CHAPTER 12

SELECTION ALGORITHMS [MEDIANS]

12.1 What are Selection Algorithms?

Selection algorithm is an algorithm for finding the k^{th} smallest/largest number in a list (also called as k^{th} order statistic). This includes finding the minimum, maximum, and median elements. For findingthe k^{th} order statistic, there are multiple solutions which provide different complexities, and in this chapter we will enumerate those possibilities.

12.2 Selection by Sorting

A selection problem can be converted to a sorting problem. In this method, we first sort the input elements and then get the desired element. It is efficient if we want to perform many selections.

For example, let us say we want to get the minimum element. After sorting the input elements we can simply return the first element (assuming the array is sorted in ascending order). Now, if we want to find the second smallest element, we can simply return the second element from the sorted list.

That means, for the second smallest element we are not performing the sorting again. The same is also the case with subsequent queries. Even if we want to get k^{th} smallest element, just one scan of the sorted list is enough to find the element (or we can return the k^{th}-indexed value if the elements are in the array).

From the above discussion what we can say is, with the initial sorting we can answer any query in one scan, $O(n)$. In general, this method requires $O(nlogn)$ time (for *sorting*), where n is the length of the input list. Suppose we are performing n queries, then the average cost per operation is just $\frac{n\,logn}{n} \approx O(logn)$. This kind of analysis is called *amortized* analysis.

12.3 Partition-based Selection Algorithm

For the algorithm check Problem-6. This algorithm is similar to Quick sort.

12.4 Linear Selection Algorithm - Median of Medians Algorithm

Worst-case performance	$O(n)$
Best-case performance	$O(n)$
Worst-case space complexity	$O(1)$ auxiliary

Refer to Problem-11.

12.5 Finding the K Smallest Elements in Sorted Order

For the algorithm check Problem-6. This algorithm is similar to Quick sort.

12.6 Selection Algorithms: Problems & Solutions

Problem-1 Find the largest element in an array A of size n.

Solution: Scan the complete array and return the largest element.

```
def FindLargestInArray(A):
    max = 0
    for number in A:
        if number > max:
            max = number
    return max
print(FindLargestInArray([2,1,5,234,3,44,7,6,4,5,9,11,12,14,13]))
```

Time Complexity - O(n). Space Complexity - O(1).

Note: Any deterministic algorithm that can find the largest of n keys by comparison of keys takes at least $n - 1$ comparisons.

Problem-2 Find the smallest and largest elements in an array A of size n.

Solution:

```
def FindSmallestAndLargestInArray(A):
    max = 0
    min = 0
    for number in A:
            if number > max:
                    max = number
            elif number < min:
                    min = number
    print("Smallest: %d", min)
    print("Largest: %d", max )
FindSmallestAndLargestInArray([2,1,5,234,3,44,7,6,4,5,9,11,12,14,13])
```

Time Complexity - O(n). Space Complexity - O(1). The worst-case number of comparisons is $2(n - 1)$.

Problem-3 Can we improve the previous algorithms?

Solution: Yes. We can do this by comparing in pairs.

```
def findMinMaxWithPairComparisons(A):
    ## for an even-sized Aray
    _max = A[0]
    _min = A[0]
    for indx in range(0, len(A), 2):
        first = A[indx]
        second = A[indx+1]
        if (first < second):
            if first < _min: _min = first
            if second > _max: _max = second
        else:
            if second < _min: _min = second
            if first > _max: _max = first
    print(_min)
    print(_max)
findMinMaxWithPairComparisons([2,1,5,234,3,44,7,6,4,5,9,11,12,14,13,19])
```

Time Complexity – O(n). Space Complexity – O(1).

Number of comparisons: $\begin{cases} \frac{3n}{2} - 2, if\ n\ is\ even \\ \frac{3n}{2} - \frac{3}{2}\ if\ n\ is\ odd \end{cases}$

Summary:

Straightforward comparison – $2(n - 1)$ comparisons
Compare for min only if comparison for max fails

Best case: increasing order – $n - 1$ comparisons
Worst case: decreasing order – $2(n - 1)$ comparisons
Average case: $3n/2 - 1$ comparisons

Note: For divide and conquer techniques refer to *Divide and Conquer* chapter.

