Homework Assignment 1 (Theory)

Submitted by
Vidhi Sharma 2019286
Dolly Sidar 2019304

1. **a) <u>Algorithm</u>**:

   //Return any index i such that low ≤ i ≤ high and A[i] = i

```
low = 1;
high = n;
Search( Array A , low , high)
{
        if (low > high)                          -> O(1)
                return none;
        mid = (low + high) / 2 ;                 -> O(1)
        if( A[mid] == mid )                      -> O(1)
                return mid ;
        else if ( A[mid] < mid )
                return Search (mid +1 , high ) ;        ->T(n/2)
        else
                return Search (low , mid-1 );           ->T(n/2)

 }
```

<u>Runtime</u>: O (log n)
Let T(n) be the number of steps taken by the algorithm Search on an array of size n.

$$T(n) = \begin{cases} c & \text{if low>high} \\ T(n/2) + c & \text{otherwise} \end{cases}$$

- due to the recursive calls on array of size n/2.

where c is constant.

$T(n) = T(n/2) + c$
By using Master's Theorem the runtime is O(log n).
$T(n) = O(\log n)$

Correctness :
Proof by induction,
Base case: In the case where n=0
such that low >high, No elements found. The Algorithm is Correct.
Induction Hypothesis: We assume the algorithm Search is correct for an array of size <=n, where n>=0.
Inductive step: We take an arbitrary array of length n+1. Since, the induction hypothesis applies an array of size <=n. Our goal is to prove that it is also true for the array of size n+1.
As Array A is sorted -
Case 1: A[mid]=mid
It returns mid, which is the index we are trying to find.
Hence, Clearly, the algorithm works correctly.
Case 2: if A[mid]>mid then
for all k>mid:

A[k]>A[mid] as array A is strictly increasing.
So, A[k]>A[mid]>=k>mid
This shows that A[k]>k for all k>mid

This implies we have to search only in A[low….mid], A[i]=i cannot appear within the right portion of the array because all the elements on that portion of the array have A[i]>i. A[low...mid] has a size less than n. And we assumed that our algorithm works for size<=n (Induction Hypothesis applies to size<=n). Hence, the algorithm is Correct.
Case 3: if A[mid]<mid then
for all k<mid:

A[k]<A[mid] as array A is strictly increasing.
So, A[k]<A[mid]<=k<mid
This shows that A[k]<k for all k<mid

This implies we have to search only in A[mid….n], A[i]=i cannot appear within the left portion of the array because all the elements on that portion of the array have A[i]<i. A[mid….n] has a size less than n. And we assumed that our algorithm works for size<=n (Induction Hypothesis applies to size<=n). Hence, the algorithm is Correct.

So, the algorithm is true for k=n+1.
Hence it is true for all n.

b) <u>Algorithm</u>:

```
Search (Array A)
    {
        If A[1] == 1                    ->O(1)
            return 1;
        else
            return none;
    }
```

<u>Runtime</u>: T(n) = O(1) as we are searching a key element present at the first index and which is done by O(1) time.

<u>Correctness</u>:
Proof by contradiction
Suppose that there exists an i such that A[i] = i when A[1]>1 .
So If A[1] > 1 then,
A[i] >i for every index i because the array A is strictly increasing such that A[1]< A[2] <.....< A[n]. Therefore we never get an index i such that A[i] = i .
This contradicts the fact that there exists an i such that A[i] = i when A[1]>1.
So, A[1]>0 (given) and A[1]<=1(proved by contradiction) which implies that A[1] has to be equal to 1 for there to exist an index i such that A[i]=i.
In this case, A[1]=1, So index 1 is the index such that A[i]=i.

2. **a)** Proof by strong induction
   Assuming that the size of the array is the form of 2^n.
   Base Case: n=2

When n=2:
Cruel(A[1..2]) calls Cruel(A[1..1]), Cruel(A[2..2]) and Unusual(A[1..2])

- Cruel(A[1..1]), Cruel(A[2..2]) doesn't return anything (as n<=1 for them)
- Unusual(A[1..2]) has n=2 . So, if A[1] > A[2], it swaps A[1] and A[2].Thereby, the array is sorted.

