Assignment 1

Problem 1:

Part(a):

Algorithm:

```
FindElement(arr[],l,r){
if(r<l)
        return NULL;
mid = (l+r)/2;
if(arr[mid] == mid)
        return mid;
else if(arr[mid] > mid)
        return FindElement(arr[],l,mid-1);
else
        return FindElement(arr[],mid+1,r);
```

Correctness:

The correctness can be proved using induction on the length of the array.The P(n) be the hypothesis that FindElement returns the correct index i, such that A[i] = i.
The base case is n=1.For this if arr[1] = 1, then the algorithm will return 1, else it will return NULL by performing just one more recursive call and making r<l in the next call.
Now suppose this is true for all lengths of array 1,.....,n-1.

Now, the algorithm first checks if r<l. This is the case where we have performed every relevant comparison and size of the array that we have to compare has become less than 1.
Now we calculate the index of the mid element and check whether arr[mid]==mid, then we successfully return the index mid.
If this is not the case, then the correctness of the algorithm can be proved by the following claims:

Claim 1:

If(arr[mid]> mid), then the element in the array such that arr[i] == i, cannot be present in the latter half of the array. That is, this element(if present) cannot be on the indexes from mid+1 to r.

Proof:

Now since the array is sorted,all the elements after the middle element will be greater than the middle element. Also since the elements in the array are distinct, no two elements can be the same.

Hence, any element following the middle element will at least be 1 greater than the previous element.
Since arr[mid]>mid,then arr[mid] - mid > 0; ---------- (i)
Now using the above argument,
(arr[mid+1] - 1) >= arr[mid];
Subtracting mid on both sides,
(arr[mid + 1]) - (mid+1) >= arr[mid] - mid; ----------(ii)
Equation 1 and Equation 2 together proof claim 1.

Claim 2:
If(arr[mid] < mid), then the element in the array such that arr[i] == i, cannot be present in the former half of the array. That is, this element(if present) cannot be on the indexes from l to mid-1.

Proof:
This can also be proved in a similar way as the proof of Claim 1;

The above two claims, along with the induction hypothesis proves the correctness of our algorithm.

Time Complexity:

Let T(n) be the number of steps taken by the algorithm FindElement on an array of size n.Some comparison, which can be done in constant time, and only one recursive call on either part of the array. The recursive call takes T(n/2) steps(since the size of the array on which we perform the recursive call is n/2).
Hence T(n) satisfies:
T(n) = T(n/2) + O(1).
This is the same as binary search and solves to.

T(n) = O(log n).

Part(b):

Algorithm:

```
FindElement(arr[]):
    if(arr[1] == 1)
        return 1;
    else
        return NULL;
```

Correctness:
In the above algorithm if arr[1] == 1, then it correctly reports the index 1.

Now, if this is not the case and we already know that A[1] > 0 then it means that A[1] > 1.Now every element after the first element will be greater than it's index, that is A[i] - i > 0. This can be proved similarly to the proof of Claim 1 in part(a) above.
Hence this proves the correctness of our algorithm.

Time Complexity:

Let T(n) be the time complexity for an array of size n.Since the above algorithm only runs one time and performs a constant number of steps,

T(n) = O(1).

Problem 2:

Part a)

Correctness:

The correctness can be proved by using induction on the length of the array.Let P(n) be the hypothesis that the algorithm CRUEL correctly sorts an array of input n.The base case is n=2 and in this case it simply swaps the elements if the first element is larger. So it is trivial.Now suppose this is true for all arrays of length 2…..n-1.

Consider P(n).Now the algorithm first applies the recursive calls on the first and second half of the array.

By the induction hypothesis, the first and second halves of the array are correctly sorted by the recursive calls.Now we prove the correctness of the UNUSUAL function.
The correctness of the UNUSUAL algorithm can be proved by the following claim:

Claim:

The UNUSUAL procedure correctly sorts the array of length n, given both the halves A[1….n/2] and A[n/2+1 …. n] are already sorted.

Proof:

Let us prove this by induction.Let K(n) be the induction hypothesis that the algorithm UNUSUAL correctly sorts an array of input n given that both the halves A[1...n/2[and A[n/2+1 … n] are already sorted.The base case is n=2 and it simply swaps the elements if needed.Now, let us suppose this is true for all arrays of length 2….n-1.
Consider K(n). Now at the beginning of the execution of the algorithm the first and second half of the array are already sorted.
Now the array is divided in 4 quarters. 1st,2nd,3rd and 4th.
i)Since the first half is already sorted every element in 1st quarter < every element in 2nd quarter.
ii)Since the second half is also already sorted every element in 3rd quarter < every element in 4th quarter.

