Player 1 : Kishan Sinha
2019428
Player 2 : Abhishek Chaturvedi
2019401

PROBLEM 1 :

Algorithm and Pseudocode :

The following Pseudocode is based on the reasoning that , for a sorted array , A[1...n] , which has all unique elements , if there exists an index in such a way that A[i] > i , then for all elements to the right of the that element in the array will always satisfy the condition , A[k] - k > 0 for all k > i , and the element y such that A[y] = y (if it exists) can only exist in the left side of the mid element in the array **(Claim 1)** , and similarly if there exists an index in such a way that A[i] < i , then for all elements to the left of the that element in the array will always satisfy the condition , A[k] - k < 0 for all k < i and the element y such that A[y] = y (if it exists) can only exist in the right side of the mid element in the array (**Claim 2**) . Hence , we can solve the problem using a Binary search like approach. This is clearly observed in all examples and follows from basic intuition, but we will provide a formal proof of why this algorithm works .

Assuming 1 based indexing for all arrays :

Let the procedure be called findSpecialIndex(Arr, start , end) :

// We will initialize start = 1, and end = size(Arr)

```
procedure findSpecialIndex(Arr, start , end):
        if(start > end )
                Return None
        Mid = (start + end)/2

        if(Arr[Mid] == Mid)
                Return Mid
        if(Arr[Mid] > Mid)
                Return findSpeicalIndex(Arr, start , mid-1)

        if(Arr[mid] < Mid)
                Return findSpecialIndex(Arr, mid + 1 , end )
```

Description of the algo : We take the mid element of the array and see if it satisfies the required condition that A[i] = i, if true then we return the index saying that we found the required element , if A[i] > i , we then focus on the subarray A[1… i-1] , and if A[i] < i , we focus on the subarray A[i+1….n] .

Proof of Claim 1 : Since the elements are sorted and distinct , it means if the elements are a1, a2, a3 , a4 …
Then it follows that , a1 < a2 < a3 < a4 …. <ai < .. < aN

Let i be any integer b/w 1 and n
Such that $A[i] > i$, $\Rightarrow A[i] = i + p$ (p is a +ve Integer)
The minimum value of 'p' could be 1
Hence, $A[i] \geq i+1$

Similarly, $A[i+1] > A[i]$
$\Rightarrow A[i+1] \geq i+2$ } with the same logic as above

Hence for any 'k' $> i$
we know, $A[k] > A[i]$ and $A[k] \geq A[i] + k - i$
$\Rightarrow A[k] \geq k + (A[i] - i)$
$\Rightarrow A[k] \geq k + p \Rightarrow \boxed{A[k] > k}$

* in the (Similarly) part of the above explanation , since , A[i+1] > A[i] , then , A[i+1] > i+p (given) , hence, A[i+1] >= i + p + 1 , and , A[i+1] >= i + 1 + 1 (min value of p is 1). Therefore , A[i] >= i+2 .
Similarly ,
A[i+2] >= i+3
A[i+3] >= i+4
A[i+4] >= i+5
A[i+5] >= i+6

.

.

Putting, $i + t = k$

$A[k] >= k+1$

Hence, A[k] > k

Claim 2 can be proved in a similar way .

The rest working of the algorithm is like Binary Search and can be proved through **induction.**

Induction Hypothesis : The function Returns correct answer for the array of size 'n'

Base Case : n == 1 , then if mid = A[mid] i.e., A[mid] = mid , then it returns that index else it returns None

Induction Step : Let us assume that the function returns the correct answer for all arrays of sizes 'k' such that k < n .

Now , the function calls two recursive calls on the same function with sizes , (start + mid - 1)/ 2, which is n/2 , and from the induction Step statement we assumed that the function returns correct answer for all k < n ., since here k = n/2 < n .

Hence Proved .

<u>TIME COMPLEXITY</u>:

Since we are using simply the Binary Search for this problem, hence the Recurrence Relation for the problem is simply, $T(N) = 2T(N / 2) + C$ After solving the Recurrence Relation above the Time Complexity (using Master Theorem) of this problem will be O(log n).