Problem-4 Give an algorithm for finding the second largest element in the given input list of elements.

Solution: Brute Force Method

Algorithm:

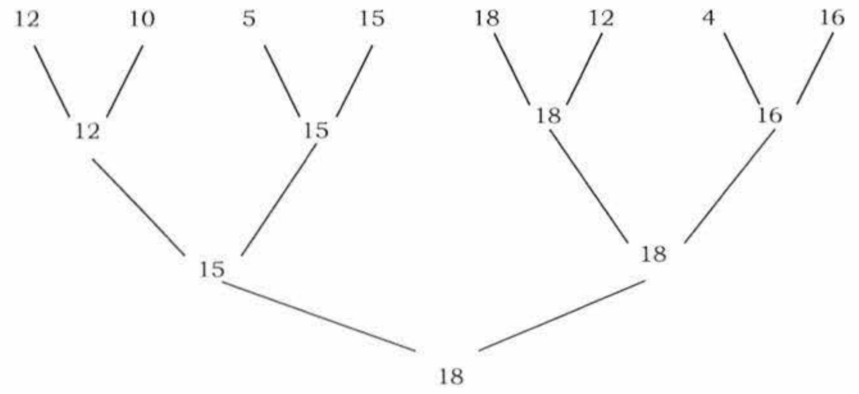
- Find largest element: needs $n - 1$ comparisons
- Delete (discard) the largest element
- Again find largest element: needs $n - 2$ comparisons

Total number of comparisons: $n - 1 + n - 2 = 2n - 3$

Problem-5 Can we reduce the number of comparisons in Problem-4 solution?

Solution: The Tournament method: For simplicity, assume that the numbers are distinct and that n is a power of 2. We pair the keys and compare the pairs in rounds until only one round remains. If the input has eight keys, there are four comparisons in the first round, two in the second, and one in the last. The winner of the last round is the largest key. The figure below shows the method.

The tournament method directly applies only when n is a power of 2. When this is not the case, we can add enough items to the end of the array to make the array size a power of 2. If the tree is complete then the maximum height of the tree is $logn$. If we construct the complete binary tree, we need $n - 1$ comparisons to find the largest. The second largest key has to be among the ones that were lost in a comparison with the largest one. That means, the second largest element should be one of the opponents of the largest element. The number of keys that are lost to the largest key is the height of the tree, i.e. $logn$ [if the tree is a complete binary tree]. Then using the selection algorithm to find the largest among them, take $logn - 1$ comparisons. Thus the total number of comparisons to find the largest and second largest keys is $n + logn - 2$.



```
def secondSmallestInArray(A):
    comparisonCount = 0
    # indexes that are to be compared
    idx = range(0,len(A))
    # list of knockout for all elements
    knockout = [[] for i in idx]
    # play tournaments, until we have only one node left
    while len(idx) > 1:
        # index of nodes that win this tournament
        idx1 = []
        # nodes in idx odd, if yes then last automatically goes to next round
        odd = len(idx) % 2
        # iterate over even indexes, as we do a paired tournament
        for i in xrange(0, len(idx) - odd, 2):
            firstIndex = idx[i]
            secondIndex = idx[i+1]
            comparisonCount += 1
            # perform tournament
```

```
                if A[firstIndex] < A[secondIndex]:
                    # firstIndex qualifies for next round
                    idx1.append(firstIndex)
                    # add A[secondIndex] to knockout list of firstIndex
                    knockout[firstIndex].append(A[secondIndex])
                else:
                    idx1.append(secondIndex)
                    knockout[secondIndex].append(A[firstIndex])
            if odd == 1:
                idx1.append(idx[i+2])
            # perform new tournament
            idx = idx1
        print "Smallest element =", A[idx[0]]
        print "Total comparisons =", comparisonCount
        print "Nodes knocked off by the smallest =", knockout[idx[0]], "\n"
        # compute second smallest
        a = knockout[idx[0]]
        if len(a) > 0:
            v = a[0]
            for i in xrange(1,len(a)):
                comparisonCount += 1
                if v > a[i]: v = a[i]
            print "Second smallest element =", v
            print "Total comparisons =", comparisonCount
    A = [2, 4, 3, 7, 3, 0, 8, 4, 11, 1]
    print(secondSmallestInArray(A))
```

Problem-6 Find the k-smallest elements in an array S of n elements using partitioning method.

