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.)

Step 1 of 3

To find the smallest element construct a tournament follows:
Compare all the numbers in pairs. Only the smallest number of each pair is potentially the smallest of all so the problem is reduced to size $\lceil n/2 \rceil$. Continuing in this fashion until there is only one left clearly solves the problem.

Step 2 of 3

Exactly n -1 comparisons are needed since the tournament can be drawn as an n – leaf binary tree which has n – 1 internal nodes (shown by induction on n). Each of these nodes correspond to a comparison.
We can use this binary tree to also locate the second smallest number. The path from the root to the smallest element (of height $\lceil \lg n \rceil$ must contain the second smallest element. Conducting a tournament among these uses $\lceil \lg n \rceil - 1$ comparisons.

Step 3 of 3

The total number of comparisons are $n-1+\lceil \lg n \rceil -1=n+\lceil \lg n \rceil -2$.

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.)

Step 1 of 2

Optimal number of comparisons to find the maximum and minimum of *n* numbers in the worst case scenario

Step 1: Set initial values for the minimum and maximum.

i. If *n* is odd, minimum and maximum are set to the value of the first element.

ii. If *n* is even, compare the first 2 elements and assign the smaller element to minimum and larger element to maximum.

Step 2: Process elements in pairs.

i. Compare pairs of elements from the input with each other.

ii. Compare the smaller to the current minimum.

iii. Compare the larger to the current maximum.

Step 2 of 2

As processing of elements is done in pairs, $n/2$ pairs are processed.

To process each pair of elements, 3 comparisons are required.

[One comparison to compare pairs of elements, one comparison to compare the smaller to the current minimum and one comparison to compare the larger to the current maximum.]

Hence $3(n/2)$ comparisons are done.

If n is odd, $3(n/2)$ comparisons are performed.

If n is even, one initial comparison is done to set initial value for maximum and minimum and $3(n-2)/2$ comparison are performed for the remaining $(n-2)/2$ pairs.

Calculate the total number of comparisons by adding these two values.

$$\begin{aligned} 3(n-2)/2+1 &= (3n-6)/2+1 \\ &= (3n-6+2)/2 \\ &= (3n-4)/2 \\ &= 3n/2-2 \end{aligned}$$

In the worst case scenario, consider the upper bound to calculate the total number of comparisons.

Hence $\boxed{\lceil 3n/2 \rceil - 2}$ comparisons are necessary to find both the minimum and the maximum of n numbers.

9.2-1

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

Step 1 of 1

Let us consider 0-length array as input to RANDOMIZED-SELECT algorithm. From the algorithm we need to pass first position, last position and ith position of the array A. A 0-length array is made by partition. So, k must be either 1 or n. The indices of sub array p and r becomes 0, because the array length is 0. Therefore the recursive call is not made to a 0-length array unless we want 0-order statistic.

9.2-2

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

Step 1 of 1

The indicator random variable $X_k = I(\text{the subarray } A[p..q] \text{ has exactly } k \text{ elements})$ is independent of the maximum value of $T(k-1)$ and $T(n-k)$.

We having chosen subarray $A[p..q]$ has k elements, next we may do is

RANDOMIZED-SELECT (A, p, q -1, i) or

RANDOMIZED-SELECT $(A, q+1, \mathrm{r}, i-k)$.

They are the same as any other RANDOMIZED-SELECT algorithm build on the arrays $\frac{A[p,\ldots,q-1]}{A[q+1,\ldots,r]}$ to find the $\frac{i}{i-k}$ order elements. The time complexity is not affected at all by the choice of the length of array, hence the random variables X_k and $T(\max(k-1, n-k))$ are independent.

9.2-3

Write an iterative version of RANDOMIZED-SELECT.

Step 1 of 2

Iterative Algorithm for RANDOMIZED-SELECT

The following algorithm demonstrates an iterative version of the RANDOMIZED-SELECT algorithm. This algorithm returns i^{th} smallest element in the array *A*.

In this algorithm,

A - an array of elements

p - lower position of the array

q - higher position of the array

i - position of an element which is to be returned

RANDOMIZED-SELECT(A, p, r, i)

1. **while** $p<r$ // verify whether the *p* is less than *r*, if yes repeat the statements 2 to 8
2. $q=$ RANDOMIZED-PARTITION(A, p, r) // get partition position
3. $k=q-p+1$
4. **if** $i \le k$ // desired element is at lower side if true
5. $r=q$ // update the value of *r*
6. **else** // desired element is at higher side
7. $p=q+1$ // update the value of *p*
8. $i=i-k$
9. **return** $A[p]$

Step 2 of 2

Note:

For RANDOMIZED-PARTITION algorithm in the RANDOMIZED-SELECT algorithm, refer the text book, page number - 179.

For PARTITION algorithm in the RANDOMIZED- PARTITION algorithm, refer the text book, page number - 171.

For RANDOM(*p*, *r*) in the RANDOMIZED- PARTITION algorithm returns a ranom value between *p* and *r* (inclusive), and refer the text book, page number - 117.

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.

Step 1 of 4

Consider the analysis of the algorithm for groups of k. The number of elements less than (or greater than) the median of the medians x will be at east $\lceil \frac{k}{2} \rceil \left(\lceil \frac{1}{2} \lceil \frac{n}{k} \rceil \rceil - 2 \right) \geq \frac{n}{4} - k$.

