7 Quicksort

7.1 Description of quicksort

7.1-1

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

$$
\begin{gathered}
\langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle \\
\langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle \\
\langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle \\
\langle 9, 19, 13, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle \\
\langle 9, 5, 13, 19, 12, 8, 7, 4, 21, 2, 6, 11 \rangle \\
\langle 9, 5, 13, 19, 12, 8, 7, 4, 21, 2, 6, 11 \rangle \\
\langle 9, 5, 8, 19, 12, 13, 7, 4, 21, 2, 6, 11 \rangle \\
\langle 9, 5, 8, 7, 12, 13, 19, 4, 21, 2, 6, 11 \rangle \\
\langle 9, 5, 8, 7, 4, 13, 19, 12, 21, 2, 6, 11 \rangle \\
\langle 9, 5, 8, 7, 4, 13, 19, 12, 21, 2, 6, 11 \rangle \\
\langle 9, 5, 8, 7, 4, 2, 19, 12, 21, 13, 6, 11 \rangle \\
\langle 9, 5, 8, 7, 4, 2, 6, 12, 21, 13, 19, 11 \rangle \\
\langle 9, 5, 8, 7, 4, 2, 6, 11, 21, 13, 19, 12 \rangle
\end{gathered}
$$

7.1-2

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

It returns r.

We can modify PARTITION by counting the number of comparisons in which $A[j] = A[r]$ and then subtracting half that number from the pivot index.

7.1-3

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

There is a for statement whose body executes $r - 1 - p = \Theta(n)$ times. In the worst case every time the body of the if is executed, but it takes constant time and so does the code outside of the loop. Thus the running time is $\Theta(n)$.

7.1-4

> How would you modify QUICKSORT to sort into nonincreasing order?

We only need to flip the condition on line 4.

7.2 Performance of quicksort

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.

We represent $\Theta(n)$ as $c_2 n$ and we guess that $T(n) \le c_1 n^2$,

$$\begin{aligned} T(n) &= T(n-1) + c_2 n \\ &\le c_1(n-1)^2 + c_2 n \\ &= c_1 n^2 - 2c_1 n + c_1 + c_2 n && (2c_1 > c_2, n \ge c_1/(2c_1 - c_2)) \\ &\le c_1 n^2. \end{aligned}$$

7.2-2

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

It is $\Theta(n^2)$, since one of the partitions is always empty (see exercise 7.1-2.)

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.

If the array is already sorted in decreasing order, then, the pivot element is less than all the other elements. The partition step takes $\Theta(n)$ time, and then leaves you with a subproblem of size $n-1$ and a subproblem of size 0. This gives us the recurrence considered in 7.2-1. Which we showed has a solution that is $\Theta(n^2)$.

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 numbers. People usually write checks in order by check number, and merchants usually cash the 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.

The more sorted the array is, the less work insertion sort will do. Namely, INSERTION-SORT is $\Theta(n+d)$, where d is the number of inversions in the array. In the example above the number of inversions tends to be small so insertion sort will be close to linear.

On the other hand, if PARTITION does pick a pivot that does not participate in an inversion, it will produce an empty partition. Since there is a small number of inversions, QUICKSORT is very likely to produce empty partitions.

7.2-5

> Suppose that the splits at every level of quicksort are in proportion $1-\alpha$ to α, where $0 < \alpha \le 1/2$ 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.)

The minimum depth corresponds to repeatedly taking the smaller subproblem, that is, the branch whose size is proportional to α. Then, this will fall to 1 in k steps where $1 \approx \alpha^k n$. Therefore, $k \approx \log_\alpha 1/n = -\frac{\lg n}{\lg\alpha}$. The longest depth corresponds to always taking the larger subproblem. we then have an identical expression, replacing α with $1-\alpha$.

7.2-6 ⋆

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

In order to produce a worse split than $1-\alpha$ to α, PARTITION must pick a pivot that will be either within the smallest αn elements or the largest αn elements. The probability of either is (approximately) $\alpha n/n = \alpha$ and the probability of both is 2α. Thus, the probability of having a better partition is the complement, $1-2\alpha$.

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 analyze the expected run time because it represents the more typical time cost. Also, we are doing the expected run time over the possible randomness used during computation because it can't be produced adversarially, unlike when doing expected run time over all possible inputs to the algorithm.

7.3-2

> When $\text{RANDOMIZED-QUICKSORT}$ 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.

In the worst case, the number of calls to RANDOM is

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

As for the best case,

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

This is not too surprising, because each third element (at least) gets picked as pivot.

