ADA Assignment 1

Ayushi Jain (2019031)
Kesar Shrivastava (2019051)

Problem 1

31. a) We need to find an algorithm that either computes an index i such that A[i] = i or reports if no such index exists. This can be done using the algorithm called binary search which is a form of decrease and conquer algorithm.

So we use binary search in the following way:

We keep two values in handy, l and h, they denote the leftmost and rightmost indices of the array respectively. Everytime we make a call we do the following steps:

1. If l>h then we return nothing as no such index exists.
2. Calculate the index that is in the middle of the array in consideration denoted by mid.
3. If A[mid] = mid, then we terminate the search and return the answer.
4. If A[mid] > mid, then we call the same function this time with l = l and h = mid-1.
5. If A[mid] < mid then we call the same function this time with l = mid+1 and h = h

Pseudocode:

```
BinarySearch(l, h){
        if ( h < l )  return;
        mid = l + ( h - l ) / 2;
        if ( A [mid] == mid )  return mid;
        else if( A [mid] > mid )  return BinarySearch( l, mid-1);
        else return BinarySearch( mid+1, h);
}
```

Proof by Induction:

Base case: n = 0; an array with no value, nothing will happen in this case. n=1. l=h here. Thus, mid = l = h. If A[mid] = mid then the index is returned. If A[mid]>mid, then mid-1 = h for next call -> h<l and hence no such index exists. If A[mid]<mid, then mid+1 = l for next call -> h<l and hence no such index exists. Thus, the base case is proved.

Induction Hypothesis: The algorithm works correctly for any array size <= n-1.

Inductive Step: Consider an array of size n. There are three cases in the routine:

Case 1: A[mid] = mid. Here, the algorithm works correctly.

Case 2: A[mid] > mid then claim is as follows:

If A[i] > i then for all i < j <= n A[j] > j.
Proof: Take a j such that i <= j <= n. Now A[j] > A[j - 1] > A[j - 2]..... > A[i].
A[i] > i i.e A[i] >= i + 1
A[i + 1] > A[i] >= i + 1
A[i + 2] > A[i + 1] >= i + 2
Hence the solution, if it exists, would be in the left subarray, ie, from l to mid-1.
Now, the size of the array in consideration is strictly less than n, by induction hypothesis, the algorithm works correctly.

Case 3: A[mid] < mid then claim is as follows:
If A[i] < i then for all 1 <= j < i A[j] < j.
Proof: Take a j such that 1 <= j <= i. Now A[j] < A[j + 1] < A[j + 2]..... < A[i].
A[i] < i i.e A[i] <= i - 1
A[i - 1] < A[i] <= i - 1
A[i - 2] < A[i - 1] <= i - 2
Similarly for A[i - n] < A[i] <= i - n. Thus, A[i-n] < i-n for all j < i.
Hence the solution, if it exists, would be in the right subarray, ie, from mid+1 to h.
Now, the size of the array in consideration is strictly less than n, by induction hypothesis, the algorithm works correctly.

Thus, by induction it is proved that the algorithm works correctly for any array where the elements are sorted in ascending order.

Runtime of the algorithm:

$$T(n) = T(n/2) + c$$

By Master Theorem, the running time comes out to be O(log n).

31. b) The algorithm consists of three cases:

1. If A[1] == 1: In this case we found an index i i.e 1 such that A[i] == i.Hence we return 1 and terminate the loop
2. If A[1] > 1: In this case, as A[i] < A[j] for i < j, minimum value of A[1] is 2. So A[2] has a minimum value of 2 + 1 = 3 …. and so on. So this continues till the end of the array and thus no integer i satisfying A[i] == i exists for this array.
3. A[1] < 1: This case is not valid according to the given conditions.

Thus we conclude that this algo works in O(1) that is constant time and hence it is faster than the above algorithm.

Pseudocode:

```
FindIndex(A[1...n]){
        if (A[1] == 1) return 1;
        else return;}
```

Proof by induction:

Base Case: n = 0; an array with no value, nothing will happen in this case. n = 1. There is only one element in the array which has to be positive. If it is equal to 1 then it returns 1 else it returns as no such index exists. If it is not then due to the increasing order of the element there would not be any such index.

Induction Hypothesis: The algorithm works correctly for an array of size<= n-1.

Inductive Step: Consider an array of size n. There is only one condition, if the array's first element is 1 then A[i] = i, i = 1, however, if this is not the case then by the claim of base case no such index exists.

By induction, it is proved that the algorithm works correctly.

