ADA HW 1

Abhimanyu Gupta : 2019226
Abhinav Gudipati : 2019227

Question 1:

Q1 (a)

Code:

```
def func(A): #A is an array with 1-based indexing
    low=1
    high=len(A)
    while low<=high:
        mid=(low+high)/2 #floor division
        if A[mid]==mid:
            print(mid+1) #desired index
            return #no need to continue searching
        elif A[mid]>mid: #check for previous indexes
            high=mid-1
        else:   #check for forward indexes
            low=mid+1
    print("-1") #no A[i]=i is present
```

Runtime :- O(logn)
Since we are halving the search space (our Array) in every iteration, we can have at most logn iterations For an array of n elements. We are dividing the array in 2 halves for each and every iteration. For which we have the following.
$n * (½)^k = 1$
$\Rightarrow n * \frac{1}{2^k} = 1$
$\Rightarrow \frac{n}{2^k} = 1$
$\Rightarrow n = 2^k$
$\Rightarrow k = \log(n)$
(since $\text{ceil}(2^k) = n \Rightarrow k=\log(n)$) , hence O(logn).

Correctness :-
Since the array is sorted , we can directly search for an element using our modified binary search. Now to prove our algorithm is working we present the following 3 justifications.
1) A[i] > i : Checks for the previous i_s
Suppose A[i] > i for any arbitrary element i, then adding 1 to both the sides, we get
$\Rightarrow$ A[i]+1 > i+1 ..... (I),
Now we know that the array is strictly increasing, this means that for every i ;

⇒ A[i+1] ≥ A[i] +1 ...(II). (Difference between two integers is at least 1)
Now from equation (I) and (II), we can deduce the following
⇒ A[i]+1 ≥ A[i]+1 > i+1 ⇒ A[i+1] > i+1
This means for every i, such that every i , A[i] > i ; there would not exist any index j (j > i) such that A[j] = j
Using the above results, we should check the previous i_s in the array.

2) A[i] < i : Checks for forward i_s
Suppose A[i] < i for any arbitrary element i, then adding 1 to both the sides, we get
⇒ A[i]+1 < i+1 ⇒ i+1 > A[i]+1 .....(I),
Now we know that the array is strictly increasing, this means that for every i ;
⇒ A[i+1] ≥ A[i] +1 .....(II) . (Difference between two integers is at least 1)
Now subtracting equation (I) from (II), we can deduce the following
⇒ A[i+1] - (i+1) ≥ 0 ⇒ A[i+1] ≥ i+1
This means for every i, such that every i , A[i] > i ; This means there might exist an index j (j > i) , such that A[j] = j.
Using the above results, we should check the forward i_s in the array.

3) A[i] = i : index i is the answer

Q1 (b) :

Code:

```
def func(A): #A is an array with 1-based indexing
    if A[1]==1:
        print("1") #desired index
    else:
        print("-1") #no A[i]=i is present
```

Runtime : O (1)
Since we are checking only just the 1st index, for whatever might be the length of A, therefore constant time O(1).

Correctness :
We are given A[1] > 0,
⇒ A[1] ≥ 1 (minimum non negative integer).
Case 1:
A[1] = 1 ; then the answer would be 1.
Case 2:
A[1] ≠ 1, i.e A[1] > 1,
Then adding 1 to both the sides,
⇒ A[1]+1 > 1+1 ..... (I),
Now we know that the array is strictly increasing, this means ;
⇒ **A[2] ≥ A[1] +1** .....(II) . (Difference between two integers is at least 1)
Now from equation (I) and (II), we can deduce the following
⇒ A[2] ≥ A[1]+1 > 1+1 ⇒ A[2] > 2 , i.e A[2] ≠ 2
Now continuing in the same manner we can show that for every index i > 1,
⇒ A[i] > i, i.e. A[i] ≠ i.

This means for A[1] > 1, there won't be any index i in [1 … n] such that A[i]=i.
Therefore from case 1 and 2, we deduce that the answer exists only when A[1]=1.