Solution: Brute Force Approach: Scan through the numbers k times to have the desired element. This method is the one used in bubble sort (and selection sort), every time we find out the smallest element in the whole sequence by comparing every element. In this method, the sequence has to be traversed k times. So the complexity is $O(n \times k)$.

Problem-7 Can we use the sorting technique for solving Problem-6?

Solution: Yes. Sort and take the first k elements.

1. Sort the numbers.
2. Pick the first k elements.

The time complexity calculation is trivial. Sorting of n numbers is of $O(nlogn)$ and picking k elements is of $O(k)$. The total complexity is $O(nlogn + k) = O(nlogn)$.

Problem-8 Can we use the *tree sorting* technique for solving Problem-6?

Solution: Yes.

1. Insert all the elements in a binary search tree.
2. Do an InOrder traversal and print k elements which will be the smallest ones. So, we have the k smallest elements.

The cost of creation of a binary search tree with n elements is $O(nlogn)$ and the traversal up to k elements is $O(k)$. Hence the complexity is $O(nlogn + k) = O(nlogn)$.

Disadvantage: If the numbers are sorted in descending order, we will be getting a tree which will be skewed towards the left. In that case, the construction of the tree will be $0 + 1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2}$ which is $O(n^2)$. To escape from this, we can keep the tree balanced, so that the cost of constructing the tree will be only $nlogn$.

Problem-9 Can we improve the *tree sorting* technique for solving Problem-6?

Solution: Yes. Use a smaller tree to give the same result.

1. Take the first k elements of the sequence to create a balanced tree of k nodes (this will cost $klogk$).
2. Take the remaining numbers one by one, and
    a. If the number is larger than the largest element of the tree, return.

b. If the number is smaller than the largest element of the tree, remove the largest element of the tree and add the new element. This step is to make sure that a smaller element replaces a larger element from the tree. And of course the cost of this operation is $logk$ since the tree is a balanced tree of k elements.

Once Step 2 is over, the balanced tree with k elements will have the smallest k elements. The only remaining task is to print out the largest element of the tree.

Time Complexity:

1. For the first k elements, we make the tree. Hence the cost is $klogk$.
2. For the rest $n - k$ elements, the complexity is O($logk$).

Step 2 has a complexity of $(n - k)\, logk$. The total cost is $klogk + (n - k)\, logk = nlogk$ which is O($nlogk$). This bound is actually better than the ones provided earlier.

Problem-10 Can we use the partitioning technique for solving Problem-6?

Solution: Yes.

Algorithm

1. Choose a pivot from the array.
2. Partition the array so that: $A[low...pivotpoint - 1] <= pivotpoint <= A[pivotpoint + 1..high]$.
3. if $k < pivotpoint$ then it must be on the left of the pivot, so do the same method recursively on the left part.
4. if $k = pivotpoint$ then it must be the pivot and print all the elements from low to $pivotpoint$.
5. if $k > pivotpoint$ then it must be on the right of pivot, so do the same method recursively on the right part.

The input data can be any iterable. The randomization of pivots makes the algorithm perform consistently even with unfavorable data orderings.

```
import random
def kthSmallest(data, k):
    "Find the nth rank ordered element (the least value has rank 0)."
    data = list(data)
    if not 0 <= k < len(data):
        raise ValueError('not enough elements for the given rank')
    while True:
        pivot = random.choice(data)
        pcount = 0
        under, over = [], []
        uappend, oappend = under.append, over.append
        for elem in data:
            if elem < pivot:
                uappend(elem)
            elif elem > pivot:
                oappend(elem)
            else:
                pcount += 1
        if k < len(under):
            data = under
        elif k < len(under) + pcount:
            return pivot
        else:
            data = over
            k -= len(under) + pcount
print(kthSmallest([2,1,5,234,3,44,7,6,4,5,9,11,12,14,13], 5))
```

Time Complexity: O(n^2) in worst case as similar to Quicksort. Although the worst case is the same as that of Quicksort, this performs much better on the average [O($nlogk$) – Average case].