<u>(b)</u>

<u>ALGORITHM</u>:

```
findSpeicalIndex(Array A)
    IF A[1]=1
        RETURN 1
    ELSE
        RETURN -1
```

<u>PROOF CORRECTNESS</u>

Array is in increasing order so there are no duplicates, 1st element is positive

Therefore the elements following it must be positive as they are greater than the 1st element.

If Arr[i] = x, then Arr[i+1] must be >= x+1, Arr[i+t] >= x+t

Consider following cases

Case 1- Arr[1] = 1, this means that Arr[i]=i, hence return 1.

Case 2- Arr[1] != 1, therefore Arr[1] must be >=2, Arr[1]>1, Arr[1+t]>1+t, hence Arr[1+t] != 1+t, for t<n, thus return -1.
The time complexity of the algorithm will be O(1) cause we are doing just one operation.

PROBLEM 2 :

(a) To Prove: CRUEL correctly sorts any input array whose size is a power of 2.

We will prove this using induction on the size of the array.

Base Case: We will prove the base case for array of size 2^0 and array of size 2^1.

i) Let A be the array of size = 2^0 = 1.

Execute CRUEL(A[1])

Here, the program will return from the CRUEL function because n<=1

Now, an array of size 1 will always be sorted. Therefore, CRUEL sorts an array of size 1.

ii) Let A be the array of size = 2^1 = 2.

Executing CRUEL(A[1....2]):

Here, n = 2, therefore (n>1) is true.

CRUEL(A[1]) and CRUEL(A[2]) will be executed without making any changes since sizes of these arrays are 1 (as explained above).

UNUSUAL(A[1….2]) will be executed.

In UNUSUAL function, if(n==2) condition is true since n is 2.

If (A[1]>A[2]) is true, then swapping of A[1] and A[2] will occur, otherwise no changes will take place in A.

Then, UNUSUAL returns and we get sorted A.

Hence, the claim is true for an array of size 2^0 or 2^1.

Hence, the claim is true for the base case.

Induction Hypothesis (IH): Let the claim be true for any array having size of the form of 2^x, for x>=1 and x<=k, k is an integer.

Induction: We will show that for any array of size 2^{k+1} claim holds true.

Let A be any array of length n = 2^{k+1}.

Here, k+1 >= 2 (since k>=1), so n = 2^{k+1}>=4. ----------------(1)

Since, 2^{k+1} is a multiple of 4, so A can be divided into four equal parts.

Let array P = A[1….n/4], Q = A[n/4+1….n/2], R = A[n/2+1….3n/4], S = A[3n/4+1….n]

Thus, A = P∪Q∪R∪S

Executing CRUEL(A[1….n]).

Here, if(n>1) is true ------ Using (1)

CRUEL(A[1….n/2]) i.e. CRUEL(P∪Q) will be executed

Here, size of P∪Q = n/2 = $2^{k+1}/2 = 2^k$, therefore, for P∪Q, IH is valid, therefore after program returns from CRUEL(P∪Q), we get P∪Q as sorted.

Similarly, executing CRUEL(A[n/2+1....n]) i.e., CRUEL(R∪S) will be called and we get R∪S as sorted.

UNUSUAL(A[1....n]) will be executed now.

Here, n>4 -------- using (1)

Currently, A = P∪Q∪R∪S, also P∪Q and R∪S as sorted.

P∪Q is sorted therefore P, Q are sorted as P is the first half of a sorted array P∪Q, and Q is the second half of sorted array P∪Q.

Similarly, R, S are also sorted.

P∪Q is sorted so n/4 greatest elements in A[1....n/2] lie in Q and n/4 least valued elements in A[1....n/2] lie in P.

R∪S is sorted so n/4 greatest elements in A[n/2+1….n] lie in S and n/4 least valued elements in A[n/2+1....n] lie in R.

Above statements mean that n/4 least valued elements of A lie in P∪R and n/4 greatest elements of A lie in Q∪S.

Now, the for loop will swap second and third quarters of A, so A=P∪R∪Q∪S.

Then, UNUSUAL(A[1....n/2] will be executed i.e. UNUSUAL(P ∪ R).

Before executing UNUSUAL(A[1...n/2]), let us see what CRUEL(A[1....n/2]) does.