<u>Question 2:</u>

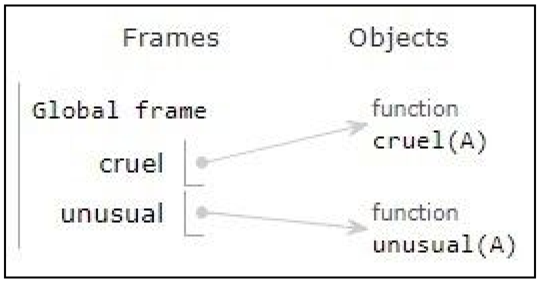
(a)

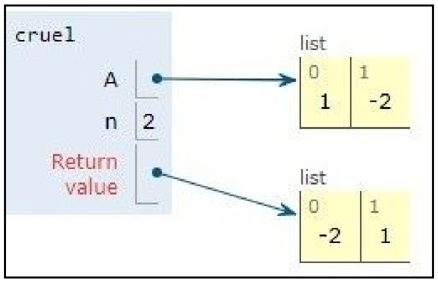
```
Cruel(A[1 … n]):                #line 1
    if n > 1                    #line 2
        Cruel(A[1 … n/2])       #line 3
        Cruel(A[n/2+1 … n])     #line 4
        Unusual(A)              #line 5

Unusual(A):                     #line 6
    if n == 2:                  #line 7
        if A[1] > A[2]:         #line 8
            A[1] <---> A[2]     #line 9
    else for i to n/4:          #line 10
        A[i+n/4] <---> A[i+n/2] #line 11
        Unusual(A[1 … n/2])     #line 12
        Unusual(A[n/2+1 … n])   #line 13
        Unusual(A[n/4+1 … 3n/4]) #line 14
```

Claim: CRUEL Algorithm sorts any input array correctly.

Synopsis : Proof by induction on the length of A (assuming the length of A is in the form of 2^k) . We are proving this using strong induction. We have taken the base case where the number of elements in array A are 2 (i.e k = 1). Further, we are taking the case for length 2^k which we assume to be true in order to further satisfy the inductive step, which is for the length 2^{k+1}.

Base case:





For Length of A, n = 2:

Number of elements in the Array are only 2.

For this case, when CRUEL(A[1..2] is being called, it further calls the following function calls.

(i) CRUEL(A[0])

This function call does not perform any operations given that length of A is 1.

(ii) CRUEL(A[1])

Similarly, this function call does not perform any operations given that the length of A is 1.

(iii) UNUSUAL(A[0..1])

Here when this function is being called, the elements A[0] and A[1] get swapped, if required, with respect to line 9 of the pseudo code.

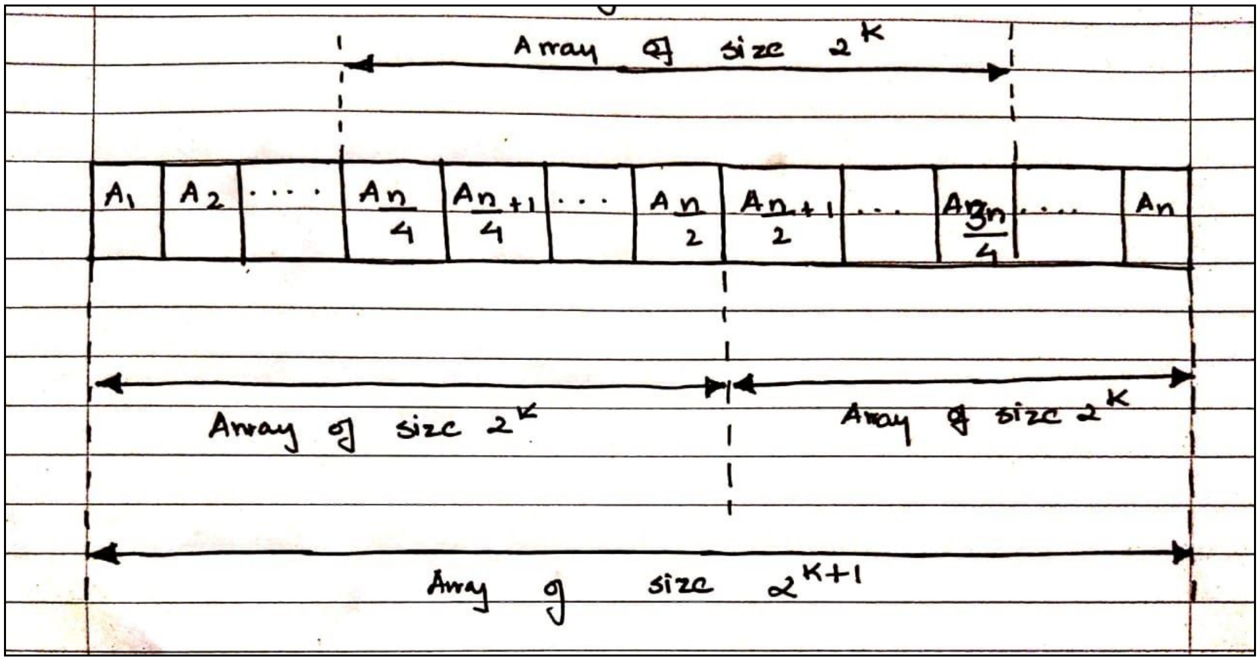
Induction hypothesis:

Let for an array of size 2^k , our claim (i.e. CRUEL Algorithm sorts any input array correctly) to be true. Here for this case we are assuming the following is taking place recursively.

1) The CRUEL Function recursively sorts all sub arrays being called with respect to Line Number 3 for A[1 .. n/2] ;
2) The CRUEL Function recursively sorts all sub arrays being called with respect to Line Number 4 for A[n/2 + 1 .... n] ;
3) Here we need to note that the previous CRUEL Functions take Arrays of size 2^{k-1} each. Since the size of the array assumed is to be 2^k, and the array is split in half for sorting. ($\frac{2^k}{2}$ = 2^{k-1})
4) Lastly the Unusual Function gets called for A[1 ... n]. Here the Sub Arrays A[1 .. n/ 2] and A[n/2 + 1 .... n] are already sorted from the previous function calls for which the whole final array gets further sorted accordingly, inside Unusual(A[1 … n]).
5) Inside the Unusual Function, Sub Arrays A[1 … n/2] and A[n/2 +1 … n] get recursively called and are sorted accordingly, after which the the middle elements A [n/4+1 … 3n/4] also get sorted accordingly. Each of these sub arrays are of size 2^{k-1}.

Induction step:

Let us consider an arbitrary array of size n = 2^{k+1} . Taking this into consideration, we need to prove that CRUEL can further sort an array of size n perfectly given our previous assumption that n = 2^k is true. The following takes place once the Array of size 2^{k+1} is being passed into CRUEL for sorting.



Here the algorithm works in this fashion.

We can see from the above diagram that.

1) **At Line 12**, For the array of size 2^{k+1}, the Array first gets sorted recursively for the left sub-array of length 2^k i.e A[1 .. n /2] elements.
2) **At Line 13**, Likewise, the same is applied to the right sub-array of length 2^k , the Array A[n/2 +1 … n] gets sorted recursively.
3) With respect to Induction Hypothesis we know that both the left and right sub-array of length 2^k will be sorted accordingly.
4) **At Line 14** , Lastly the sub-array [n/4+1 … 3n/4] (which is also of size 2^k) gets sorted recursively. This can be applied here with reference to Induction Hypothesis.
5) In this fashion, we can deduce that the array of size 2k + 1 gets sorted.

Therefore from the induction, it is proved that CRUEL sorts any input array of size a power of two correctly.

(b)

For an array of size n, Unusual(A[1 … n]) make 3 recursive calls to self, each for array of size n/2; namely Unusual(A[1 …. n/2]), Unusual(A[n/2 +1 … n]) and Unusual(A[n/4+1 … 3n/4]). Addition to this Unusual(A[1 … n]) performs n/4 swaps in Line 10-11, thus additional O(n/4).

Time complexity of Unusual,

$T(n)= 3*T(n/2) + O(n/4)$

$\Rightarrow T(n) = 3*T(n/2) + O(n)$

Using masters theorem [$T(n) = a * T(n/b) + n^c$] , for a = 3, b = 2 and c = 1

Now, $c < \log_b a \therefore T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

Time complexity of Unusual is $O(n^{1.585})$

(c)

For an array of size n, Cruel(A[1 … n]) makes 2 recursive calls to self, each for array of size n/2 ; namely Cruel(A[1 … n/2]) and Cruel(A[n/2 + 1 … n]), other than this it also makes a third call to Unusual(A[1 … n]).

