 : r-1]$. The last call is TRE-QUICKSORT$(A, 1, 1)$, therefore the maximum number of procedure calls stored in the stack is n, with $O(1)$ space for each. Therefore the stack depth is $\Theta(n)$.

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

Solution

Algorithm 11 MODIFIED-TRE-QUICKSORT(A, p, r)

```
1: while p < r do
2:    q = PARTITION(A, p, r)
3:    if q < (p + r)/2 then
4:       MODIFIED-TRE-QUICKSORT(A, p, q − 1)
5:       p = q + 1
6:    else
7:       MODIFIED-TRE-QUICKSORT(A, q + 1, r)
8:       r = q − 1
9:    end if
10: end while
```

Due to the **if** statement which restricts the size of the subarray to be at most half of its original size, there are at most $\lg n$ nested calls of MODIFIED-TRE-QUICKSORT, and

therefore the stack depth is $O(\lg n)$. In the worst case, the array reduces (almost) exactly half and the stack depth is $\Theta(\lg n)$. To see its expected running time is $O(n \lg n)$, consider the average case where $q \approx (p+r)/2$ whenever PARTITION(A, p, r) is called. For simplicity, we assume n is a power of 2. The associated recurrence is

$$\begin{aligned} T(n) &= T(n/2) + T(n/4) + \cdots + T(2) + T(1) + f(n) \\ &= \sum_{i=1}^{n} T(n/2^i) + f(n), \end{aligned}$$

where $f(n) = \Theta(n \lg n)$ because there are $\Theta(\lg n)$ iterations of the **while** loop, with $\Theta(n)$ running time (the partition procedure) for each. We will use Akra-Bazzi method (See Section 4.7) to solve this recurrence. First, find a real number p such that

$$\left(\frac{1}{2}\right)^p + \left(\frac{1}{4}\right)^p + \cdots + \left(\frac{1}{n}\right)^p = 1,$$

which has solution $0 < p < 1$. Let $f(n) = cn \lg n$ be the function hidden in $\Theta(n \lg n)$, we have

$$\begin{aligned} \int_1^n \frac{cx \lg x}{x^{p+1}} dx &= \frac{c}{\ln 2} \int_1^n \frac{\ln x}{x^p} dx \\ &= \frac{c}{\ln 2} \left(\frac{n^{1-p} \ln n}{1-p} - \int_1^n \frac{x^{-p}}{1-p} dx \right) \\ &= \frac{c}{\ln 2} \left(\frac{n^{1-p} \ln n}{1-p} - \frac{n^{1-p} - 1}{(1-p)^2} \right) \\ &= \Theta(n^{1-p} \lg n). \end{aligned}$$

By Akra-Bazzi method, we conclude that

$$T(n) = \Theta\left(n^p \left(1 + \Theta(n^{1-p} \lg n)\right)\right) = \Theta(n \lg n).$$

6. **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. A 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, assume that the n elements in the input array $A[p:r]$ are distinct and that $n \geq 3$. Denote the sorted version of $A[p:r]$ by $z_1, z_2, \ldots, z_n$. Using the median-of-3 method to choose the pivot element x, define $p_i = \Pr\{x = z_i\}$.

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

Solution For simplicity, we assume $A = \{1, 2, \ldots, n\}$, and suppose $n \geq 3$. There are $\binom{n}{3}$ ways with equal probability to choose 3 elements from n elements, and each combination determines a unique median. In Exercise 7.2.6, we have shown that the

index returned by PARTITION is exactly the index of the pivot in the sorted subarray. For each $1 \le i \le n$, $A[i]$ is the median of a combination if and only if the three chosen elements are one less than $A[i]$, one equals $A[i]$, and one greater than $A[i]$. There are $(i-1)(1)(n-i)$ such combinations. So, the probability of $A[i]$ being the pivot, resulting a split of A at i, is

$$p_i = \frac{(i-1)(n-i)}{\binom{n}{3}}.$$

b. By what amount does the median-of-3 method increase the likelihood of choosing the pivot to be $x = z_{\lfloor (n+1)/2 \rfloor}$, the median of $A[p:r]$, compared with the ordinary implementation? Assume that $n \to \infty$, and give the limiting ratio of these probabilities.