Problem 2

12. a) As we know that Merge sort is a divide and conquer algorithm which sorts the given array in O(n log n) by dividing the array into two halves, calling itself to sort the two halves and then merging it to give the final result. Thus, the merge sort has two main functions: sort, which divides the array further into their halves; and merge, which finally merges the sorted halves of the array. This cruel and unusual algorithm also sorts the input array and the cruel function works exactly as the sort function in the merge sort. The only difference lies in the unusual algorithm. Since we know that merge sort sorts any given array correctly, it suffices to prove that the unusual works exactly as merge of merge sort.

Statement to be proved: GIven that A[1…(n/2)] and A[(n/2+1)...n] is correctly sorted, Unusual(A[1...n]) sorts the array A[1...n] correctly.

Induction on unusual routine:

Base case: For n = 0 nothing happens and for n = 1, the algorithm does nothing. For n=2. The array contains only two elements. There can be only two possibilities for this case. The first element can be either greater than than the second element or less than or equal to the second element. If the first element is greater than the second element then the elements are swapped and hence the array is sorted. However, if the latter is the case then nothing happens as it is already sorted.

Induction Hypothesis: For any size of the array <=n-1, the routine correctly merges the elements of two already sorted arrays and returns the sorted array.

Inductive Step: Consider an array A of size n. Let there be four quarters of A defined as follows:

A1 = A[1...n/4]
A2 = A[n/4+1...n/2]
A3 = A[n/2+1...3n/4]
A4 = A[3n/4+1...n]

The size of each of the above mentioned arrays is strictly less than n and hence they are already sorted by induction hypothesis. Henceforth, A1 ∪ A2 and A3 ∪ A4 are also sorted.

- Thus, the smallest n/4 elements of A would be in A1 and A3.
- The for loop swaps the second and the third quarter. Hence, A2 and A3 are swapped. This implies that now the smallest n/4 elements of A are now in A1 and A2 while they both are individually sorted by induction hypothesis.
- First recursive call to the unusual routine sorts A1 ∪ A2 as advocated by induction hypothesis. Thus, after this call the smallest n/4 elements of A lie in A1 in sorted order.
- Similarly, the largest n/4 elements of A would initially be in A2 and A4. However, after the for loop swaps A2 and A3, the largest n/4 elements of A would now lie in A3 and A4 while they are individually sorted.
- The first recursive call to unusual does not modify A3 or A4. By induction hypothesis, the second recursive call sorts A3 ∪ A4. Now, the n/4 largest elements of A lie in A4.
- Since A1 and A4 contain the smallest and the largest n/4 elements of the original array respectively, A2 and A3 thus contain the middle elements perhaps in unsorted order. But note that A2 and A3 are individually sorted.
- The last recursive call to unusual sorts A2 and A3 as their sizes are less than n.

Induction on cruel routine:

Base case: n = 0, the algorithm does nothing. n=1; the routine does nothing as an array containing one element is already sorted.

Inductive Hypothesis: For any size of the array<=n-1, the cruel routine correctly divides the array into two halves and merges to give a completely sorted array.

Inductive Step: Cruel divides the given array A of size n into A1 = A[1..n/2] and A2 = A[n/2+1...n]. By the inductive hypothesis, both the arrays A1 and A2 are correctly sorted as their size is strictly less than n. Moreover, in the proof above the unusual routine correctly merges the two sorted arrays into one single array so this means that this function correctly sorts the given array.

Thus, by Induction it is proved that Cruel correctly sorts any input array.

12. d) The recurrence relation of the Unusual is:

$$T(n) = 3T(n/2) + O(n)$$

The routine calls itself three more times on a size of n/2. Moreover, in one call the for loop runs for n/4 times but in the big-oh notation the constant does not affect the runtime and hence the above recurrence relation.

By Master Theorem, the running time of Unusual is O (n^(log 2 3)) (n raised to log 3 base 2)

12. e) The recurrence relation of the Cruel is:

T(n) = 2T(n/2) + running time of unusual

The routine calls itself two times and one time unusual is called on size n, hence we get the above recurrence relation.

T(n) = 2T(n/2) + O (n^(log 2 3))

By Master Theorem, the running time of Cruel is O (n^(log 2 3)) (n raised to log 3 base 2)

Problem 3

14. b) The circle contains n sets of two points which creates n line segments in the circle. These line segments form sectors of a circle. To find the number of intersections of lines we can find the number of overlapping sectors. But one thing to note is that the sectors should be partially overlapping because if one sector is contained inside the other then the corresponding line segments would not intersect. The steps of the algorithm are described below:

- Swap pi and qi in each pair if angle wrt x-axis of pi > qi (this step is optional as without swapping also this would work because the line segment does not change if the notation is swapped)
- Now we write all pi and qi in an anticlockwise order from any point as the starting point. Writing in anticlockwise means only to sort the given 2n points in ascending order using their angle as the parameter. For example, in the given figure we have [p1, p6, p3, p7, p4, q4, p2, q1, q6, q7, q3, p5, q2, q5]. Here we have swapped pi and qi if pi>qi.
- Taking [pi, qi as one interval], we get an array of tuples. In this case [(1, 8), (2, 9), (3, 11), (4, 10), (5, 6), (7, 13), (12, 14)]. So we have n such intervals for n pairs of points.
- Now we use DIVIDE AND CONQUER Step:
    a. We get two subarrays of this array ie. array[0, m - 1] = L and array[m , n] = R, m is the mid point. We, beforehand, know the number of partially intersecting sectors in both the subarrays, ie, the number of overlapping intervals. Also, the start and end times are sorted individually in both.
    b. So for a L[tuple_i].end we find the largest R[tuple_j].start such that R[tuple_j].start < L[tuple_i].end. Moreover, since R is sorted according to the start values, we

can add the number of sectors, ie, intervals whose start value comes out to be less than L[tuple_i].end

c. To find the sectors that are entirely contained inside some other sector, we need to count the number of intervals whose end value < L[tuple_i].end. This needs to be removed from the answer.
d. So to find such sectors we need to find the largest end value in R such that it is less than L[tuple_i].end. Let the index be k. So we need to subtract (k - m) if such a k exists.
e. To get at the final answer, we will need to merge both the subarrays. The merge step is derived from the merge sort.

As at each step(while iterating through the arrays) we need exactly 2 binary searches and we are dividing the array into two halves, we get the recurrence relation as:
T(n) = 2*T(n/2) + n log n

Complexity with Master's Theorem : O(n(logn)^2)

14. c)

In the above approach we use divide and conquer only for sorting. Also, decrease and conquer is used when we do binary search.

- We sort the start and end values of the tuples first.
- The steps for counting the number of overlaps will be the same as above.
- If A is the array of the intervals created above which is sorted according to the start time
- For some interval A([pi, qi]) we need to find the interval A([pj, qj]) that has the largest A[j].start < A[i].end. Add j − i to answer (answer is the number of intersections).
- Now we need to find the interval which has the largest end value which is <A[i].end.(This is because if both start and end of A[j] are less than A[i].end then it means A[j] fully lies in A[i]) If position of this end time in the sorted end time array is k and if it exists, we subtract k − i from answer.

Sorting takes O(nlogn) time and for each element of the array we do binary search that takes O(logn) time, thus for n elements, O(nlogn) time.
Time Complexity: O(n*log(n))

Proof by arguments:
The total number of intersections can be found out by the following way: we can see the total number of sectors subtract from it the number of sectors that overlap completely and that are disjoint. This will give us the number of sectors that overlap partially.

1. Moreover, from geometry it is evident that for intersection of any two line segments in the circle [pi, qi] and [pj, qj] the condition that should be true is angle(pi) < angle(pj)< angle(qi)< angle(qj). (angle with respect to x-axis)
2. With the tuple A[i] corresponding to each segment, we know that line segments from each tuple are intersecting if A[i].end is greater than A[j].start and A[j].end is greater than A[i].end. This is observable with geometry. So, we add j - i to the answer as all tuples between A[i] and A[j] satisfy this condition.

3. Now we need to find the interval which has the largest end time which is < A[i].end.(This is because if both start and end of A[j] are less than A[i].end then it means A[j] fully lies in A[i]) If position of this end time in the sorted end time array is k and it exists, we subtract k − i from count(As all tuples lying between i and k will not satisfy the condition of intersection)
4. As stated in lectures, Merge sort sorts the array correctly.
5. So in a combined fashion it is proved that the algorithm works correctly and reports the number of intersection points in a circle.

PSEUDO CODE FOR O(nlogn)

```
int main()
    for (segments in array):
        If (pi > qi){swap(pi, qi)}
            //generateTuples is to generate tuples for each segment [pi, qi]
    int[][] arraylist_of_states = generateTuples(array)
    arrayX = SortX(arraylist_of_intervals)
            //sorting done according to the start time with merge sort
    arrayY = SortY(arraylist_of_intervals)
           //sorting done according to end time with merge sort
    int Ans = 0;

    int i1 = 0; int j1 = 0;

    int[][] A = arraylist_of_intervals;

    for (int i = 0; i < n; i++){
          interval_j = findLargestStartValue<A[i].end
               /*find the index of a tuple j with BinarySearch such that j has the largest start
index less than the end value of A[i] */

          interval_k =  findLargestEndValue<A[i].end
                /* find the index of a tuple k with BinarySearch such that k has the largest end
value less than the end value of A[i]*/

           Ans += (j - i)
           if (k - i >0){
                  Ans -= (k - i);
           }

      return Ans;
}
```