Hence, the above two statements imply that the minimum n/4 elements are present in the 1st and the 3rd quarter and the maximum n/4 elements are present in the 2nd and the 3rd quarter.

After the iteration of the first for loop in the UNUSUAL procedure, the second and the third quarters swapped.

Now the minimum n/4 elements are present in the first half of the array and the maximum n/4 elements are present in the second half of the array.
We apply recursive calls on the 1st half, then 2nd half of the array.
Our induction hypothesis proves that the above two recursive calls correctly sorts the 1st and 2nd half of the array given that all the quarters were already sorted.
Now after the first two recursive calls, the minimum and maximum n/4 elements are already in their place.
Now we apply the recursive call on the middle half of the array and again according to our induction hypothesis this recursive call also correctly sorts the middle half.Hence our array is sorted.

Hence the above claim along with the induction hypothesis proves the correctness of the CRUEL Algorithm.

Part d:

Time Complexity of UNUSUAL:

Let T(n) be the number of steps taken by the algorithm UNUSUAL on an array of size n.The algorithm does 4 kinds of work. Three recursive calls on first, second and the middle half of the array, and swapping the 2nd and 3rd quarter of the array.The three recursive calls take T(n/2) steps each and swapping is done in O(n) time.
Hence T(n) satisfies:
$T(n) = 3T(n/2) + O(n)$
which solves to $T(n) = O(n^{\log 3})$

Part e:

Time Complexity of CRUEL:
Let T(n) be the number of steps taken by the algorithm CRUEL on an array of size n.The algorithm does 3 kinds of work. Two recursive calls on the first and second half of the array which each take T(n/2) steps and call the UNUSUAL procedure which takes $O(n^{\log 3})$ time.
Hence T(n) satisfies:
$T(n) = 2T(n/2) + O(n^{\log 3})$
which solves to $T(n) = O(n^{\log 3})$

Problem 3:

Part a:

Algorithm:
//Note: findInversion function caclculates the number of inversions in an array.

```
findIntersection(P[],Q[])
        HashMap<Integer,Integer> HM = new HashMap<>();
        for(i=0;i < Q.size(); i++)
                HM.put(Q[i],i);
        int indexArray[] = new int[N];
        for(i=0; i<N; i++)
                indexArray[i] = HM.get(P[i]);
        return findInversion(indexArray,0,Q.size() - 1);
```

Correctness:
The correctness of the above algorithm can be proved by the following claims:

Claim 1:
For any pair of line segments (p(i), q(i)) and (p(j),q(j)) to intersect they should be present in the opposite order in the arrays.That is in the P array if rank(p(i)) < rank(p(j)) then rank(q(j)) < rank (q(i)).
Proof:
The above statement is quite self explanatory and it can easily be seen that it is true for all the cases.

Claim 2:
The findInversion function correctly returns the number of intersections by counting the number of inversions in the array.
Proof:
Following claim 1, we made an index array that stores all the indexes of the q(i) with respect to p(i) at index i.Now let us take any pair of line segments say (p(i),q(i)) and (p(j), q(j)).If the rank(q(i) > rank(q(j)) when rank(p(i)) < rank(p(j)), the indexArray[i] > indexArray[j] for i<j(since rank(p(i) < rank(p(j)).
Hence we just need to find the number of inversions in the index array and that will be equal to the number of intersection points.

Claim1 and Claim2 together prove the correctness of the above algorithm.

Time Complexity:
Creating of the HashMap and the indexArray are done in O(n) time and the findInversion function executes in O(nlogn) time.
Hence the time complexity of the above algorithm is O(nlogn).

Part B:

Algorithm:

We first split the circle at any point and then count the points in any one direction.Say the points are {p1,q1,p4,p3,q2,q4,p2,q3}.

Now we will number the above points according to their relative order.For the above example, we will get ordering like, {(1,2),(3,6),(4,8),(5,7)}

Now the number of intersection points in the above example:

We will just start by looking at the start interval of the first point and if there is another start interval before the end interval of the first point we count that as one intersection.

We do this for the whole array.But if the start interval closes between an interval then that is not counted as an intersection.

For this we have to sort the start intervals and end intervals individually.

Since at each level we are sorting the start and end intervals.

That takes O(nlogn) time. And to find thestart interval it takes a binary search in the sorted array.

And one more search is required to find the end interval it takes another binary serach to check whether the end interval has ended between the it's super interval.

Hence $T(n) = 2T(n/2) + O(nlogn)$

which solves to $T(n) = O(nlog^2n)$