Problem-11 Find the k^{th}-smallest element in an array S of n elements in best possible way.

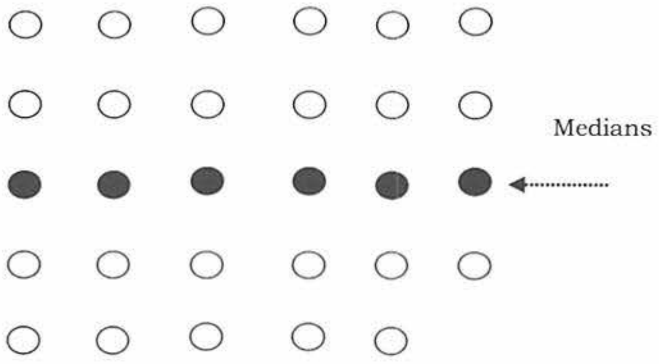
Solution: This problem is similar to Problem-6 and all the solutions discussed for Problem-6 are valid for this problem. The only difference is that instead of printing all the k elements, we print only the k^{th}element. We can improve the solution by using the *median of medians* algorithm. Median is a special case of the selection algorithm. The algorithm Selection(A, k) to find the k^{th} smallest element from set A of n elements is as follows:

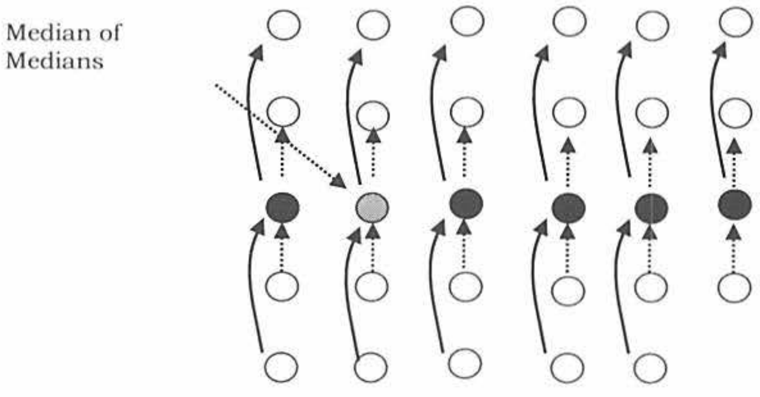
Algorithm: *Selection*(A, k)