Hence, in the worst − case SELECT will be called recursively on at most $n - (\frac{n}{4} - k) = \frac{3n}{4} + k$ elements.

Step 2 of 4

The recurrence is

$$T(n) \leq T(\lceil n/k \rceil) + T\left(\frac{3n}{4} + 4\right) + O(n)$$

Solving by substitution we obtain a bound for which k the algorithm will be linear. Assume $T(n) \leq cn$ for all smaller n.

Step 3 of 4

We have:

$$\begin{aligned} T(n) &\leq c\lceil \frac{n}{k} \rceil + c\left(\frac{3n}{4} + k\right) + O(n) \\ &\leq c\left(\frac{n}{k} + 1\right) + \frac{3cn}{4} + ck + O(n) \\ &\leq \frac{cn}{k} + \frac{3cn}{4} + c(k+1) + O(n) \\ &= cn\left(\frac{1}{k} + \frac{3}{4}\right) + c(k+1) + O(n) \\ &\leq cn \end{aligned}$$

Step 4 of 4

where the last equation only holds for k>4. Thus, we have shown that the algorithm will compute in linear time for any group size of 4 or more. If fact, the algorithm is $\Omega(n \lg n)$ for k =3.

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.

Step 1 of 3

We have the recurrence,

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \leq 140 \\ T(\lceil n/5 \rceil) + T(7n/10+6) + O(n) & \text{if } n > 140 \end{cases}$$

$T(n) \leq cn \quad$ for some $n \leq 140$

Step 2 of 3

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10+6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

which is at most cn if

$$-cn/10 + 7c + an \leq 0 \qquad -(1)$$

Step 3 of 3

Inequality (1) is equivalent to the in equality $c \geq 10a(n/(n-70))$ when $n > 70$

because we assume that $n \geq 140$, we have $n/(n-70) < 2$ and so choosing

$c \geq 20a$ will satisfy the inequality (1). Thus the worst – case running time of select is therefore linear.

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.

Step 1 of 3

A modification to quick sort that allows it to run in O(n lg n) time in the worst case uses the deterministic PARTITION algorithm that was modified to take an element to partition around as input parameter.

SELECT takes an array A, the bounds p and r of the sub array in A, and the rank i of an order statistic, and in time linear in the size of the sub array $A[p...r]$ it returns the i^{th} smallest element in $A[p....r]$.

Step 2 of 3

BEST-CASE-QUICKSORT (A, p, r)

1. if $p < r$
2. then $i = \lfloor (r-p+1)/2 \rfloor$
3. $x = SELECT(A, p, r, i)$
4. $q = PARTITION(x)$
5. BEST-CASE-QUICKSORT (A, p, q − 1)
6. BEST-CASE-QUICKSORT (A, q+1, r)

Step 3 of 3

For an n − element array, the largest sub array that BEST-CASE-QUICKSORT recurses on has n/2 elements. This situation occurs when $n = r - p+1$ is even; then the sub array A[q+1 .....r] has n/2 elements, and the sub array A[p.....q-1] has n/2 -1 elements.

Because BEST-CASE-QUICKSORT always recurses on sub arrays that are at most half the size of the original array, the recurrence for the worst − case running time is

$$T(n) \le 2T(n/2)+\Theta(n)= \boxed{O(n \lg n)}$$

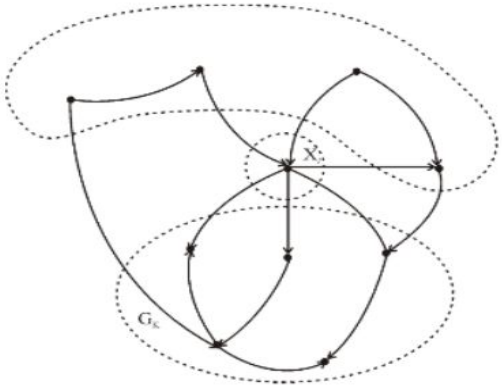
9.3-4 ★

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 the $n - i$ larger elements without performing any additional comparisons.

Step 1 of 4

We can construct a graph, whose nodes are the elements of the array, whose edges are comparison results.
Let's denote an edge from j to k as $X_j > X_k$ and vice versa.
Therefore, an algorithm uses only comparisons to find the i^{th} smallest element can be represented by a set of edges such that i^{th} smallest element can be found.

Step 2 of 4



Step 3 of 4

Let's denote X_i is the i^{th} smallest number.
Based on the definition of i^{th} smallest number, it should be longer that $(i-1)$ elements, and smaller than $(n-i-1)$ elements.

So, the algorithm, represented by the directional edge, should allow us to derive these $(n-4)$ comparisons, therefore, when we construct the comparison between i^{th} smallest number and the $(i-1)$ smaller elements we should be able to follow the directional edges starting from Xi to each these $(i-1)$ elements such that as comparison result $X_i > X_j$ can be verified.

Step 4 of 4

In addition only $(i-1)$ smaller elements can be found w/o any other comparison at all. Similarly, we can argue that by reverse following the edges (meaning from dest to source) that ends at X_i, we can found $(n-i-1)$ longer elements as well.

In summary, by following the edges originating from X_i, $(i-1)$ smaller elements can be found, by following only the reverse edges from X_i, $(n-i)$ (including X_i) longer elements can be found. And all edges, or comparison results are already derived from the algorithm.

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.

Step 1 of 3

We assume that are given a procedure median that takes as parameters an array A and sub array indices p and r, and returns the value of the media element of A[p.....r] in O(n) time in the worst case.