Here, $2^{k+1}/2 = 2^k$ so n>1 since k>1.

Then CRUEL(A[1….n/4], n/4 = 2^{k-1}, so IH is valid on A[1...n/4], so after executing from CRUEL(A[1….n/4]), we get sorted A[1....n/4].

Similarly, after executing from CRUEL(A[n/4+1….n/2], we get sorted A[n/4+1....n/2].

Then, UNUSUAL(A[1....n/2]) is executed i.e. Union of two sorted arrays that are A[1....n/4] and A[n/4+1….n/2] is sent as an argument to UNUSUAL.

After execution of UNUSUAL(A[1....n/2]), CRUEL(A[1….n/2) executes, and by IH that by executing CRUEL(A[1….n/2) we get sorted array A[1....n/2].

Therefore if we send union of two sorted arrays (A[1….n/4] ∪ A[n/4+1….n/2] = A[1….n/2]) in UNUSUAL(), we get a sorted union (here A[1….n/2] as sorted). -------------(2)

Here, UNUSUAL(P ∪ R) is executed, and we have shown that P and R sorted. So, on sending P ∪ R in UNUSUAL, here size of P and R is n/4, therefore by (2), we get P ∪ R as sorted array.

We have P ∪ R as sorted array. Also, we have shown that P ∪ R contains n/4 smallest elements of A, P ∪ R is sorted so P contains the n/4 smallest elements of A in sorted order.

Similarly, UNUSUAL(Q ∪ S) will return the sorted Q ∪ S. Also, we have shown that Q ∪ S contains n/4 largest elements of A, Q ∪ S is sorted implies that S contains the n/4 largest elements of A in sorted order.

Till now, we have n/4 smallest elements of A in sorted order in P, also P = A[1….n/4]. Therefore, the smallest n/4 elements of A are present in the correct position. ----------------------(i)

Also, we have n/4 largest elements of A in sorted order in S, also S = A[3n/4+1….n]. Therefore, the largest n/4 elements of A are present in the correct position. ------------------------(ii)

Now, UNUSUAL(A[n/4+1….3n/4]) is executed, here, R and Q are already sorted, so, by (2) we get R ∪ Q as sorted array.

Therefore, A[n/4+1….3n/4] also gets sorted. --------------------(iii)

So, by combining (i), (ii) and (iii), we get A as sorted.

Hence, CRUEL correctly sorts A.

Part d) Time Complexity of UNUSUAL

Now, for loop runs for n/4 times, and each iteration takes O(1) time, therefore it only runs for O(n) iterations.

For each recursive call, an array of size n/2 is passed as argument to UNUSUAL so each step takes T(n/2) steps.

3 recursive calls are made so T(n)= 3T(n/2)

Time taken for unusual algorithm - T(n) = 3T(n/2) {Due to 3 calls to **UNUSUAL**} + cn {Due to for loop swapping}.

Here, a=3, b=2, d=1

Hence, Time Complexity=$O(n^{\log(3)/\log(2)}) = O(n^{1.585})$ (Using Master Theorem)

Part e) Time Complexity of CRUEL

Now, in CRUEL, 3 kinds of work is done : recursive call on A[1...n/2], recursive call on A[n/2+1….n] and calling UNUSUAL on array A[1….n].

For recursive calls on A[1...n/2] and A[n/2 + 1….n], an array of size n/2 is passed as argument to CRUEL so each of these recursive calls takes T(n/2) steps.

2 recursive calls are made so T(n)= 2T(n/2)