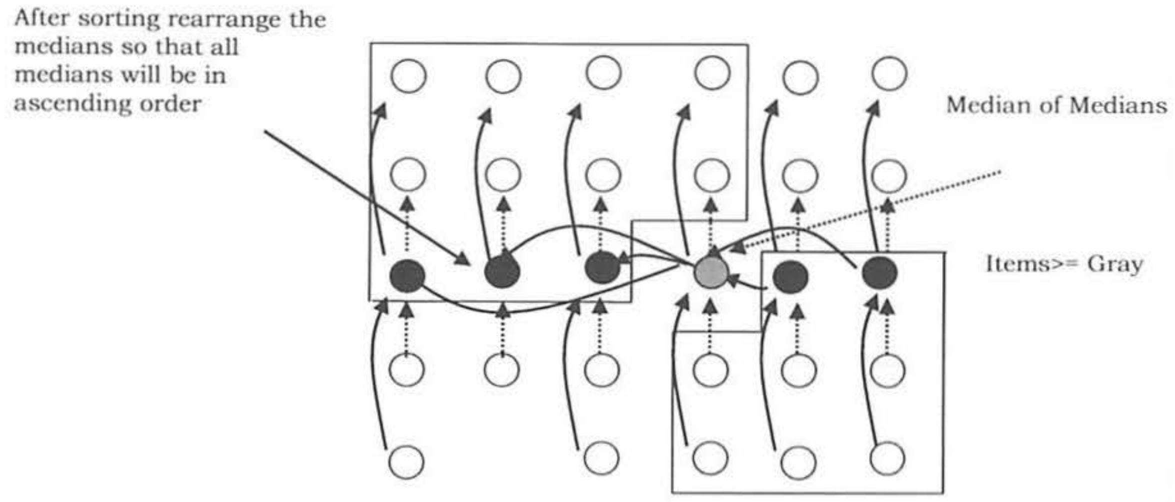
```
1.  Partition $A$ into $ceil\left(\frac{length(A)}{5}\right)$ groups, with each group having five items (the last group may have fewer
    items).
2.  Sort each group separately (e.g., insertion sort).
3.  Find the median of each of the $\frac{n}{5}$ groups and store them in some array (let us  say $A'$).
4.  Use Selection recursively to find the median of $A'$ (median of medians). Let us asay the median of
    medians is $m$.
            $m = Selection(A', \frac{\frac{length(A)}{5}}{2});$
5.  Let $q$ = # elements of $A$ smaller than $m$;
6.  If($k == q + 1$)
        return $m$;
    /* Partition with pivot */
7.  Else partition $A$ into $X$ and $Y$
        • $X$ = {items smaller than $m$}
        • $Y$ = {items larger than $m$}

    /* Next,form a subproblem */
8.  If($k < q + 1$)
        return Selection(X, k);
9.  Else
        return Selection(Y, k – (q+1));
```

Before developing recurrence, let us consider the representation of the input below. In the figure, each circle is an element and each column is grouped with 5 elements. The black circles indicate the median in each group of 5 elements. As discussed, sort each column using constant time insertion sort.






After sorting rearrange the medians so that all medians will be in ascending order
Median of Medians
Items>= Gray

In the figure above the gray circled item is the median of medians (let us call this m). It can be seen that at least 1/2 of 5 element group medians $\leq m$. Also, these 1/2 of 5 element groups contribute 3 elements that are $\leq m$ except 2 groups [last group which may contain fewer than 5 elements, and other group which contains m]. Similarly, at least 1/2 of 5 element groups contribute 3 elements that are $\geq m$ as shown above. 1/2 of 5 element groups contribute 3 elements, except 2 groups gives: $3(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2) \approx \frac{3n}{10} - 6$. The remaining are $n - \frac{3n}{10} - 6 \approx \frac{7n}{10} + 6$. Since $\frac{7n}{10} + 6$ is greater than $\frac{3n}{10} - 6$ we need to consider $\frac{7n}{10} + 6$ for worst.

Components in recurrence:

- In our selection algorithm, we choose m, which is the median of medians, to be a pivot, and partition A into two sets X and Y. We need to select the set which gives maximum size (to get the worst case).
- The time in function *Selection* when called from procedure *partition*. The number of keys in the input to this call to *Selection* is $\frac{n}{5}$.
- The number of comparisons required to partition the array. This number is $length(S)$, let us say n.

We have established the following recurrence: $T(n) = T\left(\frac{n}{5}\right) + \Theta(n) + Max\{T(X), T(Y)\}$

From the above discussion we have seen that, if we select median of medians m as pivot, the partition sizes are: $\frac{3n}{10} - 6$ and $\frac{7n}{10} + 6$. If we select the maximum of these, then we get:

$$\begin{aligned} T(n) &= T\left(\frac{n}{5}\right) + \Theta(n) + T\left(\frac{7n}{10} + 6\right) \\ &\approx T\left(\frac{n}{5}\right) + \Theta(n) + T\left(\frac{7n}{10}\right) + O(1) \\ &\leq c\frac{7n}{10} + c\frac{n}{5} + \Theta(n) + O(1) \\ &\text{Finally, } T(n) = \Theta(n). \end{aligned}$$

```
CHUNK_SIZE = 5
def kthByMedianOfMedian(unsortedList, k):
    if len(unsortedList) <= CHUNK_SIZE:
        return get_kth(unsortedList, k)
    chunks = splitIntoChunks(unsortedList, CHUNK_SIZE)
    medians_list = []
    for chunk in chunks:
        median_chunk = get_median(chunk)
        medians_list.append(median_chunk)
    size = len(medians_list)
    mom = kthByMedianOfMedian(medians_list, size / 2 + (size % 2))
    smaller, larger = splitListByPivot(unsortedList, mom)
    valuesBeforeMom = len(smaller)
    if valuesBeforeMom == (k - 1):
        return mom
    elif valuesBeforeMom > (k - 1):
        return kthByMedianOfMedian(smaller, k)
    else:
        return kthByMedianOfMedian(larger, k - valuesBeforeMom - 1)
```

Problem-12 In Problem-11, we divided the input array into groups of 5 elements. The constant 5 play an important part in the analysis. Can we divide in groups of 3 which work in linear time?

Solution: In this case the modification causes the routine to take more than linear time. In the worst case, at least half of the $\lceil \frac{n}{3} \rceil$ medians found in the grouping step are greater than the median of medians m, but two of those groups contribute less than two elements larger than m. So as an upper bound, the number of elements larger than the pivotpoint is at least:

$$2(\lceil \frac{1}{2}\lceil \frac{n}{3} \rceil\rceil - 2) \geq \frac{n}{3} - 4$$

Likewise this is a lower bound. Thus up to $n - (\frac{n}{3} - 4) = \frac{2n}{3} + 4$ elements are fed into the recursive call to *Select*. The recursive step that finds the median of medians runs on a problem of size $\lceil \frac{n}{3} \rceil$, and consequently the time recurrence is:

$$T(n) = T(\lceil \frac{n}{3} \rceil) + T(2n/3 + 4) + \Theta(n).$$

Assuming that $T(n)$ is monotonically increasing, we may conclude that $T(\frac{2n}{3} + 4) \geq T(\frac{2n}{3}) \geq 2T(\frac{n}{3})$, and we can say the upper bound for this as $T(n) \geq 3T(\frac{n}{3}) + \Theta(n)$, which is $O(nlogn)$. Therefore, we cannot select 3 as the group size.

Problem-13 As in Problem-12, can we use groups of size 7?

Solution: Following a similar reasoning, we once more modify the routine, now using groups of 7 instead of 5. In the worst case, at least half the $\lceil \frac{n}{7} \rceil$ medians found in the grouping step are greater than the median of medians m, but two of those groups contribute less than four elements larger than m. So as an upper bound, the number of elements larger than the pivotpoint is at least:

$$4(\lceil 1/2 \lceil n/7 \rceil\rceil - 2) \geq \frac{2n}{7} - 8.$$

Likewise this is a lower bound. Thus up to $n - (\frac{2n}{7} - 8) = \frac{5n}{7} + 8$ elements are fed into the recursive call to Select. The recursive step that finds the median of medians runs on a problem of size $\lceil \frac{n}{7} \rceil$, and consequently the time recurrence is

$$\begin{aligned}
T(n) &= T(\lceil \frac{n}{7} \rceil) + T(\frac{5n}{7} + 8) + O(n) \\
T(n) &\leq c\lceil \frac{n}{7} \rceil + c(\frac{5n}{7} + 8) + O(n) \\
&\leq c\frac{n}{7} + c\frac{5n}{7} + 8c + an, a \text{ is a constant} \\
&= cn - c\frac{n}{7} + an + 9c \\
&= (a + c)n - (c\frac{n}{7} - 9c).
\end{aligned}$$

This is bounded above by $(a + c)n$ provided that $c\frac{n}{7} - 9c \geq 0$. Therefore, we can select 7 as the group size.

Problem-14 Given two arrays each containing n sorted elements, give an $O(logn)$-time algorithm to find the median of all $2n$ elements.