Given MEDIAN, here is a linear – time algorithm SELECT' for finding the i^{th} smallest element in A[p....r]. This algorithm uses the deterministic partition algorithm that was modified to take an element to partition around as an input parameter.

Step 2 of 3

SELECT' (A, p, r, i)

```
1.   if p = r
2.      then return A[p]
3.   x = MEDIAN (A, p, r)
4.   q = PARTITION (x)
5.   k = q – p +1
6.   if  i = k
7.      then return A[q]
```

8. else if i < k
9. then return SELECT (A, p, q-1, i)
10. else return SELECT$'$(A, q+1, r, i – k)

Step 3 of 3

Because x is the median of A[p.....r], each of the sub arrays A[p...q-1] and A[q+1, ...r] has at most half the number of elements of A [p...r]. The recurrence for the worst – case running time of SELECT$'$ is $T(n) \le T(n/2)+O(n)= O(n)$.

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.

Step 1 of 3

The k^{th} quantiles of an n – element set are k -1 order statistics that divide the sorted set into k equal – sized sets (to within 1).

The k quantiles of an n – element sorted array A are

$A[\lfloor 1.n/k \rfloor], A[\lfloor 2.n/k \rfloor],\ldots\ldots, A[\lfloor (k-1).n/k \rfloor]$

We have inputs as an unsorted array A of distinct keys, an integer k, and an empty array Q of length k -1 and we have to find the k^{th} Quantiles of A.

Step 2 of 3

Quantiles (A, k, Q)

1. if k = 1 then return
2. else
3. n = length [A]
4. i = $\lfloor k/2 \rfloor$
5. x = SELECT (A, $\lfloor i.n/k \rfloor$)
6. partition (A, x)
7. ▷ Add to list Q: Quantiles (A[1] ....A[$\lfloor 1.n/k \rfloor$, $\lfloor k/2 \rfloor$, Q)
8. ▷ Add to list Q: Quantiles $\left(A[\lfloor i.n/k \rfloor+1].....A[n], \lceil k/2 \rceil, Q\right)$
9. return x

Step 3 of 3

The output of Q be the recursive algorithm quantiles contains the k^{th} quantiles of A. The algorithm first finds the key that turns out to be the lower median of Q, and then partitions the input array about this key. So we have two smaller arrays that each contains at most (k -1)/2 of the k -1 order statistics of the original array A at each level of the recursion the number of order statistics in the array is at most half the number of the previous array.

Consider a recursion tree for this algorithm. At the top level we need to find k -1 order statistics, and it costs O (n) to find one. The root has two children, one contains at most $\lfloor (k-1)/2 \rfloor$ order statistics, and the other $\lceil (k-1)/2 \rceil$ order statistics. The sum of the costs of all nodes at depth i is O (n) for $O \le 1 \le \log_2 (k-1)$, because the total number of elements at any depth is n. The depth of the tree is $d = \log_2 (k-1)$. Hence, the worst case running time of quantiles is $O(n \lg k)$.

9.3-7

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

Step 1 of 1

Suppose the k numbers are in the range $S[i_1, i_2]$. To find the elements at $S[i_1]$ and $S[i_2]$, then compare all the elements in S with $S[i_i]$ and $S[i_2]$.

An O(n)-time algorithm that, given a set S of n distinct numbers and a positive integer $k \le n$ are

Select-k(S,k)

1. if n=k then return S
2. else i ← k mod 2
3. low ← Select(A,1,n, $\lfloor n/2 \rfloor - \lfloor k/2 \rfloor$)
4. high ← Select(A,1,n, $\lfloor n/2 \rfloor + \lfloor k/2 \rfloor$)
5. if i=0 then if low ≤ high
6. then high ← Select(A,1,n, $\lfloor n/2 \rfloor + \lfloor k/2 \rfloor - 1$
7. else low ← Select(A,1,n, $\lfloor n/2 \rfloor - \lfloor k/2 \rfloor + 1$
8. for i ← 1 to n do
9. if low ≤ S ≤ high then return S[i]

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.

Step 1 of 4

Let's start out by supposing that the median (the lower median, since we know we have an even number of elements) is in X. Let's call the median value m and lets suppose that its in X[k] then k elements of X are less than or equal to m and n-k elements of X are greater than or equal to m. We know that in the two arrays combined, there must be n elements less than are equal to m, and so there must be $n-k$ elements of Y that are less than or equal to m and $n-(n-k)=k$ elements of Y that are greater than or equal to m.

Thus, we can check that X[k] is the lower median by checking whether $Y[n\text{-}k] \le X[k] \le Y[n\text{-}k+1]$. A boundary case occurs for $k=n$. Then $n-k=0$. And there is no array entry Y[0]; we only need to check that $X[n] \le Y[1]$.

Step 2 of 4

Now, if the median is in X but is not in X[k], then the above condition will not hold. If the median is in X[k'], where $k' < k$, then X[k] is above the median, and $Y[n-k+1] < X[k]$. Conversely, if the median is in X[k"], where $k" > k$, then X[k] is below the median, and $X[k] < Y[n\text{-}k]$.