Time complexity Cruel,

$T(n) = 2*T(n/2) + O(n^{\log_2 3})$ { Time Complexity of Unusual is $O(n^{\log_2 3})$ from previous part (b) }

Using masters theorem [$T(n) = a * T(n/b) + n^c$] , for a = 2, b = 2 and c = $\log_2 3$

Now, $c > \log_b a \therefore T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

Time complexity of Cruelis $O(n^{1.585})$

Question 3: Accounted only for the C part keeping in mind Optimised Solution O(nlogn)

```
countIntersection(pointPairs)                                 #line 1
   orderedPoints = getOrderedPoints(pointPairs)               #line 2
   orderedPairs = makeIndexPairs(pointPairs,orderedPoints)    #line 3

   n = length(orderedPairs)                                   #lin 4
   firstPointSortedList = orderedPairs                        #line 5
   secondPointOrderedSet = getOrderedSetBasedOnSecondIndex(orderedPairs)  #line 6

   count = 0                                                  #line 7
   for i in [1 ... n]                                         #line 8
       p = firstPointSortedList[i]                            #line 9

       j = findLessThanOrEqual(firstPointSortedList, i, n, pair(P.secondIndex, INFINITY))
#line 10
       count += (j - i)                                       #line 11

       k = secondPointOrderedSet.findPosition(P)              #line 12
       count -= (k - 1)                                       #line 13

       secondPointOrderedSet.delete(P)                        #line 14

   return count                                               #line 15
```

Explanation of some names used(with example):

#countIntersection(List) - a function to count the number of intersections between line segments having end points on the circumference of a Circle.

#pointPair - a list of objects of type PointPair (PointPair has two members -firstPoint and secondPoint, storing both ends of the line segment

{(p1,q1),(p2,q2),(p3,q3),(p4,q4),(p5,q5),(p6,q6),(p7,q7)}

{Here ; (pi,qi) represents the 2 ends of a line segment.}

#orderedPoints - a list of all points on circumference in a manner if all points are traversed in anticlockwise order starting from a particular point. This can be done by sorting the points on the basis of the angle the line joining the point and the center of the circle make with a vertical line passing through the center.

{p1 , q6 , p3 , q7 , p4 , q4 , p2 , q1 , p6 , p7 , q3 , q5 , q2 , p5 }

#orderedPairs - a list of pairs of two indexes each, based on the index-position of PointPair.firstPoint and PointPair.secondPoint in the orderedPoints

{(1,8) , (2,9) , (3,11) , (4,10) , (5,6) , (7,13) , (12,14) }

#firstPointSortedList - a list of pairs of indexes from orderedPairs which are sorted on the basis of PointPair.firstPoint

{(1,8) , (2,9) , (3,11) , (4 ,10) , (5,6) , (7,13) , (12,14) }

#secondPointOrderedSet - a set (like ordered_set provided as PBDS in GNU C++) to provide O(log(n)) time access to element position in the set, other than O(log(n)) time deletions, it is sorted on the PairPoint.secondPoint member

{(5,6) , (1,8) , (2,9) , (4 ,10) , (3,11) , (7,13) , (12,14) }

#findLessThanOrEqual(sortedList, start , end , searchValue) - This is a function which returns the location of the maximum element just smaller or equal to the passed 4th parameter

findLessThanOrEqual(firstPointSortedList, 1 , n , (8,infinity)}

⇒ This function will return 6.

#findPosition(value) - a member function which returns the location of passed value present inside itself

secondPointOrderedSet.findPosition((1,8))

⇒ This function will return 2.

#delete(value) - a member function which removes the passed value present in itself

Correctness:

- When the list of points are converted into relative ordering, we can observe that intersections between two line segments occur only when the interval of one line segment (inclusive range of the indices of both the points) overlaps the interval of other.
- If the intervals **do not overlap** then one pair of points would completely lie inside the other or would be present in either left or right of the other interval in this relative ordering.
- If the two intervals (a1,b1) and (a2,b2) are in ascending order then an overlap would happen if and only if the following condition satisfies → a1<a2<b1<b2
- Instead of checking for the complete condition we can just check if a2<b1<b2 [as all a_is and b_is are unique] for all the pairs of points. To further simplify we can break the condition into a2<b1 and b1<b2, so as to ease comparisons.
- So, we can individually count all the pairs where **a2<a1** is satisfied and then reduce the counts of elements where b1>b2 i.e the condition **b1<b2** is **violated**. This will give the count of all the elements in the list which satisfies **a2<b1<b2**.
- In a sorted list of (a_i,b_i) sorted with respect to a_i ; if we find the maximum a_j (towards the right side of the ai) which satisfies **$a_j < b_i$** , here we take the index position of the pair that aj belonging to in the firstSortedList using the binary search after which we add (j-i) to the count, since all the indices k_s in range (i,j] will satisfy the condition. # Line 11.
- Similarly, if we have a set ordered with respect to the 2nd index of a pair of indices { (a ,b) , b is the 2nd index } ; in this case we will find the index of the pair P in the secondPointOrderedSet and perform the operation followed in Line 13.
- Performing the steps for all the pairs of points will give us the total number of intersections.

Complexity:

The following operations are performed in countInversion function:

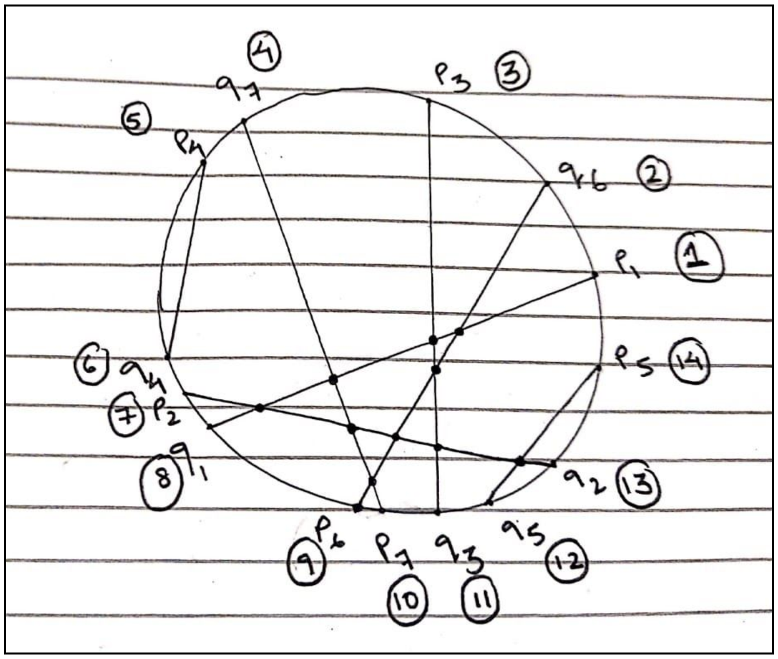
- ➔ Make a sorted list of all the points on the basis of the angle they form with the center of the circle and x-axis. → O(n *log(n))
- ➔ Make a list of pairs of indices from the above sorted list where the indices in the pair denote the two ends of the line segment. → O(n) [with the help of hashmap]
- ➔ Make a set of the above pairs list on the basis of the second index present in a pair → O(n * log(n))

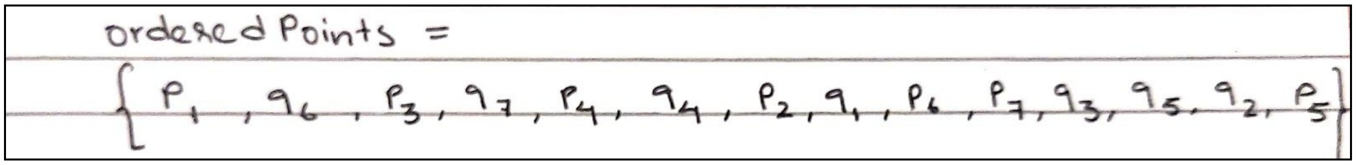
- ➔ Count forward intersection for a single pair of points. → O(log(n))

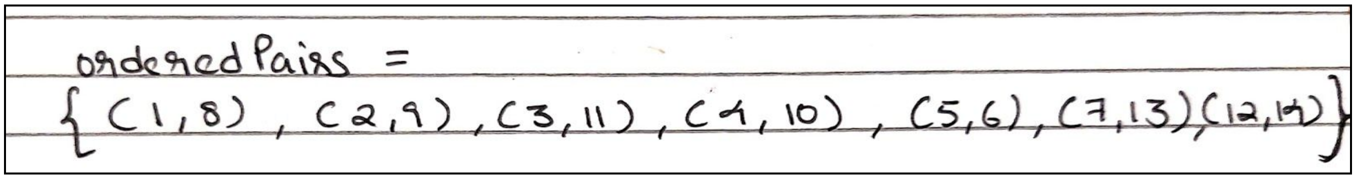
  ∴ Counting intersections for n number of pairs → O(n*log(n))