The base case holds true.
Induction Hypothesis: We assume that Cruel sorts an array for any n<= 2^k, where k>=2 and A[1 …. 2^(k-1)] and A[2^(k-1) +1…..n] are sorted arrays.
Induction Step: Consider an arbitrary array of length 2^(k+1). Since the induction Hypothesis applies to an array of length 2^k, we have to prove that it also applies to any array of length 2^(k+1).
Cruel(A[1…. 2^(k+1)]) calls Cruel(A[1…. 2^(k)]) and Cruel(A[2^(k)+1…. 2^(k+1)]) to sort these partitioned arrays.
According to the Induction Hypothesis, Cruel sorts arrays for any array length<=2^k. Since, n=2^k for them, these two partitioned arrays are sorted by Cruel.
Now, Cruel calls Abnormal(A[1…. 2^(k+1)]).
Consider that sorted array 1 A[1…. 2^(k)] - It consists of 1st quarter A[1…. 2^(k-1)] and 2nd quarter A[2^(k-1)+1….2^k].
Since array 1 is sorted, Every element of quarter 1<Every element of quarter 2.
Consider that sorted array 2 A[2^(k)+1….n] - It consists of 3rd quarter A[2^(k)+3n/4]…. and 4th quarter A[3n/4+1….n].
Since array 2 is sorted, Every element of quarter 3<Every element of quarter 4.
The first loop of Unusual swaps the 2nd and 3rd quarters.
So, now 1 st and 3 rd quarters have almost all minimum elements and have the lowest element too.
Similarly, now 2nd and 4th quarters have almost all maximum elements and have the highest element too.
Now, the call U(A[1 .. n/2]) - First smallest n/4 elements of the array are sorted so 1 st quarter is sorted.
The call U(A[n/2 + 1 .. n]) - Last maximum n/4 elements of the array are sorted so 4th quarter is sorted.

The call U(A[n/4 + 1 .. 3n/4]) - 2nd and 3 rd quarters are sorted and compared.
2^k elements are left. Since IH applies to any array having length <= 2^k. These elements are also sorted.Thereby, the complete array is sorted.
So, the algorithm is true for n= 2^(k+1)
Hence it is true for all n.

(d) Let T(n) be the number of steps taken by the algorithm Unusual on an array of size n. The algorithm does four kinds of work : recursive call on array of size n/2, again recursive call on array of size n/2, and again recursive call on array of size n/2 and swapping the 2nd and 3rd quarters.
The three recursive calls take T(n/2) steps each (since all three arrays have size n/2).
Now let us find the number of steps taken in swapping the 2nd and 3rd quarters of the array. The for loop runs from i=1 to i=n/4.
So, it runs only n/4 iterations, each iteration takes O(1) steps.
Hence, the time complexity of this loop is O(n).
Hence, T(n) satisfies

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

$$T(n) = a\,T(n/b) + c*n^d$$

In the above recurrence relation:

$$a=3,\ b=2,\ d=1$$

$$b^d = 2^1 = 2$$

Since, $a>b^d$

Therefore According to Master's Theorem -

$$T(n) = O(n^{\log_b a})$$

$$= O(n^{\log_2 3})$$

(e) Let T(n) be the number of steps taken by the algorithm Cruel on an array of size n. The algorithm does three kinds of work : recursive call on array of size n/2, again recursive call on array of size n/2, and Abnormal procedure on the whole array of size n.
The two recursive calls take T(n/2) steps each (since both arrays have size n/2).
The time complexity of Abnormal procedure for an array of size n is

$O(n^{\log_2 3})$,using the solution of part (d)
Hence, T(n) satisfies

$$T(n) = 2T(n/2) + O(n^{\log_2 3})$$

$T(n) = a\,T(n/b) + c*n^d$
In the above recurrence relation:
$a=2, b=2, d= \log_2 3$
$b^d = 2^{(\log_2 3)} = 3$
Since, $a < b^d$
Therefore According to Master's Theorem -
$T(n) = O(n^d)$
$= O(n^{\log_2 3})$

3.