Thus, we can use a binary search to determine whether there is an X[k] such that either $k<n$ and $Y[n\text{-}k] \le X[k] \le Y[n\text{-}k+1]$ or $k=n$ and $X[k] \le Y[n\text{-}k+1]$, if we find such an X[k] then it is the median. Otherwise, we know that the median is in Y, and we use a binary search to find a Y[k] such that either $k < n$ and $X[n\text{-}k] \le Y[k] \le X[n\text{-}k+1]$ or $k=n$ and $Y[k] \le X[n\text{-}k+1]$ such a Y[k] is the median. Since each binary search takes O(lg n) time, we spend a total of O(lg n) time.

Step 3 of 4

The algorithm in pseudecode:


```
TWO – ARRAY – MEDIAN (X, Y)

1.   n ← length [x]          ▷ n also equals length [Y]
2.   median ← FIND-MEDIAN (X, Y, n, 1, n)
3.   if median = NOT-FOUND
4.      then median ← FIND-MEDIAN (Y, X, n, 1, n)
5.   return median
```

Step 4 of 4

```
FIND-MEDIAN (A, B, n, low, high)

1.    if low > high
2.       then return NOT-FOUND
3.    else k ← ⌊(low+high)/2⌋
4.         if k = n and A [n] ≤ B [1]
5.            then return A [ n ]
6.         else if k < n and B [ n - k ] ≤ A [ k ] ≤ B [ n - k +1 ]
7.               then return A [ k ]
8.         else if A [ k ] > B [n – k +1 ]
9.               then return FIND-MEDIAN ( A, B, n, low, k – 1 )
10.        else return find – median (A, B, n, k+1, high )
```

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

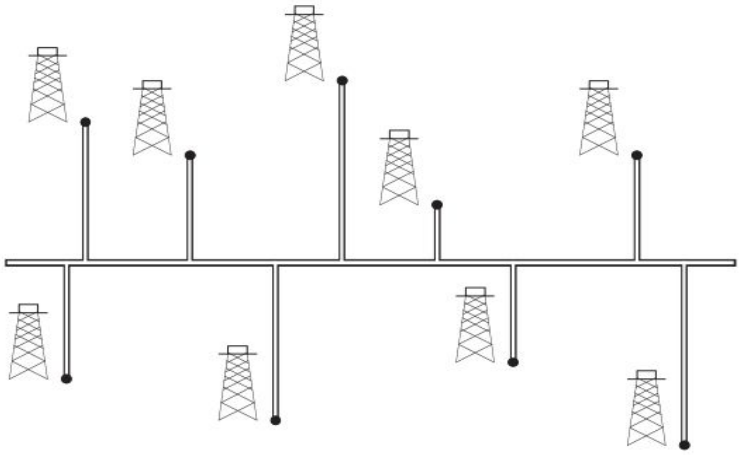


Figure 9.2 Professor Olay needs to determine the position of the east-west oil pipeline that minimizes the total length of the north-south spurs.

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.

Step 1 of 5

In order to find the optimal placement for professor Olay's pipeline, we need only find the median (s) of the y – coordinates of his oil wells, as the following proof explains.

Claim:

The optimal Y – coordinate for professor Olay's east – west oil pipeline is as follows:

- If n is even, then on either the oil well whose y – coordinate is the lower median or the one whose y – coordinate is the upper median, or anywhere between them.
- If n is odd, then on the oil well whose y – coordinate is the median.

Step 2 of 5

Proof:

We examine various cases. In each case, we will start out with the pipeline at a particular y – coordinate and see what happens when we move it. We'll denote by s the sum of the north –south spurs with the pipeline at the starting location, and s' will denote the sum after moving the pipeline.

We start with the case in which n is even. Let us start with the pipeline somewhere on or between the two oil wells whose y – coordinates are the lower and upper medians. If we move the pipeline by a vertical distance d without crossing either of the median wells, then n/2 of the wells become d farther from the pipeline and n/2 become d closer, and so $s' = s+dn/2 - dn/2 = s$; thus all locations on or between the tow medians are equally good.

Step 3 of 5

Now suppose that the pipeline goes throughout the oil well whose y – coordinate is the upper median. What happens when we increase the y – coordinates of the pipeline by d > 0 units so that it moves above the oil well that achieves the upper median? All oil wells whose y – coordinates are at or below the upper median become d units farther from the pipeline, and there are at least n /2+1 such oil wells (the upper median, and every well at or below the lower median) there are at most n/2-1 oil wells whose y – coordinates are above the upper median, and each of these oil wells becomes at most d units closer to the pipeline when it moves up. Thus, we have a lower bound on s' of $s' \geq s+d(n/2+1)-d(n/2-1)=s+2d>s$.

Step 4 of 5

We conclude that moving the pipeline up from the oil well at the upper median increases the total spur length. A symmetric argument shows that if we start with pipeline going through the oil well whose y - coordinate is the lower median and move it down, then the total spur length increases.

We see, therefore, that when n is even, an optimal placement of the pipeline is anywhere on or between the two medians.

Step 5 of 5

Now we consider the case when n is odd. We start with the pipeline going through the oil well whose y – coordinate is the median, and we consider what happens when we move it up by d > 0 units. All oil wells at or below the median become d units farther from the pipeline, and there are at least (n+1)/2 such wells (the one at the median and the (n-1)/2 at or below the median. There are at most (n-1)/2 oil wells above the median, and each of these becomes at most d units closer to the pipe line. We get a lower bound on s' of $s' \geq s+d(n+1)/2 -d(n-1)/2=s+d>s$, and, we conclude that moving the pipeline up from the oil well at median increases the total spur length. A symmetric argument shows that moving the pipeline down from the median also increases the total spur length, and so the optimal placement of the pipeline is on the median.