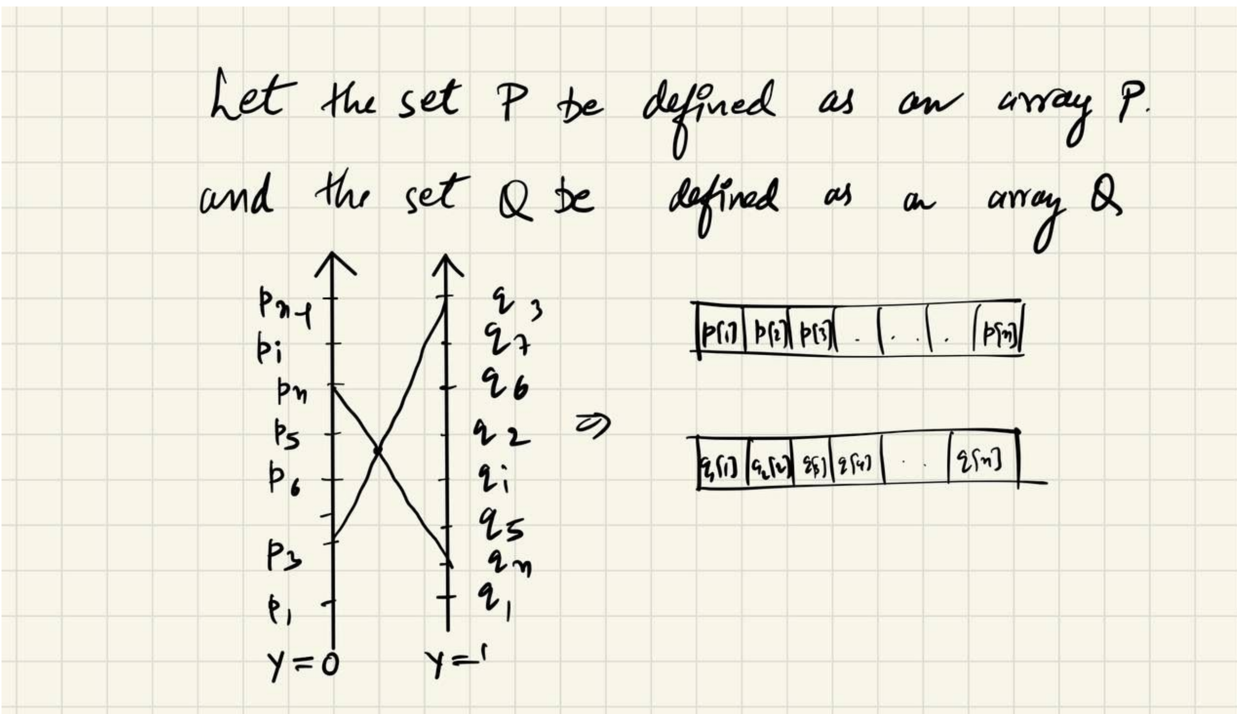
Time taken for cruel algorithm- T(n) = 2T(n/2) {Due to 2 calls to **CRUEL**} + $O(n^{1.585})$ {Due to **UNUSUAL**}

Here, a=2, b=2, d=1.585

Hence, Time Complexity=$O(n^d)=O(n^{1.585})$ (Using Master Theorem)

PROBLEM 3 :

a) The question is same as the one which came in the lab :



Now, $P[i_1]$ $p[i_2]$ $i_1 < i_2$

Let there be 4 elements $q[j_1]$ $q[j_2]$ $j_1 < j_2$

such $p[i_1] = q[j_2]$

$q[j_1] = p[i_2]$

p: | 5 | 4 | 2 | 1 | 3 |

q: | 2 | 5 | 4 | 1 | 3 |

Let us define an order “<” for the array p , such that elements are in the order :
5 < 4 < 2 < 1 < 3 wrt to that order.
If we are mark the elements of P with labels corresponding to the order “<” , let us define as ,
P[1] = 1 , P[2] = 2 similarly , P[n] = n .
If we swap the elements of Q with respect to the label , the elements of the Q will become :
3 , 1, 2, 4 , 5 ,
Now the number of inversions in Q will be the number of Intersections.