Algorithm:

The array contains 2n points , such that p1...pn & q1...qn. The pairs (pi,qi) and (pj,qj) of the line segments intersect if and only if i points appear exactly once in the array between a pair (pj,qj) . We create an array obtained by reading points in one direction in which they appear around the circle starting from any point. This array arr contains labels instead of p1...pn,q1….qn such that the label for pi and qi is i.
The idea behind the algorithm is to traverse the Array arr which contain all the given labels . When a label i is seen for the first time , an interval is opened for it .When i is seen again, we close its interval and find out how many intervals have been opened and not closed since i was first seen. Array startLocations[n] stores the indexes of the first endpoints of line segments, while Array visited[n] stores those endpoints of a line segment which has already been seen. The algorithm also has a treeset S that stores endpoint indexes.
After completion, the value of the variable intersections is the number of intersections. For each label with start.i < end.i (start.i and end.i index the start and end points of line segments i). Let's consider two line segments i and j with start.i<start.j . Line segments i and j intersect once if start.i < start.j < end.i and line segments i and j do intersect if end.j < end.i.

```
Array visited[n];     //stores if one of the endpoint of the line segment is
                                                          already seen
Array arr[2n];        // stores the points in the order which they appear in
                                                          the circle
Array startLocations[n];  //stores the indexes of the first endpoint of the
                                                          line segment
i=1;
Intersection =0;
TreeSet<Integer> S;  // a data structure that stores endpoint indexes
                    and perform insert, remove and search in O(log n) time.
while(i<=2n)
{
     j= arr[i];
     if( visited[j] == False)
     {
         visited[j]= true;
         startLocations [j] =i;
         S.add(i);
      }
      else
      {
/*intersection = intersections + no of elements x such that x is in S and
   x > startLocations[j]  /*
//to calculate no of elements in S such that they are >startLocations[j]
//using binary search to find the index of startLocation[j] in S

        find= binarySearch(S.toArray(), startLocations[j]);
        intersections = intersections + (S.size() - find -1)
        S.remove(startLocations[j]);
         }
         i=i+1;
      }
```

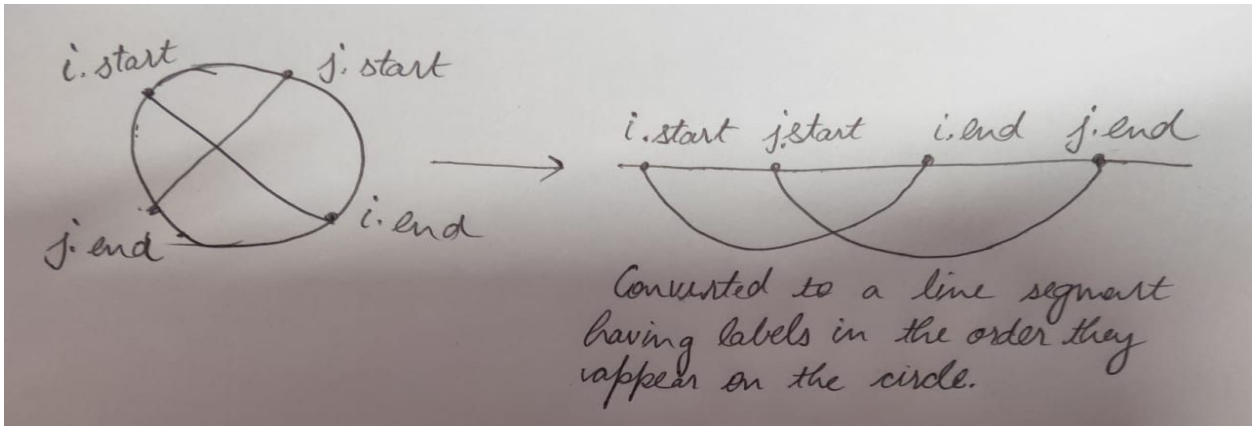
Runtime:
Let T(n) be the number of steps taken by the algorithm to find the number of intersections. From the algorithm it is clear that every label i.e. (1…..n) Is inserted once in the TreeSet S. Now, three operations can be performed on the label: insert, remove, binary search. Since the time complexity for all these operations are O(log n) and the number of labels are n.
Time complexity is O(n log n).