Since we know are looking for the median, we can use the linear time median – finding algorithm.

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.

Step 1 of 3

We assume that the numbers starts out in an array.

(A) Sort the numbers using merge sort or heapsort, which take $\Theta(n \lg n)$ worst – case time. (Don't use quicksort or insertion sort, which can take $\Theta(n^2)$ time.) Put the i largest elements (directly accessible in the sorted array) into the output array taking $\Theta(i)$ time.

Total worst – case running time: $\Theta(n \lg n + i) = \Theta(n \lg n)$ (since $i \leq n$)

Step 2 of 3

(B) Implement the priority queue as a heap. Build the heap using BUILD-HEAP, which takes $\Theta(n)$ time, then call HEAP-EXTRACT-MAX i times to get the i largest elements in $\Theta(i \lg n)$ worst - case – time, and store them in reverse order of extraction in the output array. The worst – case extraction time is $\Theta(i \lg n)$ because

- i extractions from a heap with O(n) elements takes $i \cdot O(\lg n) = O(i \lg n)$ time, and
- half of the i extractions are from a heap with $\geq n/2$ elements, so those i/2 extractions takes $(i/2)\Omega(\lg(n/2)) = \Omega(i \lg n)$ time in the worst case.

Step 3 of 3

(C) Use the SELECT algorithm to find the i^{th} largest number in $\Theta(n)$ time. Partition around that number in $\Theta(n)$ time. Sort the i largest numbers in $\Theta(i \lg i)$ worst - case time (with merge sort or heap sort).

Total worst – case running time: $\Theta(n + i \lg i)$.

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

$$\sum_{x_i > x_k} w_i \leq \frac{1}{2}.$$

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 $O(n \lg n)$ worst-case 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 post-office 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 $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

Step 1 of 8

a .

Consider that $x_1, x_2, x_3, \ldots, x_n$ are the n samples and the weights of the samples are :

$$w_i = \frac{1}{n} \text{ for } i = 1, 2, 3, \ldots, n$$

Proving that, the median of above numbers is also the weighted median. The weighted mean is found by using the formula:

$$\overline{x} = \frac{\sum w.x}{\sum w}$$

$$\overline{x} = \frac{\sum_{i=1}^{n} w_i.x_i}{\sum_{i=1}^{n} w_i}$$

$$\overline{x} = \frac{\frac{1}{1} \times 1 + \frac{1}{2} \times 2 + \cdots + \frac{1}{n} \times n}{\sum_{i=1}^{n} n}$$

$$\overline{x} = \frac{1 + 1 + \cdots + 1}{\sum_{i=1}^{n} n}$$

$$\bar{x} = \frac{n}{n(n+1)/2}$$

$$\bar{x} = \frac{2}{n+1} \quad \ldots\ldots (1)$$

Step 2 of 8

Keeping the values of $n = 1, 2, 3, 4, \ldots, n$. When there are 3 elements then median is 2nd element by formula:

Median, $i = \frac{n+1}{2}$

And second element of $w_i = \frac{1}{n}$ is $1/2$

Similarly, on substituting $n = 3$ in equation (1),

$$\bar{x} = \frac{2}{n+1}$$
$$= \frac{1}{2}$$

Similarly, same for all values of n. Hence, it is proved that median of $x_1, x_2, x_3, \ldots, x_n$ is the weighted median of x_i when the weights

$w_i = \frac{1}{n}$ for $i = 1, 2, 3, \ldots, n$.

b .

To point out the weighted median of *n* elements in worst time running of $O(n \lg n)$, sort the items using **merge-sort** or **heap-sort** whose complexity is $O(n \lg n)$. For this, starting with the first item iterate over the items in first list, taking the sum of the weights of the items at the time of execution, until the running sum does not go over0.5. Thus, on going with the algorithm as:

Sort-Weighted- Median $(x, 1, n)$

// applying any of the merge sort or heap sort on the items.

1. Merge sort $(x, 1, n)$ or heap sort $(x, 1, n)$

// initializing the value of variable *sum*

2. $sum = 0$

// M - number of the present element

3. $M = 0$

// iterate the loop until the less than 0.5

4. **while** ($sum < 0.5$)

// increasing the value of *M* by one.

5. $M = M + 1$

// adding the wm - weight of the element in the *sum*

6. $sum = sum + wm$

// end of while loop.

7. **end** while

8. **return** M

Step 3 of 8

Analysis of algorithm:

The analysis of the above algorithm for the calculation of complexity is as:

Step 1 has complexity of $O(n \lg n)$ as it the complexity of merge sort and heap sort.

Step 2 and **step 3** are of $O(1)$ since, they are simple assignments.

Steps 4, 5, 6 are of $O(n)$, since loops for *n* items.

Thus, on calculating the complexity:

$$= O(n \lg n) + 2 \times O(1) + 4 \times O(n)$$
$$= O(n \lg n)$$

Hence, Proved.

Step 4 of 8

c .

To get the weighted mean of n elements in worst time running of $O(n)$, sort the items using **radix-sort** or **bucket-sort** whose complexity is $O(n)$. For this starting take the very first item and iterate over the items in first list, putting a iterative sum of the weights of the items, until the running sum is not go over 0.5.The algorithm is as:

// applying one of the sorting either Redix-sort or bucket-sort on data items.

