```
/*************************************************************
 * EECS2011: Fundamentals of Data Structures, Fall 2016
 * Assignment 4, a4sol.pdf
 * Student Name: Khushal Patel
 * Student CSE account: york18
 * Student ID number: 214037618
 *************************************************************/
```

Q1: kth smallest element

a) For k = 6, output will be **16**
For k = 10, output will be **35**

b) Algorithm:

1) Calculate the medians m1 and m2 of the input arrays ar1[] and ar2[] respectively.

2) If m1 and m2 both are equal then we are done.
return m1 (or m2)

3) If m1 is greater than m2, then median is present in one of the below two subarrays.

a) From first element of ar1 to m1 (ar1[0...|_k/2_|])
b) From m2 to last element of ar2 (ar2[|_k/2_|...k-1])

4) If m2 is greater than m1, then median is present in one of the below two subarrays.

a) From m1 to last element of ar1 (ar1[|_k/2_|...k-1])
b) From first element of ar2 to m2 (ar2[0...|_k/2_|])

5) Repeat the above process until size of both the subarrays becomes 2.

6) If size of the two arrays is 2 then use below formula to get the median.
Median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2

c) **Basic Idea**: Name the arrays a and b. Obviously we can ignore all a[i] and b[i] where i > k. First let's compare a[k/2] and b[k/2]. Let b[k/2] > b[k/2]. Therefore, we can discard also all b[i], where i > k/2. Now we have all a[i], where i < k and all b[i], where i < k/2.

Pseudo-code:

```
i = k/2
j = k - i
loop = k/4
while loop > 0
   if a[i-1] > b[j-1]
      i -= loop
      j += loop
   else
      i += loop
      j -= loop
   loop /= 2

if a[i-1] > b[j-1]
   return a[i-1]
else
   return b[j-1]
```

Running Time: Since length of arrays a and b are > k, the complexity here is O(log k), which is O(log a.length + log b.length).

Q2: countRange

Basic Idea: We can use AVL trees to solve this problem that has an augmented field size at each node v which stores the number of nodes in a subtree rooted at v (including itself). Therefore, size(v) is the size of subtree rooted at v.

Algorithm:

```
countRange(k1, k2)
{
        AllinRangefromNode(root, k1, k2)
}
```

```
AllinRangefromNode(v, k1, k2)
{
        if v = NULL return 0
        if (v.key < k1) return AllinRangefromNode(rightchild(v), k1, k2)
        if (v.key > k1) return AllinRangefromNode(leftchild(v), k1, k2)
        else return 1+ countAllBiggerThan(leftchild(v), k1) + countAllSmallerThan(rightchild(v),
        k2)
}
countAllBiggerThan(v, k)
{
        if v = NULL return 0
        if (v.key < k) return countAllBiggerThan(rightchild(v), k)
        else return 1 + countAllBiggerThan(leftchild(v), k) + size(rightchild(v))
}
countAllSamllerThan(v, k)
{
if v = NULL return 0
if (v.key > k) return countAllSmallerThan(leftchild(v), k)
else return countAllBiggerThan(rightchild(v), k) + size(leftchild(v))
}
```

Correctness for AllinRangefromNode(v, k1, k2):

If v.key < k1 then we ignore the elements in the left subtree of v and continue with the same query to the right child of v. If k1 <= v.key <= k2 then v should be counted, and for the two subtrees of v we only need to count the elements that are bigger than k1 in the left subtree, and smaller than k2 in the right subtree.

For countAllBiggerThan(v, k) and countAllSmallerThan(v, k):

We analyze the countAllBiggerThan method (the other can be proved similarly). If the condition in line 2 holds then we can avoid looking at the left subtree as all elements there are, are clearly

too small. Otherwise, all elements of the right subtree must be counted, the root must be counted and in addition all sufficiently large elements in the right subtree should, hence the recursion.

Running Time: We can bound the running time of this method by the height of the tree, that is O(logn) plus the running time of the other countAllBiggerThan and countAllSmallerThan method. This is since every time the conditions in line 2 or line 3 of AllinRangefromNode are satisfied in the recursion tree, we move to a deeper node. Hence there can be as many recursive calls to AllinRangefromNode as the height of the tree. i.e., O(h).

Q3 Voting Problem

Basic Idea: We are given a sequence S of n-elements where each element in that sequence represents a vote. We loop through this sequence and create an AVL Tree (each node is key-value pair) such that while looping we search for the current element in S in our AVL Tree (by looking at all the keys). If we can't find that element then we insert a new key-value pair in the AVL tree, key being the candidate and value being the vote, which is initially 1. If we do find that element in the tree, we just increment it's value by 1. After the loop ends, we search for an element of the AVL tree whose value is maximum and we return the corresponding key.

Pseudocode: Name the AVL tree as AVL and the array of sequence S. And say if S[i] is representing a vote for a candidate then function f(S[i]) represents the candidate

getResult(sequence S)

```
i = 0
while(i < S.length)
        if (AVL.search(f(S[i])) == null) then
                insert (f(S[i]), 1)
        else
                new = AVL.get(f(S[i])).value++
                AVL.get(f(S[i])).setValue = new
        i = i + 1
        end if
```

```
end while
Map.Entry<Integer, Integer> maxEntry = null;
for each Map.Entry<Integer, Integer> entry in AVL.entrySet
   if (maxEntry == null || entry.getValue().compareTo(maxEntry.getValue()) > 0)
   then
      maxEntry = entry;
  end if
end for
return maxEntry
end getResult
```

Running Time: Since we know that candidates running for president are k < n. The algorithm will take O(n log k) running time.