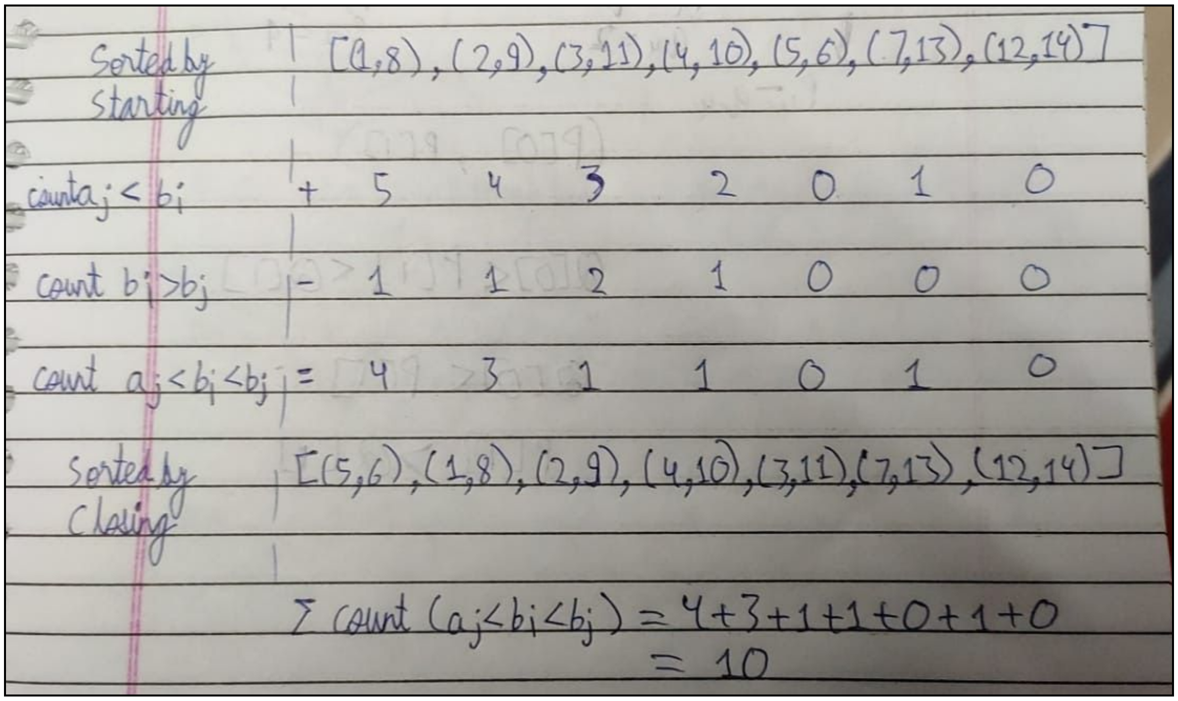
Overall Complexity: O(n*log(n)) + O(n) + O(n*log(n)) + O(n*log(n))
= O(n*log(n))

Example :-




orderedPoints =
$\{p_1, q_6, p_3, q_7, p_4, q_4, p_2, q_1, p_6, p_7, q_3, q_5, q_2, p_5\}$


orderedPairs =
{(1,8), (2,9), (3,11), (4,10), (5,6), (7,13), (12,14)}



Sorted by Starting	[(1,8), (2,9), (3,11), (4,10), (5,6), (7,13), (12,14)]							
count $a_j < b_i$	+	5	4	3	2	0	1	0
count $b_i > b_j$	-	1	1	2	1	0	0	0
count $a_j < b_i < b_j$	=	4	3	1	1	0	1	0
Sorted by Closing	[(5,6), (1,8), (2,9), (4,10), (3,11), (7,13), (12,14)]							

$$\sum \text{count}\,(a_j < b_i < b_j) = 4+3+1+1+0+1+0$$
$$= 10$$

This gives the number of intersections between line segments to be 7.