1. Redix-sort $(x, 1, n)$ or bucket sort $(x, 1, n)$

// initializing the value of variable *sum*

2. $sum = 0$

3. $M = 0$ // M - number of the present element

// iterate the loop until the less than 0.5

4. **while** ($sum < 0.5$)

// increasing the value of *M* by one.

5. $M = M + 1$

// adding the wm - weight of the element in the *sum*

6. $sum = sum + wm$

// end of while loop

7. **end** while

8. **return** M

Step 5 of 8

Analysis:

Step 1 has complexity of $\Theta(n)$ as it is the complexity of redix sort and bucket sort for *n* elements.

Step 2 and **step 3** take $\Theta(1)$ time since, they are simple assignments.

Steps 4, 5, 6 are of $\Theta(n)$, since loops for *n* items.

Thus, on calculating the complexity:

$$= \Theta(n) + 2 \times \Theta(1) + 4 \times \Theta(n)$$
$$= \Theta(n)$$

Hence, proved.

d .

Solution to the problem of one-dimensional post-office position is the weighted median of the points. Now, choosing *p* to minimize the cost.

$$c(p) = \sum_{i=1}^{n} w_i d(p, p_i)$$

The $c(p)$ can be stated as the sum of the cost contributed by points less than *p* and points greater than *p*:

$$c(p) = \sum_{p_i < p} w_i (p - p_i) + \sum_{p_i > p} w_i (p_i - p)$$

Now, if, $p = p_k$ for some *k*, then that point will not donate to the cost .The representative function of cost is continuous because:

$\lim_{p \to x} c(p) = c(x)$ for all *x*.

Step 6 of 8

To find the minima of this function, take the derivative according to *p*.

$$\frac{dc}{dp} = \left(\sum_{p_i < p} w_i \right) - \left(\sum_{p_i > p} w_i \right)$$

Note that this derivative is undefined where $p = p_i$ for some *i* because the left- and right-hand limits of *c*(*p*) differ. Keep in mind that $\frac{dc}{dp}$ is also a non-decreasing function because *dc* as *p* increases the point's numbers $p_i < p$ cannot decrease.

Note that $\frac{dc}{dp} < 0$ for $p < \min(p_1, p_2, \ldots, p_n)$ and $dc > 0$ for $p > \max(p_1, p_2, \ldots, p_n)$.

Therefore there is some point *p* such that $\frac{dc}{dp} \leq 0$ for points $p <$ <i>p_i and 0

$\frac{dc}{dp} \geq 0$ for points $p_i > p$

This point is a global minimum. It is showed that the weighted median y is such a point for all points p <<i> y where p is not the weighted median and $p \neq p_i$. For some i,

$$\left(\sum_{p_i<p} w_i\right) < \left(\sum_{p_i>p} w_i\right)$$

This implies that $\frac{dc}{dp} < 0$.

Step 7 of 8

Similarly, for points $p > y$ where p is not the weighted median and $p \neq p_i$ for some i,

$$\left(\sum_{p_i<p} w_i\right) > \left(\sum_{p_i>p} w_i\right)$$

This implies that $\frac{dc}{dp} > 0$

For the cases where $p = p_i$ for some i and $p \neq y$, both the left- and right-hand limits of $\frac{dc}{dp}$ always have the same sign so, the same argument applies.

Therefore, $c(p) > c(y)$ for all p that is not the weighted median, so the weighted median y is a **global minimum**. **Hence, proved.**

Step 8 of 8

***e*.**

Solution of the 2-dimensional problem of post-of?ce location using **Manhattan distance** is equivalent to solution of the one-dimensional post-office location problem **separately for each dimension**.

Consider the solution is $p = (p_x, p_y)$. Notice that using Manhattan distance which can write the cost function as the sum of two one-dimensional post-office location cost functions as follows:

$$g(p) = \sum_{i=1}^{n} w_i |x_i - px| + \left(\sum_{i=1}^{n} w_i |y_i - py|\right)$$

From above, $\delta q / \delta px$ does not have the dependency on the y coordinates of the input points. It has the form exactly as dc/dp from the part done before using the x coordinates as input. Similarly, $\delta q / \delta px$ is based only on the y coordinate.

Therefore to minimize $g(p)$, which can lessen the cost for the two dimensions independently. The appropriate result to the two dimensional problem can be done as following:

Consider p_x be the solution to the one-dimensional post-of?ce location problem for inputs $x_1, x_2, \ldots, x_n$, p_y be the solution to the one-dimensional post-of?ce location problem for inputs $y_1, y_2, \ldots, y_n$.

Thus, it is the optimal solution.

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 $U_i(n) = n + O(T(2n/k)\lg k)$.

Step 1 of 11

(A) Our algorithm relies on a particular property of SELECT that not only does it return the i^{th} smallest element, but that it also partitions the input array so that the first i positions contain the i smallest elements (though not necessarily in sorted order). To see that SELECT has this property, observe that there are only two ways in which returns a value: when n =1, and when immediately after partitioning in step 4, it finds that there are exactly i elements on the low side of the partition.

Taking the hint from the book, here is our modified algorithm to select the i^{th} smallest element of n elements. Whenever it is called with $i \geq n/2$, it just calls SELECT and returns its results. In this case, $U_i(n)=T(n)$.

Step 2 of 11

When $i < n/2$, our modified algorithm works as follows. Assume that the input is in a subarray $A[p+1,.....p+n]$, and let $m = \lfloor n/2 \rfloor$. In the initial call, $p = 1$.

