9 Medians and Order Statistics

9.1 Minimum and maximum

9.1-1

> Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. (Hint: Also find the smallest element.)

We can compare the elements in a tournament fashion - we split them into pairs, compare each pair and then proceed to compare the winners in the same fashion. We need to keep track of each "match" the potential winners have participated in.

We select a winner in $n - 1$ matches. At this point, we know that the second smallest element is one of the lgn elements that lost to the smallest - each of them is smaller than the ones it has been compared to, prior to losing. In another $\lceil \lg n \rceil - 1$ comparisons we can find the smallest element out of those.

9.1-2 ⋆

> Prove the lower bound of $\lceil 3n/2 \rceil - 2$ comparisons in the worst case to find both the maximum and minimum of n numbers. (Hint: Consider how many numbers are potentially either the maximum or minimum, and investigate how a comparison affects these counts.)

If n is odd, there are

$$\begin{aligned} 1 + \frac{3(n-3)}{2} + 2 &= \frac{3n}{2} - \frac{3}{2} \\ &= \left(\left\lceil \frac{3n}{2} \right\rceil - \frac{1}{2}\right) - \frac{3}{2} \\ &= \left\lceil \frac{3n}{2} \right\rceil - 2 \end{aligned}$$

comparisons.

If n is even, there are

$$\begin{aligned} 1 + \frac{3(n-2)}{2} &= \frac{3n}{2} - 2 \\ &= \left\lceil \frac{3n}{2} \right\rceil - 2 \end{aligned}$$

comparisons.

9.2 Selection in expected linear time

9.2-1

> Show that RANDOMIZED-SELECT never makes a recursive call to a 0-length array.

Calling a 0-length array would mean that the second and third arguments are equal. So, if the call is made on line 8, we would need that $p = q - 1$, which means that $q - p + 1 = 0$.

However, i is assumed to be a nonnegative number, and to be executing line 8, we would need that $i < k = q - p + 1 = 0$, a contradiction. The other possibility is that the bad recursive call occurs on line 9. This would mean that $q + 1 = r$. To be executing line 9, we need that $i > k = q - p + 1 = r - p$. This would be a nonsensical original call to the array though because we are asking for the ith element from an array of strictly less size.

9.2-2

> Argue that the indicator random variable X_k and the value $T(\max(k-1, n-k))$ are independent.

The probability that X_k is equal to 1 is unchanged when we know the max of $k-1$ and $n-k$. In other words, $\Pr\{X_k = a \mid \max(k-1, n-k) = m\} = \Pr\{X_k = a\}$ for $a = 0, 1$ and $m = k-1, n-k$ so X_k and $\max(k-1, n-k)$ are independent.

By C.3-5, so are X_k and $T(\max(k-1, n-k))$.

9.2-3

> Write an iterative version of RANDOMIZED-SELECT.

```
PARTITION(A, p, r)
    x = A[r]
    i = p
    for k = p - 1 to r
       if A[k] < x
           i = i + 1
           swap A[i] with A[k]
    i = i + 1
    swap A[i] with A[r]
    return i
```

```
RANDOMIZED-PARTITION(A, p, r)
    x = RANDOM(p - 1, r)
    swap A[x] with A[r]
    return PARTITION(A, p, r)
```

```
RANDOMIZED-SELECT(A, p, r, i)
    while true
        if p == r
            return A[p]
        q = RANDOMIZED-PARTITION(A, p, r)
        k = q - p + 1
        if i == k
            return A[q]
        if i < k
            r = q - 1
        else
            p = q + 1
            i = i - k
```

9.2-4

> Suppose we use RANDOMIZED-SELECT to select the minimum element of the array $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

When the partition selected is always the maximum element of the array we get worst-case performance. In the example, the sequence would be $\langle 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 \rangle$.

9.3 Selection in worst-case linear time

9.3-1

> In the algorithm SELECT, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7? Argue that SELECT does not run in linear time if groups of 3 are used.

It will still work if they are divided into groups of 7, because we will still know that the median of medians is less than at least 4 elements from half of the $\lceil n/7 \rceil$ groups, so, it is greater than roughly $4n/14$ of the elements.

Similarly, it is less than roughly $4n/14$ of the elements. So, we are never calling it recursively on more than $10n/14$ elements. $T(n) \leq T(n/7) + T(10n/14) + O(n)$. So, we can show by substitution this is linear.

We guess $T(n) < cn$ for $n < k$. Then, for $m \geq k$,

$$\begin{aligned} T(m) &\leq T(m/7) + T(10m/14) + O(m) \\ &\leq cm(1/7 + 10/14) + O(m), \end{aligned}$$

therefore, as long as we have that the constant hidden in the big-Oh notation is less than $c/7$, we have the desired result.