We have, $p[i_1]$ $p[i_2]$ $i_1 < i_2$

$q[j_1]$ $q[j_2]$ $j_1 < j_2$

$$p[i_1] = q[j_2] \quad \text{and} \quad p[i_2] = q[j_1]$$

Label for $p[i_1]$ is i_1 (According to our defined relation "<")

& $p[i_2]$ is i_2

we swap $q[j_2]$ with label for $p[i_1]$ & similarly for $q[j_1]$ $\Downarrow$ i_2

so in array

so, $q[j_2] \Rightarrow i_1$

$q[j_1] \Rightarrow i_2$

Array q will look like

$q[j_1]$	$q[j_2]$
j_1	j_2

i_2	i_1
j_1	j_2

$\Rightarrow$ An inversion $\because$ $j_1 < j_2$, but $i_2 > i_1$

PSEUDOCODE:

Size of P = Size of Q = n

```
findIntersections(Arr P , Arr Q, n):
        HashMap <int, int> M;
        For i = 0 to n-1 :
                M[P[i]] = i+1;
          For i = 0 to n-1
                Q[i] = M[Q[i]];
        Return inversionCount(Arr Q,n)
```

TIME COMPLEXITY :

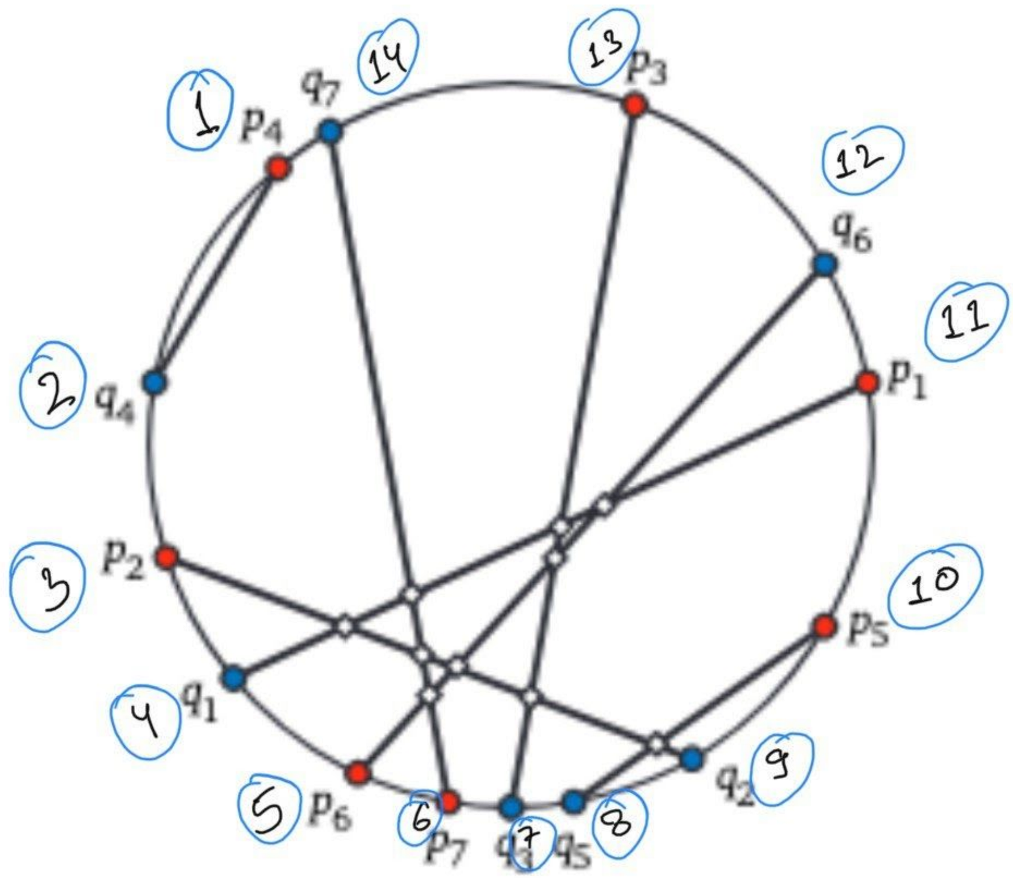
The two for loops in the algorithm take O(n) time , since accessing from HashMap takes O(1) time .