Solution: The simple solution to this problem is to merge the two lists and then take the average of the middle two elements (note the union always contains an even number of values). But, the merge would be $\Theta(n)$, so that doesn't satisfy the problem statement. To get *logn* complexity, let *medianA* and *medianB* be the medians of the respective lists (which can be easily found since both lists are sorted). If *medianA == medianB*, then that is the overall median of the union and we are done. Otherwise, the median of the union must be between *medianA* and *medianB*. Suppose that *medianA < medianB* (the opposite case is entirely similar). Then we need to find the median of the union of the following two sets:

$$\{x \text{ in } A \mid x >= medianA\} \{x \text{ in } B \mid x <= medianB\}$$

So, we can do this recursively by resetting the *boundaries* of the two arrays. The algorithm tracks both arrays (which are sorted) using two indices. These indices are used to access and compare the median of both arrays to find where the overall median lies.

```
def findKthSmallest(A, B, k):
    if len(A) > len(B):        A, B = B, A
    # stepsA = (endIndex + beginIndex_as_0) / 2
    stepsA = (min(len(A), k) -1)/ 2
    # stepsB = k - (stepsA + 1) -1 for the 0-based index
    stepsB = k - stepsA - 2
    # Only array B contains elements
    if len(A) == 0:                 return B[k-1]
    # Both A and B contain elements, and we need the smallest one
    elif k == 1:                    return min(A[0], B[0])
    # The median would be either A[stepsA] or B[stepsB], while A[stepsA] and
    # B[stepsB] have the same value.
    elif A[stepsA] == B[stepsB]:    return A[stepsA]
    # The median must be in the right part of B or left part of A
    elif A[stepsA] > B[stepsB]:     return findKthSmallest(A, B[stepsB+1:], k-stepsB-1)
    # The median must be in the right part of A or left part of B
    else: return findKthSmallest(A[stepsA+1:], B, k-stepsA-1)
def findMedianSortedArrays(A, B):
        # There must be at least one element in these two arrays
        assert not(len(A) == 0 and len(B) == 0)
        if (len(A)+len(B))%2==1:
            # There are odd number of elements in total. The median the one in the middle
            return findKthSmallest(A, B, (len(A)+len(B))/2+1) * 1.0
        else:
            # There are even number of elements in total. The median the mean value of the
            # middle two elements.
            return ( findKthSmallest(A, B, (len(A)+len(B))/2+1) + findKthSmallest(A, B, (len(A)+len(B))/2) ) / 2.0
A = [127, 220, 246, 277, 321, 454, 534, 565, 933]
B = [12, 22, 24, 27, 32, 45, 53, 65, 93]
print(findMedianSortedArrays(A,B))
```

Time Complexity: O($logn$), since we are reducing the problem size by half every time.

Problem-15 Let A and B be two sorted arrays of n elements each. We can easily find the k^{th} smallest element in A in O(1) time by just outputting $A[k]$. Similarly, we can easily find the k^{th} smallest element in B. Give an O($logk$) time algorithm to find the k^{th} smallest element overall { *i.e.*, the k^{th} smallest in the union of A and B.

Time Complexity: O($logn$), since we are reducing the problem size by half every time.

Problem-16 Let A and B be two sorted arrays of n elements each. We can easily find the k^{th} smallest element in A in O(1) time by just outputting $A[k]$. Similarly, we can easily find the k^{th} smallest element in B. Give an O($logk$) time algorithm to find the k^{th} smallest element overall { *i.e.*, the k^{th} smallest in the union of A and B.

Solution: It's just another way of asking Problem-14.

Problem-17 **Find the k smallest elements in sorted order:** Given a set of n elements from a totally-ordered domain, find the k smallest elements, and list them in sorted order. Analyze the worst-case running time of the best implementation of the approach.

Solution: Sort the numbers, and list the k smallest.

$T(n)$ = Time complexity of sort + listing k smallest elements = $\Theta(nlogn) + \Theta(n) = \Theta(nlogn)$.

Problem-18 For Problem-17, if we follow the approach below, then what is the complexity?