1. Divide the input as follows. If n is even, divide the input into two parts: $A[p+1.....p+m]$ and $A[p+m+1....p+n]$. If n is odd, divide the input into three parts: $A[p+1....p+m]$, $A[p+m+1.....p+n-1]$ and $A[p+n]$ as a leftover piece.
2. Compare A[p+i] and $A[p+i+m]$ for i= 1,2,....m. putting the smaller of the two elements into A[p+i+m] and the larger into A[p+i].

Step 3 of 11

3. Recursively find the i^{th} smallest element in the $A[p+m+1.....p+n]$ but with an additional action performed by the partitioning procedure: whenever it exchanges A[j] and A[k] (where $p+m+1 \le j, k \le p+2m$) it also exchanges $A[j-m]$ and $A[k-m]$. The idea is that after recursively finding the i^{th} smallest element in $A[p+m+1...p+n]$, the subarray $A[p+m+1...p+m+i]$ contains the i smallest elements that had been in A[p+m+1....p+n] and the sub array A[p+1...p+i] contains their larger counter parts as found in stiep1. the i^{th} smallest element of A[p+1...p+n] must be either one of the i smallest, as placed into $A(p+m+1....p+m+i)$ or it must be one of the larger counter parts, as placed into A[p+1....p+i].

Step 4 of 11

4. Collect the sub arrays A[p+1....p+i] and A[p+m+1....p+m+i] into a single array B[1...2i], call SELECT to find the i^{th} smallest element of B, and return the result of this call to SELECT.

Step 5 of 11

The number of comparisons in each step is as follows.

1. No comparisons.
2. $m = \lfloor n/2 \rfloor$ comparisons.
3. Since we recurse on $A[p+m+1.....p+n]$ which has $\lceil n/2 \rceil$ elements, the number of comparisons is $U_i(\lceil n/2 \rceil)$.
4. Since we call SELECT on an array with 2i elements, the number of comparisons is T(2i).

Thus, when $i<n/2$, the total number of comparisons is $\lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i)$.

Step 6 of 11

(B) We show by substitutions that if $i < n/2$, then $U_i(n) = n + O(T(2i)\lg(n/i))$. In particular, we shall show that $U_i(n) \le n + cT(2i)\lg(n/i) - d(\lg\lg n)T(2i) = n + cT(2i)\lg n - cT(2i)\lg i - d(\lg\lg n)T(2i)$ for some positive constant c, some positive constant d to be chosen later, and $n \ge 4$.
We have

$$\begin{aligned}
U_i(n) &= \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) \\
&\le \lfloor n/2 \rfloor + \lceil n/2 \rceil + cT(2i)\lg\lceil n/2 \rceil - cT(2i)\lg i - d(\lg\lg\lceil n/2 \rceil)T(2i) \\
&= n + cT(2i)\lg\lceil n/2 \rceil - cT(2i)\lg i - d(\lg\lg\lceil n/2 \rceil)T(2i) \\
&\le n + cT(2i)\lg(n/2+1) - cT(2i)\lg i - d(\lg\lg(n/2)T(2i)) \\
&= n + cT(2i)\lg(n/2+1) - cT(2i)\lg i - d(\lg(\lg(n-1))T(2i)) \\
&\le n + cT(2i)\lg n - cT(2i)\lg i - d(\lg\lg n)T(2i)
\end{aligned}$$

if $cT(2i)\lg(n/2+1) - d(\lg(\lg n-1)T(2i)) \le cT(2i)\lg n - d(\lg\lg n)T(2i)$.

Step 7 of 11

Simple algebraic manipulations give the following sequence of equivalent conditions:

$$cT(2i)\lg(n/2+1)-d(\lg(\lg n-1))T(2i)\le cT(2i)\lg n-d(\lg\lg n)T(2i)$$

$$c\lg(n/2+1)-d(\lg(\lg n-1))\le c\lg n-d(\lg\lg n)$$

$$c(\lg(n/2+1)-\lg n)\le d(\lg(\lg n-1)-\lg\lg n)$$

$$c\left(\lg\frac{n/2+1}{n}\right)\le d\lg\frac{\lg n-1}{\lg n}$$

$$c\left(\lg\left(\frac{1}{2}+\frac{1}{n}\right)\right)\le d\lg\frac{\lg n-1}{\lg n}$$

Step 8 of 11

Observe that $1/2+1/n$ decreases as n increases, but $(\lg n-1)/\lg n$ increases as n increases. When $n=4$, we have $1/2+1/n=3/4$ and $(\lg n-1)/\lg n=1/2$. Thus we just need to choose d such that $c\lg(3/4)\le d\lg(1/2)$ or, equivalently, $c\lg(3/4)\le -d$. Multiplying both sides by -1, we get $d\le -c\lg(3/4)=c\lg(4/3)$. Thus, any value of d that is at most $c\lg(4/3)$ suffices.