7.4 Analysis of quicksort

7.4-1

> Show that in the recurrence
>
> $$\begin{aligned} T(n) &= \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n), \\ T(n) &= \Omega(n^2). \end{aligned}$$

We guess $T(n) \ge cn^2 - 2n$,

$$\begin{aligned} T(n) &= \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) \\ &\ge \max_{0 \le q \le n-1} (cq^2 - 2q + c(n-q-1)^2 - 2n - 2q - 1) + \Theta(n) \\ &\ge c \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2 - (2n + 4q + 1)/c) + \Theta(n) \\ &\ge cn^2 - c(2n-1) + \Theta(n) \\ &\ge cn^2 - 2cn + 2c & (c \le 1) \\ &\ge cn^2 - 2n. \end{aligned}$$

7.4-2

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

We'll use the substitution method to show that the best-case running time is $\Omega(n \lg n)$. Let $T(n)$ be the best-case time for the procedure QUICKSORT on an input of size n. We have

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

Suppose that $T(n) \ge c(n \lg n + 2n)$ for some constant c. Substituting this guess into the recurrence gives

$$\begin{aligned} T(n) &\ge \min_{1 \le q \le n-1} (cq \lg q + 2cq + c(n-q-1) \lg(n-q-1) + 2c(n-q-1)) + \Theta(n) \\ &= (cn/2) \lg(n/2) + cn + c(n/2-1) \lg(n/2-1) + cn - 2c + \Theta(n) \\ &\ge (cn/2) \lg n - cn/2 + c(n/2-1)(\lg n - 2) + 2cn - 2c\Theta(n) \\ &= (cn/2) \lg n - cn/2 + (cn/2) \lg n - cn - c \lg n + 2c + 2cn - 2c\Theta(n) \\ &= cn \lg n + cn/2 - c \lg n + 2c - 2c\Theta(n). \end{aligned}$$

Taking a derivative with respect to q shows that the minimum is obtained when $q = n/2$. Taking c large enough to dominate the $-\lg n + 2 - 2c + \Theta(n)$ term makes this greater than $cn \lg n$, proving the bound.

7.4-3

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

$$\begin{aligned} f(q) &= q^2 + (n-q-1)^2 \\ f'(q) &= 2q - 2(n-q-1) = 4q - 2n + 2 \\ f''(q) &= 4. \end{aligned}$$

$f'(q) = 0$ when $q = \frac{1}{2}n - \frac{1}{2}$. $f'(q)$ is also continious. $\forall q : f''(q) > 0$, which means that $f'(q)$ is negative left of $f'(q) = 0$ and positive right of it, which means that this is a local minima. In this case, $f(q)$ is decreasing in the beginning of the interval and increasing in the end, which means that those two points are the only candidates for a maximum in the interval.

$$\begin{aligned} f(0) &= (n-1)^2 \\ f(n-1) &= (n-1)^2 + 0^2. \end{aligned}$$

7.4-4

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

We use the same reasoning for the expected number of comparisons, we just take in a different direction.

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

Using the master method, we get the solution $\Theta(n \lg n)$.

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 practice?

In the quicksort part of the proposed algorithm, the recursion stops at level $\lg(n/k)$, which makes the expected running time $O(n \lg(n/k))$. However, this leaves n/k non-sorted, non - intersecting subarrays of (maximum) length k.

Because of the nature of the insertion sort algorithm, it will first sort fully one such subarray before consider the next one. Thus, it has the same complexity as sorting each of those arrays, that is $\frac{n}{k} O(k^2) = O(nk)$.

In theory, if we ignore the constant factors, we need to solve

$$\begin{aligned} & n \lg n \ge nk + n \lg n/k \\ \Rightarrow & \lg n \ge k + \lg n - \lg k \\ \Rightarrow & \lg k \ge k. \end{aligned}$$

Which is not possible.

If we add the constant factors, we get

$$\begin{aligned} & c_q n \lg n \geq c_i nk + c_q n \lg(n/k) \\ \Rightarrow & \ c_q \lg n \geq c_i k + c_q \lg n - c_q \lg k \\ \Rightarrow & \lg k \geq \frac{c_i}{c_q} k. \end{aligned}$$

Which indicates that there might be a good candidate. Furthermore, the lower-order terms should be taken into consideration too.

In practice, k should be chosed by experiment.

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 $0 < \alpha < 1$.

First, for simplicity's sake, let's assume that we can pick the same element twice. Let's also assume that $0 < \alpha \leq 1/2$.