And inversionCount (Arr, n) takes O(nlogn) time.(Already done in class)

Total Time complexity : O(n) + O(nlogn)

Hence, **T(n) = O(nlogn)**

b) Consider the following figure :



Let us cut (mark) the perimeter of the circle at one point. If we make a traversal in any one direction there will be a relative ordering of the points (I have defined the direction as 1 ,2 ,3 ,4 etc.) in the given figure. Now if we try to see each (pi , qi) to be one interval or segment of the perimeter, then we can view the problem as finding the number of overlapping intervals. For example, in the figure if we cut the circle at say p4, then the ordering will be {p4, q4, p2, q1, p6, p7, q3, q5, q2, p5, p1, q6, p3, q7} and now if we number them according to the defined order, we will be able to get segments like , U = {(1, 2),(3, 9),(4, 11),(5, 12),(6, 14),(7, 13),(8, 10)}.
Let us assume we have **n** number of such segments and we can divide them equally into subparts of segments say X and Y. It is implied that the # of overlapping segments in each of these subsets and the first and second elements of the pair are sorted on their own.
For some X[i].second find the greatest first element of a pair in Y which is lesser than X[i].second. Since Y is sorted according to the ‘first’ elements of the pair, we can add the number of intervals whose first element of pair is less than X[i].second.
Now we just need to remove the number of intervals whose second element in the pair will also be lesser than A[i].second due to the reason that such intervals are completely contained inside some other intervals(in this case inside X[i]). For this find the greatest second element in

the pair in Y which is lesser than X[i].second. The remaining merging step is exactly the same as that of merge sort. Since for each X[i], 2 binary searches are required, therefore running time T(n) = 2T(n/ 2) + O(n log n) = O(n log^2 n).

c) Divide and conquer is only used while sorting (and binary search). Sorting the first element of the pair and second element of the pair in the beginning. Steps for counting the number of overlaps are the same. Taking a variable count for counting the number of overlaps. Assume that the interval array is U which is sorted according to the first element of the pair. For a U[i] search the interval U[j] which has the greatest second element of the pair < U[i].second. Add j − i to count. Now find the interval which has the greatest second element of the pair < U[i].second. The position of this second element of the pair in U is k so deduct k − i from count if and only if k − i > 0.

PSEUDOCODE:

```
countIntersection(){

        Sort(Arr, 1, size(Arr))

        C1 = compare_pj_qi(Arr)

        C2 = compare_qi_qj(Arr)

        Return totalInversionCount = C1 - C2

}

Sort(Arr, s, e) {

        if(size(Arr) <=1)

                return Arr;

        m = size(Arr)/2;

        new Array Right = Sort(Arr, m+1, n)

        new Array Left = Sort(Arr, 1, m)

        return Merge(Left, Right);

}

Merge(Light, Right){

        L_size = size(Left)
```

```
        R_size = size(Right)
        result = new Array of size[L_size+R_size]
        L_index = 1, R_index = 1, result_size = 1;
        while (L_index<L_size and R_index<R_size){
                If (Left[L_index]<=Right[R_index]) {
                        Result[result_size] = L[L_index]
                        result_size +=1, L_index+=1
                }
                else {
                        result[result_Size] = R[R_index]
                        result_size+=1, R_index+=1
                }
        }
        while (L_index<L_size){
                result[result_size] = Left[L_index]
                retLength+=1, left index+=1
        }
        while (R_index<R_size){
                result[result_size] = Right[R_index]
                result_size+=1, R_index+=1
        }
        return result
}
/*
```

We have sorted the array

Intersection occurs only when: pi < pj < qi < qj (we already have pi < qi and pj < qj)

Just check for each Point I such that pj < qi and qi < qj.

```
*/
compare_pj_qi(Arr){ // this function is to get count of no points such that pj < qi
    pointsCount = 0
    n=size(Arr)
    for i = 1 to n {
        pointsCount += binarySearch(Arr, qi)
        pointsCount-=1
        // counts and returns the number of pj < qi, 1 is subtracted because pi < qi
    }
    return pointsCount
}
// we got the number the points such that pj < qi
/*
```

We are not done with the intersection because we have just checked the points satisfying pj < qi, now we need to also check the condition qi < qj for all such points. To do so, we can remove those points where qi > qj, this will be sufficient as

removing points qi > qj ⇒ removing points qi > pj {because qi > pi for every point at index i in Arr (shown above)}