Suppose now that we use groups of size 3 instead. So, For similar reasons, we have that the recurrence we are able to get is $T(n) = T(\lceil n/3 \rceil) + T(4n/6) + O(n) \geq T(n/3) + T(2n/3) + O(n)$ So, we will show it is $\geq cn \lg n$.

$$\begin{aligned} T(m) &\geq c(m/3)\lg(m/3) + c(2m/3)\lg(2m/3) + O(m) \\ &\geq cm \lg m + O(m), \end{aligned}$$

therefore, we have that it grows more quickly than linear.

9.3-2

> Analyze SELECT to show that if $n \geq 140$, then at least $\lceil n/4 \rceil$ elements are greater than the median-of-medians x and at least $\lceil n/4 \rceil$ elements are less than x.

$$\begin{aligned} \frac{3n}{10} - 6 &\geq \lceil \frac{n}{4} \rceil \\ \frac{3n}{10} - 6 &\geq \frac{n}{4} + 1 \\ 12n - 240 &\geq 10n + 40 \\ n &\geq 140. \end{aligned}$$

9.3-3

> Show how quicksort can be made to run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

We can modify quicksort to run in worst case $n \lg n$ time by choosing our pivot element to be the exact median by using quick select. Then, we are guaranteed that our pivot will be good, and the time taken to find the median is on the same order of the rest of the partitioning.

9.3-4 ⋆

> Suppose that an algorithm uses only comparisons to find the ith smallest element in a set of n elements. Show that it can also find the $i - 1$ smaller elements and $n - i$ larger elements without performing additional comparisons.

Create a graph with n vertices and draw a directed edge from vertex i to vertex j if the ith and jth elements of the array are compared in the algorithm and we discover that $A[i] \geq A[j]$. Observe that $A[i]$ is one of the $i - 1$ smaller elements if there exists a path from x to i in the graph, and $A[i]$ is one of the $n - i$ larger elements if there exists a path from i to x in the graph. Every vertex i must either lie on a path to or from x because otherwise the algorithm can't distinguish between $i \leq x$ and $i \geq x$. Moreover, if a vertex i lies on both a path to x and a path from x, then it must be such that $x \leq A[i] \leq x$,so $x = A[i]$. In this case, we can break ties arbitrarily.

9.3-5

> Suppose that you have a "black-box" worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

To use it, just find the median, partition the array based on that median.

- If i is less than half the length of the original array, recurse on the first half.
- If i is half the length of the array, return the element coming from the median finding black box.
- If i is more than half the length of the array, subtract half the length of the array, and then recurse on the second half of the array.

9.3-6

> The kth ***quantiles*** of an n-element set are the $k-1$ order statistics that divide the sorted set into k equal-sized sets (to within 1). Give an $O(n \lg k)$-time algorithm to list the kth quantiles of a set.

Pre-calculate the positions of the quantiles in $O(k)$, we use the $O(n)$ select algorithm to find the $\lfloor k/2 \rfloor$th position, after that the elements are divided into two sets by the pivot the $\lfloor k/2 \rfloor$th position, we do it recursively in the two sets to find other positions. Since the maximum depth is $\lceil \lg k \rceil$, the total running time is $O(n \lg k)$.

```
PARTITION(A, p, r)
    x = A[r]
    i = p
    for k = p to r
        if A[k] < x
            i = i + 1
            swap A[i] with A[k]
    i = i + 1
    swap a[i] with a[r]
    return i
```

```
RANDOMIZED-PARTITION(A, p, r)
    x = RANDOM(p, r)
    swap A[x] with A[r]
    return PARTITION(A, p, r)
```

```
RANDOMIZED-SELECT(A, p, r, i)
    while true
        if p == r
            return p, A[p]
        q = RANDOMIZED-PARTITION(A, p, r)
        k = q - p + 1
        if i == k
            return q, A[q]
        if i < k
            r = q
        else
            p = q + 1
            i = i - k
```

```
k-QUANTITLES-SUB(A, p, r, pos, f, e, quantiles)
    if f + 1 > e
        return
    mid = (f + e) / 2
    q, val = RANDOMIZED-SELECT(A, p, r, pos[mid])
    quantiles[mid] = val
    k = q - p + 1
    for i = mid + 1 to e
```

```
        pos[i] = pos[i] - k
    k-QUANTILES-SUB(A, q + 1, r, pos, mid + 1, e, quantiles)
```

```
k-QUANTITLES(A, k)
    num = A.size() / k
    mod = A.size() % k
    pos = num[1..k]
    for i = 1 to mod
        pos[i] = pos[i] + 1
    for i = 1 to k
        pos[i] = pos[i] + pos[i - 1]
    quantiles = [1..k]
    k-QUANTITLES-SUB(A, 0, A.length, pos, 0, pos.size(), quantiles)
    return quantiles
```

9.3-7

> Describe an $O(n)$-time algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S.