In order to get such a split, two out of three elements need to be in the smallest αn elements. The probability of having one is $\alpha n/n = \alpha$. The probability of having exactly two is $\alpha^2 - \alpha^3$. There are three ways in which two elements can be in the smallest αn and one way in which all three can be in the smallest αn so the probability of getting such a median is $3\alpha^2 - 2\alpha^3$. We will get the same split if the median is in the largest αn. Since the two events are mutually exclusive, the probability is

$$\Pr\{\text{OK split}\} = 6\alpha^2 - 4\alpha^3 = 2\alpha^2(3 - 2\alpha).$$

Problem 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)
>     x = A[p]
>     i = p - 1
>     j = r + 1
>     while true
>         repeat
>             j = j - 1
>         until A[j] ≤ x
>         repeat
>             i = i + 1
>         until A[i] ≥ x
>         if i < j
>             exchange A[i] with A[j]
>         else return j
> ```
>
> **a.** Demonstrate the operation of HOARE-PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, showing the values of the array and auxiliary values after each iteration of the **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 $A[p..r]$ 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 $A[p..r]$.
>
> **c.** When HOARE-PARTITION terminates, it returns a value j such that $p \leq j < r$.

> **d.** Every element of $A[p..j]$ is less than or equal to every element of $A[j+1..r]$ when HOARE-PARTITION terminates.
>
> The PARTITION procedure in section 7.1 separates the pivot value (originally in $A[r]$) from the two partitions it forms. The HOARE-PARTITION procedure, on the other hand, always places the pivot value (originally in $A[p]$) into one of the two parititions $A[p..j]$ and $A[j+1..r]$. Since $p \le j < r$, this split is always nontrivial.
>
> **e.** Rewrite the QUICKSORT procedure to use HOARE-PARTITION.

a. After the end of the loop, the variables have the following values: $x = 13$, $j = 9$ and $i = 10$.

b. Because when HOARE-PARTITION is running, $p \le i < j \le r$ will always hold, i, j won't access any element of A outside the subarray $A[p..r]$.

c. When $i \ge j$, HOARE-PARTITION terminates, so $p \le j < r$.

d. When HOARE-PARTITION terminates, $A[p..j] \le x \le A[j+1..r]$.

e.

```
HOARE-QUICKSORT(A, p, r)
    if p < r
        q = HOARE-PARTITION(A, p, r)
        HOARE-QUICKSORT(A, p, q)
        HOARE-QUICKSORT(A, q + 1, r)
```

Problem 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 quick-sort's running time in this case?
>
> **b.** The PARTITION procedure returns an index q 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 $\text{PARTITION}'(A, p, r)$ which permutes the elements of $A[p..r]$ 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 $\text{PARTITION}'$ procedure should take $\Theta(r-p)$ time.
>
> **c.** Modify the RANDOMIZED-QUICKSORT procedure to call $\text{PARTITION}'$, and name the new procedure $\text{RANDOMIZED-QUICKSORT}'$. Then modify the QUICKSORT procedure to produce a procedure $\text{QUICKSORT}'(p, r)$ that calls $\text{RANDOMIZED-PARTITION}'$ and recurses only on partitions of elements not know to be equal to each other.
>
> **d.** Using $\text{QUICKSORT}'$, how would you adjust the analysis of section 7.4.2 to avoid the assumption that all elements are distinct?

a. Since all elements are equal, RANDOMIZED-QUICKSORT always returns $q = r$. We have recurrence $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.

b.

```
PARTITION'(A, p, r)
    x = A[p]
    low = p
    high = p
    for j = p + 1 to r
        if A[j] < x
            y = A[j]
            A[j] = A[high + 1]
            A[high + 1] = A[low]
            A[low] = y
            low = low + 1
            high = high + 1
        else if A[j] == x
            exchange A[high + 1] with A[j]
            high = high + 1
    return (low, high)
```

c.

```
QUICKSORT'(A, p, r)
    if p < r
        (low, high) = RANDOMIZED-PARTITION'(A, p, r)
        QUICKSORT'(A, p, low - 1)
        QUICKSORT'(A, high + 1, r)
```

d. Since we don't recurse on elements equal to the pivot, the subproblem sizes with $\text{QUICKSORT}'$ are no larger than the subproblem sizes with QUICKSORT when all elements are distinct.