So, now we will remove those points, where qi > qj.

```
*/
compare_qi_qj(Arr) {
```

```
    Q_arr= new array of size(Arr)
    for j = 1 to size(Arr) {
        Q_arr[ j ] = Arr[size(Arr) - j][ 2 ]
        // getting q corresponding index to j
    }
    Return totalInversionsCount = sort&countInversion(Q_arr)
}
sort&countInversion(Arr[1...n]) {
    n=size(Arr)
    if(n==1)
        return 0, Arr;
    else{
        X = Arr[1...n/2]
        Y = Arr[n/2+1... n]
        a, P = sort&countInversion(X,n/2)
        b, Q = sort&countInversion(Y,n/2)
        c, R = countAndMerge(P, Q, n)
    }
    return a+b+c, R
}
countAndMerge(P, Q) {
    //count the number of inversions while merging P and Q
    cnt = 0
    Result = new array[size(P) + size(Q)]
```

```
	int Pindex, Qindex, resultIndex
	while(Pindex<size(P) and Qindex<size(Q)){
		if(P[Pindex]<Q[Qindex])){
			ret[resultIndex] = P[Pindex]
			resultIndex+=1, Pindex+=1
		}
		else{
			result[resultIndex] ← Q[Qindex]
			resultIndex+=1, Qindex+=1
			cnt += size(P)-Pindex+1
		}
	}
	while(Pindex<size(P)){
		result[resultIndex] = P[Pindex]
		resultIndex+=1, Pindex+=1
	}

	while(Qindex<size(Q)){
		ret[resultIndex++] = Q[Qindex++]
		resultIndex+=1, Qindex+=1
	}
	return cnt, result
}
binarySearch(Arr, qi, s, e){
```

```
// returns the number of pj such that pj < qi for every index j in Arr

While (s <= e) {

        m = s + (e-s)/2

        If (Arr[m-1][1]<qi<Arr[m][1]){

                return m – 1

        }

        else if (Arr[m+1][1]>qi>Arr[m][1]){

                return m

        }

        else if (Arr[m][1]<qi){

                s = m + 1

        }

        else if (Arr[m][1]>qi){

                e = m - 1

        }

}
```

EXPLANATION OF CODE :

Iinterchange the values of pi and qi if pi>qi will not make a difference. What it does is that the line joining pi and qi does not change and exchanging the contact points of a line segment with the perimeter (end points) will not bring any difference in the count of number of points of intersection. Points are stored as pairs([pi, qi]) in array X. Array A is such that for every index i in A is storing [pi qi], [pj qj], this will be adequate as removing points qi > qj implies removing points like qi > pj (since qi > pi for each point at index i in A as shown above). Remove those points, where qi > qj.Now, the compare_qi_qj() function counts the number of points qi that have lesser rank in relative ordering (or smaller angle) than qj. Henceforth, the absolute difference of in above two counts calculated is the total number of intersections in the circle.

TIME COMPLEXITY:

Time complexity of countAndMerge:

T(n) = O(n) since it only merges and counts in two arrays in linear time

Time complexity of sort&CountInversion:

T(n) = 2T(n/2) + O(n) since it recursively calls the two halves of array so T(n/2) and it calls merge which is O(n)

By master's theorem, asymptotic runtime for this algorithm is nlogn .

Time Complexity of Sort : O(nlogn) since merge sort is used

Time Complexity of binarySearch: O(logn)

Time Complexity of compare_pj_qi :

It runs for linear time (n) in a loop and each time calls binarySearch() which takes O(logn) time

Therefore, time complexity = O(nlogn)

Time Complexity of compare_qi_qj :

Loop runs for n time with each time doing O(1) work thus T=O(n)and it calls sort&countInversion()

sort&countInversion just gives the number of inversions, so, the time complexity is O(nlogn).

Therefore, T(n) = O(n) + O(nlogn) = O(nlogn)

Time Complexity of countIntersection():

1. First sorting of array is done using Sort() = O(nlogn)

2. Second, calling compare_pj_qi(A) = O(nlogn)

3. Third, calling compare_qi_qj(A) = O(nlogn)

Therefore, T(n) = O(nlogn) + O(nlogn) + O(nlogn) = O(nlogn)

Overall time complexity : O(nlogn)