Find the median in $O(n)$; create a new array, each element is the absolute value of the original value subtract the median; find the kth smallest number in $O(n)$, then the desired values are the elements whose absolute difference with the median is less than or equal to the kth smallest number in the new array.

9.3-8

> Let $X[1..n]$ and $Y[1..n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays X and Y.

Without loss of generality, assume n is a power of 2.

```
MEDIAN(X, Y, n)
    if n == 1
        return min(X[1], Y[1])
    if X[n / 2] < Y[n / 2]
        return MEDIAN(X[n / 2 + 1..n], Y[1..n / 2], n / 2)
    return MEDIAN(X[1..n / 2], Y[n / 2 + 1..n], n / 2)
```

9.3-9

> Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. The company wants to connect a spur pipeline from each well directly to the main pipeline along a shortest route (either north or south), as shown in Figure 9.2. Given the x- and y-coordinates of the wells, how should the professor pick the optimal location of the main pipeline, which would be the one that minimizes the total length of the spurs? Show how to determine the optimal location in linear time.

- If n is odd, we pick the y coordinate of the main pipeline to be equal to the median of all the y coordinates of the wells.
- If n is even, we pick the y coordinate of the pipeline to be anything between the y coordinates of the wells with y-coordinates which have order statistics $\lfloor (n+1)/2 \rfloor$ and the $\lceil (n+1)/2 \rceil$. These can all be found in linear time using the algorithm from this section.

Problem 9-1 Largest i numbers in sorted order

> Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time,

> and analyze the running times of the algorithms in terms of n and i.
>
> **a.** Sort the numbers, and list the i largest.
>
> **b.** Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.
>
> **c.** Use an order-statistic algorithm to find the ith largest number, partition around that number, and sort the i largest numbers.

a. The running time of sorting the numbers is $O(n \lg n)$, and the running time of listing the i largest is $O(i)$. Therefore, the total running time is $O(n \lg n + i)$.

b. The running time of building a max-priority queue (using a heap) from the numbers is $O(n)$, and the running time of each call EXTRACT-MAX is $O(\lg n)$. Therefore, the total running time is $O(n + i \lg n)$.

c. The running time of finding and partitioning around the ith largest number is $O(n)$, and the running time of sorting the i largest numbers is $O(i \lg i)$. Therefore, the total running time is $O(n + i \lg i)$.

Problem 9-2 Weighted median

> For n distinct elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the ***weighted (lower) median*** is the element x_k satisfying
>
> $$\sum_{x_i < x_k} w_i < \frac{1}{2}$$
>
> and
>
> $$\sum_{x_i > x_k} w_i \le \frac{1}{2}.$$
>
> For example, if the elements are $0.1, 0.35, 0.05, 0.1, 0.15, 0.05, 0.2$ and each element equals its weight (that is, $w_i = x_i$ for $i = 1, 2, \ldots, 7$), then the median is 0.1, but the weighted median is 0.2.
>
> **a.** Argue that the median of $x_1, x_2, \ldots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, 2, \ldots, n$.
>
> **b.** Show how to compute the weighted median of n elements in $O(n \lg n)$ worstcase time using sorting.
>
> **c.** Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as SELECT from Section 9.3.
>
> The ***post-office location problem*** is defined as follows. We are given n points $p_1, p_2, \ldots, p_n$ with associated weights $w_1, w_2, \ldots, w_n$. We wish to find a point p (not necessarily one of the input points) that minimizes the sum $\sum_{i=1}^{n} w_i d(p, p_i)$, where $d(a, b)$ is the distance between points a and b.
>
> **d.** Argue that the weighted median is a best solution for the 1-dimensional postoffice location problem, in which points are simply real numbers and the distance between points a and b is $d(a, b) = |a - b|$.
>
> **e.** Find the best solution for the 2-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the ***Manhattan distance*** given by $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

a. Let m_k be the number of x_i smaller than x_k. When weights of $1/n$ are assigned to each x_i, we have $\sum_{x_i < x_k} w_i = m_k/n$ and $\sum_{x_i > x_k} w_i = (n - m_k - 1)/2$. The only value of m_k which makes these sums $< 1/2$ and $\le 1/2$ respectively is when $\lceil n/2 \rceil - 1$, and this value x must be the median since it has equal numbers of x_i's which are larger and smaller than it.