Solution With the median-of-3 method, the probability of choosing the pivot to be $x = z_{\lfloor (n+1)/2 \rfloor}$ is

$$\begin{aligned} p_{\lfloor (n+1)/2 \rfloor} &= \frac{(\lfloor (n+1)/2 \rfloor - 1)(n - \lfloor (n+1)/2 \rfloor)}{\binom{n}{3}} \\ &= \frac{-\lfloor (n+1)/2 \rfloor^2 + (n+1)\lfloor (n+1)/2 \rfloor - n}{\binom{n}{3}}. \end{aligned}$$

In the ordinary implementation, this probability is $1/n$. Take the limit of the ratio of these probabilities (note that the floor function can be ignored as $n \to \infty$), we have

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

Hence, the likelihood of choosing the median is increased by $\left(\frac{3}{2} - 1\right) \times 100\% = 50\%$.

c. Suppose that we define a "good" split to mean choosing the pivot as $x = z_i$, where $n/3 \le i \le 2n/3$. By what amount does the median-of-3 method increase the likelihood of getting a good split compared with the ordinary implementation? (*Hint:* Approximate the sum by an integral.)

Solution Note that this is a special case of Exercise 7.4-6 where $\alpha = 1/3$. The probability is evaluated as

$$P(n/3 \le i \le 2n/3) = 1 - 6(1/3)^2 + 4(1/3)^3 = 13/27.$$

In the ordinary implementation, this probability is $1/3$, therefore, the likelihood increased is

$$\left(\frac{13/27}{1/3} - 1\right) \times 100\% = 44.44\%.$$

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

Solution No matter how the pivot is chosen, each call of the partition procedure takes $\Omega(n)$ running because of the **for** loop inside. Besides, the depth of the recursion tree is $\Omega(\lg n)$ even in the best case where $q \approx (p+r)/2$ is always chosen. Hence, the lower bound $\Omega(n \lg n)$ is not affected by the median-of-3 method but the constant factor.

7. **Fuzzy sorting of intervals**

Consider a sorting algorithm in which you do not know the numbers exactly. Instead, for each number, you know an interval on the real line to which it belongs. That is, you 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: 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.)

Solution In this solution, each x in A is an interval, and we will use $x.a$ and $a.b$ to denote the left endpoint of x and the right endpoint of x, respectively. Note that the main idea is similar to that in Problem 7.2. First, the program FUZZY-SORT takes two indices from PARTITION, and makes two calls to itself:

Algorithm 12 FUZZY-SORT(A, p, r)

```
1: if p < r then
2:    q, t = MODIFIED-PARTITION(A, p, r)
3:    FUZZY-SORT(A, p, q)
4:    FUZZY-SORT(A, t, r)
5: end if
```

Where, the partition procedure is modified so that if an interval overlaps with the pivot, we treat them as the same element:

Algorithm 13 MODIFIED-PARTITION(A, p, r)

```
1: d = RANDOM(p, r)
2: exchange A[d] with A[r]
3: x = A[r]
4: i = p − 1
5: for j = p to r − 1 do
6:     if A[j].a ≤ x.a then
7:         i = i + 1
8:         exchange A[i] with A[j]
9:     end if
10: end for
11: exchange A[i + 1] with A[r]
12: j = i + 2
13: while i ≥ p and A[i].b ≥ x.a do
14:     i = i − 1
15: end while
16: k = p
17: while k < i do
18:     if A[k].b ≥ x.a then
19:         exchange A[k] with A[i]
20:         while i ≥ p and A[i].b ≥ x.a do
21:             i = i − 1
22:         end while
23:     end if
24:     k = k + 1
25: end while
26: while j ≤ r and A[j].a ≤ x.b do
27:     j = j + 1
28: end while
29: l = r
30: while l > j do
31:     if A[l].a ≤ x.b then
32:         exchange A[l] with A[j]
33:         while j ≤ r and A[j].a ≤ x.b do
34:             j = j + 1
35:         end while
36:     end if
37:     l = l − 1
38: end while
39: return i + 1, j − 1
```

b. Argue that your algorithm runs in $\Theta(n \lg n)$ expected time in general, but runs in $\Theta(n)$ expected time 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, but rather, its performance should naturally improve as the amount of overlap increases.

Solution In average case, we do not expect the intervals to overlap each other, so the expect running time is still $\Theta(n \lg n)$ (to see the running time of the modified partition procedure is still $\Theta(n)$, see Problem 7.2).

When all the intervals overlap and we call FUZZY-SORT$(A, 1, n)$, the modified partition returns 1 and n and the program terminates after calling FUZZY-SORT$(A, 1, 1)$ and FUZZY-SORT(A, n, n) in $\Theta(1)$ time. The total running time is therefore just the running time of MODIFIED-PARTITION$(A, 1, n)$, which is $\Theta(n)$.