Problem 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 $\text{RANDOMIZED-QUICKSORT}$, 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
>
> $$X_i = I\{i\text{th smallest element is chosen as the pivot}\}.$$
>
> What is $E[X_i]$?
>
> **b.** Let $T(n)$ be a random variable denoting the running time of quicksort on an array of size n. Argue that
>
> $$E[T(n)] = E\left[\sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))\right]. \tag{7.5}$$
>
> **c.** Show that we can rewrite equation (7.5) as
>
> $$E[T(n)] = \frac{2}{n}\sum_{q=2}^{n-1} E[T(q)] + \Theta(n). \tag{7.6}$$
>
> **d.** Show that
>
> $$\sum_{k=2}^{n-1} k \lg k \le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2. \tag{7.7}$$
>
> $($Hint: Split the summation into two parts, one for $k = 2, 3, \ldots, \lceil n/2 \rceil - 1$ and one for $k = \lceil n/2 \rceil, \ldots, n-1$.)
>
> **e.** Using the bound from equation (7.7), show that the recurrence in equation (7.6) has the solution $E[T(n)] = \Theta(n \lg n)$. $($Hint: Show, by substitution, that $E[T(n)] \le an \lg n$ for sufficiently large n and for some positive constant a.)

a. Since the pivot is selected as a random element in the array, which has size n, the probabilities of any particular element being selected are all equal, and add to one, so, are all $\frac{1}{n}$. As such, $E[X_i] = \Pr\{i \text{ smallest is picked}\} = \frac{1}{n}$.

b. We can apply linearity of expectation over all of the events X_i. Suppose we have a particular X_i be true, then, we will have one of the sub arrays be length $i - 1$, and the other be $n - i$, and will of course still need linear time to run the partition procedure. This corresponds exactly to the summand in equation (7.5).

c.

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

d. We will prove this inequality in a different way than suggested by the hint. If we let $f(k) = k \lg k$ treated as a continuous function, then $f'(k) = \lg k + 1$. Note now that the summation written out is the left hand approximation of the integral of $f(k)$ from 2 to n with step size 1. By integration by parts, the anti-derivative of $k \lg k$ is

$$\frac{1}{\lg 2}\left(\frac{k^2}{2}\ln k - \frac{k^2}{4}\right).$$

So, plugging in the bounds and subtracting, we get $\frac{n^2 \lg n}{2} - \frac{n^2}{4 \ln 2} - 1$. Since f has a positive derivative over the entire interval that the integral is being evaluated over, the left hand rule provides a underapproximation of the integral, so, we have that

$$
\begin{aligned}
\sum_{k=2}^{n-1} k \lg k &\le \frac{n^2 \lg n}{2} - \frac{n^2}{4 \ln 2} - 1 \\
&\le \frac{n^2 \lg n}{2} - \frac{n^2}{8},
\end{aligned}
$$

where the last inequality uses the fact that $\ln 2 > 1/2$.

e. Assume by induction that $T(q) \le q \lg(q) + \Theta(n)$. Combining (7.6) and (7.7), we have

$$
\begin{aligned}
E[T(n)] &= \frac{2}{n}\sum_{q=2}^{n-1} E[T(q)] + \Theta(n) \\
&\le \frac{2}{n}\sum_{q=2}^{n-1}(q\lg q + \Theta(n)) + \Theta(n) \\
&\le \frac{2}{n}\sum_{q=2}^{n-1} q\lg q + \frac{2}{n}\Theta(n) + \Theta(n) \\
&\le \frac{2}{n}(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2) + \Theta(n) \\
&= n\lg n - \frac{1}{4}n + \Theta(n) \\
&= n\lg n + \Theta(n).
\end{aligned}
$$

Problem 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)
    while p < r
        // Partition and sort left subarray.
        q = PARTITION(A, p, r)
        TAIL-RECURSIVE-QUICKSORT(A, p, q - 1)
        p = q + 1
```

a. Argue that $\text{TAIL-RECURSIVE-QUICKSORT}(A, 1, A.length)$ 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 $\text{TAIL-RECURSIVE-QUICKSORT}$'s stack depth is $\Theta(n)$ on an n-element input array.

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

a. The book proved that QUICKSORT correctly sorts the array A. $\text{TAIL-RECURSIVE-QUICKSORT}$ differs from QUICKSORT in only the last line of the loop.

It is clear that the conditions starting the second iteration of the **while** loop in $\text{TAIL-RECURSIVE-QUICKSORT}$ are identical to the conditions starting the second recursive call in QUICKSORT. Therefore, $\text{TAIL-RECURSIVE-QUICKSORT}$ effectively performs the sort in the same manner as QUICKSORT. Therefore, $\text{TAIL-RECURSIVE-QUICKSORT}$ must correctly sort the array A.

b. The stack depth will be $\Theta(n)$ if the input array is already sorted. The right subarray will always have size 0 so there will be $n - 1$ recursive calls before the **while**-condition $p < r$ is violated.

c.