b. First use mergesort to sort the x_i's by value in $O(n \lg n)$ time. Let S_i be the sum of the weights of the first i elements of this sorted array and note that it is $O(1)$ to update S_i. Compute $S_1, S_2, \ldots$ until you reach k such that $S_{k-1} < 1/2$ and $S_k \ge 1/2$. The weighted median is x_k.

c. We modify SELECT to do this in linear time. Let x be the median of medians. Compute $\sum_{x_i<x} w_i$ and $\sum_{x_i>x} w_i$ and check if either of these is larger than $1/2$. If not, stop. If so, recurse on the collection of smaller or larger elements known to contain the weighted median. This doesn't change the runtime, so it is $\Theta(n)$.

d. Let p be the minimizer, and suppose that p is not the weighted median. Let ϵ be small enough such that $\epsilon < \min_i(|p - p_i|)$, where we don't include k if $p = p_k$. If p_m is the weighted median and $p < p_m$, choose $\epsilon > 0$. Otherwise choose $\epsilon < 0$. Thus, we have

$$\sum_{i=1}^{n} w_i d(p+\epsilon, p_i) = \sum_{i=1}^{n} w_i d(p, p_i) + \epsilon \left(\sum_{p_i<p} w_i - \sum_{p_i>p} w_i \right) < \sum_{i=1}^{n} w_i d(p, p_i),$$

the difference in sums will take the opposite sign of epsilon.

e. Observe that

$$\sum_{i=1}^{n} w_i d(p, p_i) = \sum_{i=1}^{n} w_i |p_x - (p_i)_x| + \sum_{i=1}^{n} w_i |p_y - (p_i)_y|.$$

It will suffice to minimize each sum separately, which we can do since we choose p_x and p_y individually. By part (e), we simply take $p = (p_x, p_y)$ to be such that p_x is the weighted median of the x-coordinates of the p_i's and py is the weighted medain of the y-coordiantes of the p_i's.

Problem 9-3 Small order statistics

We showed that the worst-case number $T(n)$ of comparisons used by SELECT to select the ith order statistic from n numbers satisfies $T(n) = \Theta(n)$, but the constant hidden by the Θ-notation is rather large. When i is small relative to n, we can implement a different procedure that uses SELECT as a subroutine but makes fewer comparisons in the worst case.

a. Describe an algorithm that uses $U_i(n)$ comparisons to find the ith smallest of n elements, where

$$U_i(n) = \begin{cases} T(n) & \text{if } i \geq n/2, \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) & \text{otherwise.} \end{cases}$$

(Hint: Begin with $\lfloor n/2 \rfloor$ disjoint pairwise comparisons, and recurse on the set containing the smaller element from each pair.)

b. Show that, if $i < n/2$, then $U_i(n) = n + O(T(2i)\lg(n/i))$.

c. Show that if i is a constant less than $n/2$, then $U_i(n) = n + O(\lg n)$.

d. Show that if $i = n/k$ for $k \geq 2$, then $U_i(n) = n + O(T(2n/k)\lg k)$.

(Removed)

Problem 9-4 Alternative analysis of randomized selection

In this problem, we use indicator random variables to analyze the RANDOMIZED-SELECT procedure in a manner akin to our analysis of $\text{RANDOMIZED-QUICKSORT}$ in Section 7.4.2.

As in the quicksort analysis, we assume that all elements are distinct, and we rename the elements of the input array A as $z_1, z_2, \ldots, z_n$, where z_i is the ith smallest element. Thus, the call $\text{RANDOMIZED-SELECT}(A, 1, n, k)$ returns z_k.

For $1 \leq i < j \leq n$, let

$$X_{ijk} = I\{z_i \text{ is compared with } z_j \text{ sometime during the execution of the algorithm to find } z_k\}.$$

a. Give an exact expression for $E[X_{ijk}]$. (Hint: Your expression may have different values, depending on the values of i, j, and k.)

b. Let X_k denote the total number of comparisons between elements of array A when finding z_k. Show that

$$E[X_k] \le 2\left(\sum_{i=1}^{k} \sum_{j=k}^{n} \frac{1}{j-i+1} + \sum_{j=k+1}^{n} \frac{j-k-1}{j-k+1} + \sum_{i=1}^{k-2} \frac{k-i-1}{k-i+1} \right).$$

c. Show that $E[X_k] \le 4n$.

d. Conclude that, assuming all elements of array A are distinct, RANDOMIZED-SELECT runs in expected time $O(n)$.

(Removed)