Step 9 of 11

(C) When i is a constant, $T(2i)=O(1)$ and $\lg(n/i)=\lg n-\lg i=O(\lg n)$. Thus, when i is a constant less than n/2, we have that is a constant less then n/2, we have that

$$\begin{aligned}U_i(n)&=n+O(T(2i)\lg(n/i))\\&=n+O(O(1).O(\lg n))\\&=n+O(\lg n)\end{aligned}$$

Step 10 of 11

(D) Suppose that $i=n/k$ for $k\ge 2$. Then $i\le n/2$. If $k>2$, then $i<n/2$, and we have

$$\begin{aligned}U_i(n)&=n+O(T(2i)\lg(n/i))\\&=n+O(T(2n/k)\lg(n/(n/k)))\\&=n+O(T(2n/k)\lg k)\end{aligned}$$

Step 11 of 11

If $k=2$, then $n=2i$ and $\lg k=1$. we have

$$\begin{aligned}U_i(n)&=T(n)\\&=n+(T(n)-n)\\&\le n+(T(2i)-n)\\&=n+(T(2n/k)-n)\\&=n+(T(2n/k)\lg k-n)\\&=n+O(T(2n/k)\lg k)\end{aligned}$$

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 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 RANDOMIZED-SELECT$(A, 1, n, k)$ returns z_k.

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

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

a. Give an exact expression for $\mathrm{E}[X_{ijk}]$. (*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

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

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

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

Step 1 of 4

Alternative analysis of randomized selection

a. In quick sort two elements z_i and z_j will be compared if they lie in the same partition. In other sense it can be said that two elements z_i and z_j will not be compared if they lie in separate partitions. That is two elements z_i and z_j will not be compared if a pivot element is chosen from the set $\{z_{i+1}, z_{i+2}, z_{i+3} \ldots\ldots, z_{j-1}\}$.

Now, consider that Z_{ijk} is defined as below,

$$Z_{ijk} = \begin{cases} \{z_i, z_{i+1}, \ldots, z_j\} & \text{if } i \le k \le j \\ \{z_k, z_{k+1}, \ldots, z_j\} & \text{if } k < i \\ \{z_i, z_{i+1}, \ldots, z_k\} & \text{if } j < k \end{cases}$$

It is quite obvious to see that until the pivot element is from set Z_{ijk}, each and every element of Z_{ijk} will be in the same partition and any element of the Z_{ijk} will have same probability to be selected as the pivot.

So,

$E[X_{ijk}]$ = Probability $\{z_i$ or z_j is the first pivot chosen from $z_{ijk}\}$

= Probability $\{z_i$ is the first pivot chosen from $z_{ijk}\}$

+ Probability { z_j is the first pivot chosen from z_{ijk} }

$$= \frac{1}{|Z_{ijk}|} + \frac{1}{|Z_{ijk}|}$$

$$E[X_{ijk}] = \frac{2}{\max((j-i+1),(j-k+1),(k-i+1))} \quad \ldots\ldots (1)$$

Step 2 of 4

b. Consider that X_k denote sum of the comparisons between elements of array A when finding Z_k.

Total number of comparisons that is Z_k can be given by following summation,

$$X_k = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ijk}$$

By linearity of expectation,

$$E[X_k] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ijk}\right]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ijk}]$$

Substituting the value of $E[X_{ijk}]$ from equation **(1)** results in,

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{\max((j-i+1),(j-k+1),(k-i+1))}$$

In order to solve the $\max$ function its needed to divide the summation condition in such a way as in that range there will be only one maximum out of the three.

That is; the sum could be break in three cases:

1. $i \le k \le j$
2. $k < i$
3. $j < k$

Breaking as above the result will be,

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

Further simplifying it results in,

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

Hence,

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

Step 3 of 4

c. Since, $\frac{j-k-1}{j-k+1}$ is a fraction that is strictly less than 1. Same way $\frac{k-i-1}{k-i+1}$ is a fraction strictly less than 1.

Therefore,

$\sum_{j=k+1}^{n} \frac{j-k-1}{j-k+1}$ is summation of the fraction less than 1 for $n-k$ times ... ... **(3)**

$\sum_{i=1}^{k-2} \frac{k-i-1}{k-i+1}$ is summation of the fraction less than 1 for $k-2$ times ... ... **(4)**

Form **(3)** and **(4)**,

$\sum_{j=k+1}^{n} \frac{j-k-1}{j-k+1} + \sum_{i=1}^{k-2} \frac{k-i-1}{k-i+1}$ Will always be less than n ... ... **(5)**

Now, consider the first part of summation from equation **(2)**,

$$\sum_{i=1}^{k} \sum_{j=k}^{n} \frac{1}{j-i+1}$$

Assume that $j-i=t$. It is obvious to get that there can be at max $t+1$ ways for $j-i$ to be t.

Therefore it can be rewritten as

$$\sum_{t=0}^{n-1} \frac{t+1}{t+1} = n \text{ ... ... } \textbf{(6)}$$

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

From **(5)** and **(6)**,

$$E\left[X_k\right] \le 2(n+n)$$

$$E\left[X_k\right] \le 4n$$

Hence Proved.

Step 4 of 4

d. To prove the following, assuming all elements of array *A* are distinct, RANDOMIZED-SELECT executes in the assumed time $O(n)$.

Proof:

In order to prove that RANDOMIZED-SELECT executes in primarily assumed time $O(n)$, there is need to apply the Lemma 7.1 for RANDOMIZED-SELECT. To do this replace the variable *X* in the Lemma 7.1(Refer PAGE 182) by the randomly selected variable X_k as mentioned above.

There fore, the total execution time of RANDOMIZED-SELECT as expected is $O(n+X_k)$ which can be represented as $O(n)$.

Hence, the running time of the RANDOMIZED-SELECT is $O(n)$ for the distinct elements of array *A*.