Correctness:
This algorithm is based on 2 facts that:

1) if there are 2 line segments i and j with i.start < j.start, then they only intersect if i.end<j.end
2) if there are 2 line segments i and j with i.start < j.start, then they don't intersect if i.end>j.end

To prove the correctness by Contradiction:

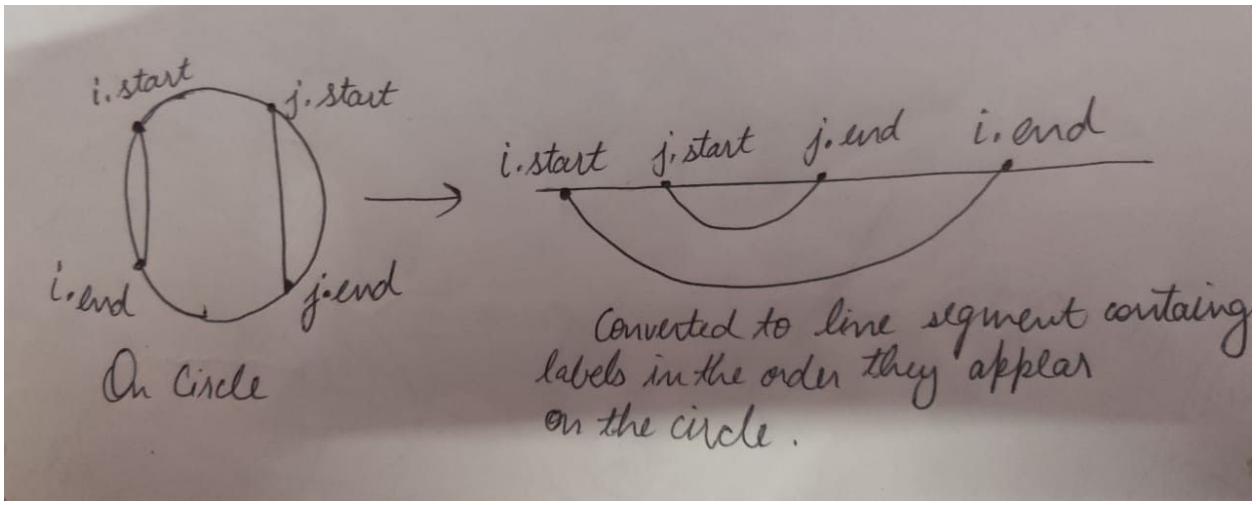
Contradiction on fact 1 -
Let's assume that two line segments i and j such that i.start < j.start. They don't intersect if i.end <j.end.
If i.end < j.end then:



These diagrams clearly show that there is intersection between line segments i and j.
This contradicts our assumption.
Therefore, two line segments i and j such that i.start < j.start intersect if i.end <j.end.

Contradiction on fact 2 -

Let’s assume that two line segments i and j such that i.start < j.start. They intersect if i.end >j.end.
If i.end > j.end then:



These diagrams clearly show that there can never be any intersection between line segments i and j.
This contradicts our assumption.
Therefore, two line segments i and j such that i.start < j.start <u>don’t intersect</u> if i.end >j.end.

Since, both facts have been proved by contradiction. The algorithm is correct.