Solution: Using the priority queue data structure from heap sort, construct a min-heap over the set, and perform extract-min k times. Refer to the *Priority Queues* (*Heaps*) chapter for more details.

Problem-19 For Problem-17, if we follow the approach below then what is the complexity?
Find the k^{th}-smallest element of the set, partition around this pivot element, and sort the k smallest elements.

Solution:

$$T(n) = Time\ complexity\ of\ kth - smallest\ +\ Finding\ pivot\ +\ Sorting\ prefix$$
$$= \Theta(n) + \Theta(n) + \Theta(klogk) = \Theta(n + klogk)$$

Since, $k \le n$, this approach is better than Problem-17 and Problem-18.

Problem-20 Find k nearest neighbors to the median of n distinct numbers in O(n) time.

Solution: Let us assume that the array elements are sorted. Now find the median of n numbers and call its index as X (since array is sorted, median will be at $\frac{n}{2}$ location). All we need to do is select k elements with the smallest absolute differences from the median, moving from $X-1$ to 0, and $X+1$ to $n-1$ when the median is at index m.

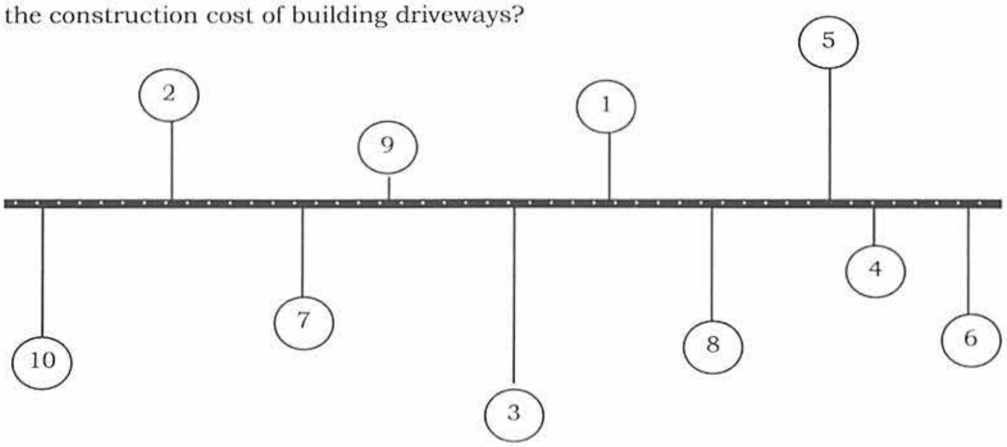
Time Complexity: Each step takes $\Theta(n)$. So the total time complexity of the algorithm is $\Theta(n)$.

Problem-21 Is there any other way of solving Problem-20?

Solution: Assume for simplicity that n is odd and k is even. If set A is in sorted order, the median is in position $n/2$ and the k numbers in A that are closest to the median are in positions $(n - k)/2$ through $(n + k)/2$.

We first use linear time selection to find the $(n - k)/2$, $n/2$, and $(n + k)/2$ elements and then pass through set A to find the numbers less than the $(n+k)/2$ element, greater than the $(n-k)/2$ element, and not equal to the $n/2$ element. The algorithm takes $O(n)$ time as we use linear time selection exactly three times and traverse the n numbers in A once.

Problem-22 Given (x, y) coordinates of n houses, where should you build a road parallel to x-axis to minimize the construction cost of building driveways?



Solution: The road costs nothing to build. It is the driveways that cost money. The driveway cost is proportional to its distance from the road. Obviously, they will be perpendicular. The solution is to put the street at the median of the y coordinates.

Problem-23 Given a big file containing billions of numbers, find the maximum 10 numbers from that file.

Solution: Refer to the *Priority Queues* chapter.

Problem-24 Suppose there is a milk company. The company collects milk every day from all its agents. The agents are located at different places. To collect the milk, what is the best place to start so that the least amount of total distance is travelled?

Solution: Starting at the median reduces the total distance travelled because it is the place which is at the center of all the places.