```
MODIFIED-TAIL-RECURSIVE-QUICKSORT(A, p, r)
    while p < r
        q = PARTITION(A, p, r)
        if q < floor((p + r) / 2)
            MODIFIED-TAIL-RECURSIVE-QUICKSORT(A, p, q - 1)
            p = q + 1
        else
            MODIFIED-TAIL-RECURSIVE-QUICKSORT(A, q + 1, r)
            r = q - 1
```

Problem 7-5 Median-of-3 partition

One way to improve the $\text{RANDOMIZED-QUICKSORT}$ 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 of the input array $A[1..n]$ are distinct and that $n \ge 3$. We denote the sorted output array by $A'[1..n]$. 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 $i = 2, 3, \dots, n-1$. (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 $A[1..n]$, 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 $n/3 \le i \le 2n/3$, 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. p_i is the probability that a randomly selected subset of size three has the $A'[i]$ as it's middle element. There are 6 possible orderings of the three elements selected. So, suppose that S' is the set of three elements selected.

We will compute the probability that the second element of S' is $A'[i]$ among all possible 3-sets we can pick, since there are exactly six ordered 3-sets corresponding to each 3-set, these probabilities will be equal. We will compute the probability that $S'[2] = A[i]$. For any such S', we would need to select the first element from $[i-1]$ and the third from $i+1, \dots, n$. So, there are $(i-1)(n-i)$ such 3-sets. The total number of 3-sets is $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$. So,

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

b. If we let $i = \lfloor \frac{n+1}{2} \rfloor$, the previous result gets us an increase of

$$\frac{6(\lfloor \frac{n-1}{2} \rfloor)(n - \lfloor \frac{n+1}{2} \rfloor)}{n(n-1)(n-2)} - \frac{1}{n}$$

in the limit n going to infinity, we get

$$\lim_{n \to \infty} \frac{\frac{6(\lfloor \frac{n-1}{2} \rfloor)(n-\lfloor \frac{n+1}{2} \rfloor)}{n(n-1)(n-2)}}{\frac{1}{n}} = \frac{3}{2}.$$

c. To save the messiness, suppose n is a multiple of 3. We will approximate the sum as an integral, so,

$$\begin{aligned}\sum_{i=n/3}^{2n/3} &\approx \int_{n/3}^{2n/3} \frac{6(-x^2+nx+x-n)}{n(n-1)(n-2)}dx \\ &= \frac{6(-7n^3/81+3n^3/18+3n^2/18-n^2/3)}{n(n-1)(n-2)},\end{aligned}$$

which, in the limit n goes to infinity, is $\frac{13}{27}$ which is a constant that $> \frac{1}{3}$ as it was in the original randomized quicksort implementation.

d. Even though we always choose the middle element as the pivot (which is the best case), the height of the recursion tree will be $\Theta(\lg n)$. Therefore, the running time is still $\Omega(n \lg n)$.

Problem 7-6 Fuzzy sorting of intervals

> Consider the 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 $a_i \le b_i$. 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 exists $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \le c_2 \le \cdots \le c_n$.
>
> **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 becoes progressively easier. Your algorithm should take advantage of such overlapping, to the extend that it exists.)
>
> **b.** Argue that your algorithm runs in expected time $\Theta(n \lg n)$ in general, but runs in expected time $\Theta(n)$ when all of the intervals overlap (i.e., when there exists a value x such that $x \in [a_i, b_i]$ for all i). Your algorithm should not be checking for this case explicitly; rather, its performance should naturally improve as the amount of overlap increases.

a. With randomly selected left endpoint for the pivot, we could trivially perform fuzzy sorting by quicksorting the left endpoints, a_i's. This would achieve the worst-case expected running time of $\Theta(n \lg n)$. We definitely can do better by exploit the characteristic that we don't have to sort overlapping intervals. That is, for two overlapping intervals, $[a_i, b_i]$ and $[a_j, b_j]$. In such situations, we can always choose $\{c_i, c_j\}$ (within the intersection of these intervals) such that $c_i \le c_j$ or $c_j \le c_i$.

Since overlapping intervals do not require sorting, we can improve the expected running time by modifying quicksort to identify overlaps:

```
FIND-INTERSECTION(A, p, r)
    rand = RANDOM(p, r)
    exchange A[rand] with A[r]
    a = A[r].a
    b = A[r].b
    for i = p to r - 1
        if A[i].a ≤ b and A[i].b ≥ a
            if A[i].a > a
                a = A[i].a
            if A[i].b < b
                b = A[i].b
    return (a, b)
```

On lines 2 through 3 of FIND-INTERSECTION, we select a random *pivot interval* as the initial region of overlap $[a, b]$. There are two situations:

- If the intervals are all disjoint, then the estimated region of overlap will be this randomly-selected interval;
- otherwise, on lines 6 through 11, we loop through all intervals in arrays A (except the endpoint which is the initial pivot interval). At each iteration, we determine if the current interval overlaps the current estimated region of overlap. If it does, we update the estimated region of overlap as $[a, b] = [a_i, b_i] \cap [a, b]$.

FIND-INTERSECTION has a worst-case running time $\Theta(n)$ since we evaluate the intersection from index 1 to $A.\text{length}$ of the array.

We can extend the QUICKSORT to allow fuzzy sorting using FIND-INTERSECTION.

First, partition the input array into "left", "middle", and "right" subarrays. The "middle" subarray elements overlap the interval $[a, b]$ found by FIND-INTERSECTION. As a result, they can appear in any order in the output.

We recursively call FUZZY-SORT on the "left" and "right" subarrays to produce a fuzzy sorted array in-place. The following pseudocode implements these basic operations. One can run $\text{FUZZY-SORT}(A, 1, A.\text{length})$ to fuzzy-sort an array.

The first and last elements in a subarray are indexed by p and r, respectively. The index of the first and last intervals in the "middle" region are indexed by q and t, respectively.

```
FUZZY-SORT(A, p, r)
    if p < r
        (a, b) = FIND-INTERSECTION(A, p, r)
        t = PARTITION-RIGHT(A, a, p, r)
        q = PARTITION-LEFT(A, b, p, t)
        FUZZY-SORT(A, p, q - 1)
        FUZZY-SORT(A, t + 1, r)
```

We need to determine how to partition the input arrays into "left", "middle", and "right" subarrays in-place.

First, we PARTITION-RIGHT the entire array from p to r using a pivot value equal to the left endpoint a found by FIND-INTERSECTION, such that $a_i \le a$.

Then, we PARTITION-LEFT the subarray from p to t using a pivot value equal to the right endpoint b found by FIND-INTERSECTION, such that $b_i < b$.

```
PARTITION-RIGHT(A, a, p, r)
    i = p - 1
    for j = p to r - 1
        if A[j].a ≤ a
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

```
PARTITION-LEFT(A, b, p, t)
    i = p - 1
    for j = p to t - 1
        if A[j].b < b
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[t]
    return i + 1
```

The FUZZY-SORT is similar to the randomized quicksort presented in the textbook. In fact, PARTITION-RIGHT and PARTITION-LEFT are nearly identical to the PARTITION procedure on page 171. The primary difference is the value of the pivot used to sort the intervals.

b. We expect FUZZY-SORT to have a worst-case running time $\Theta(n \lg n)$ for a set of input intervals which do not overlap each other. First, notice that lines 2 through 3 of FIND-INTERSECTION select a *random interval* as the initial pivot interval. Recall that if the intervals are disjoint, then $[a, b]$ will simply be this initial interval.

Since for this example there are no overlaps, the "middle" region created by lines 4 and 5 of FUZZY-SORT will only contain the initially-selected interval. In general, line 3 is $\Theta(n)$. Fortunately, since the pivot interval $[a, b]$ is randomly-selected, the expected sizes of the "left" and "right" subarrays are both $\left\lfloor \frac{n}{2} \right\rfloor$. In conclusion, the reccurrence of the running time is

$$\begin{aligned} T(n) &\le 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n). \end{aligned}$$

The FIND-INTERSECTION will always return a non-empty region of overlap $[a, b]$ containing x if the intervals all overlap at x. For this situation, every interval will be within the "middle" region since the "left" and "right" subarrays will be empty, lines 6 and 7 of FUZZY-SORT are $\Theta(1)$. As a result, there is no recursion and the running time of FUZZY-SORT is determined by the $\Theta(n)$ running time required to find the region of overlap. Therfore, if the input intervals all overlap at a point, then the expected worst-case running time is $\Theta(n)$.