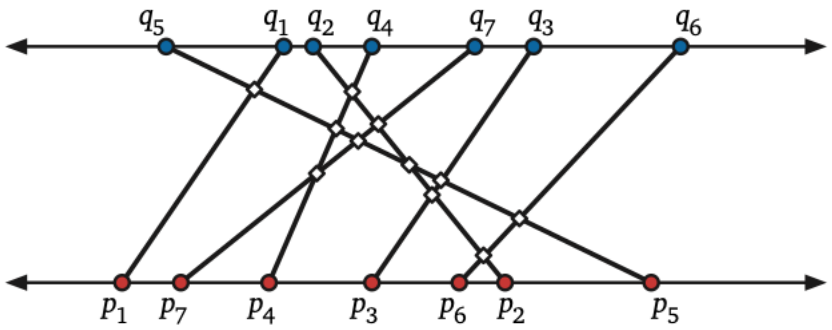
# SOLUTION TO HOMEWORK 1

Jatin Batra, Atishay Jain

## 1. Answer 14 (a)




**Assumption:** All $p_i$ and $q_i$ lie on distinct points. The input is given in the following form : we have two arrays $A, B$. $A[i]$ contains the line number corresponding to the $i$th largest $x$-coordinate on the first parallel line (for $p$'s). Similarly, $B$ is constructed for $q$'s. Now, the input is in the same format as the coding assignment.

**Solution:** We reduce the problem to counting inversions. The idea is to relate the intersection of two line segments to the relative order of their end points on the two parallel lines.

First some notation: let $s_i$ (respectively $t_i$) denote the position of $i$ in the array $A$ (resp. $t_i$ in $B$). More precisely, $A[s_i] = i$ and $B[t_j] = j$. Then we have the following simple but useful observation:

**Observation 1.1.** *Consider two line segments joining $(s_i, t_i)$ and $(s_j, t_j)$ respectively, with $s_i < s_j$. Then, they intersect if and only if $t_i > t_j$.*

Hence we need to count the number of pairs $i, j$ such that $s_i < s_j$ and $t_i > t_j$.

Now we reduce to counting inversions. The idea is to *relabel* $i$ as $s_i$ in the array $B$. More precisely, consider a new array $B'$ such that $B'[t_i] = s_i$. Now, we have the following claim:

**Claim 1.2.** *Line segments joining $(s_i, t_i)$ and $(s_j, t_j)$ intersect if and only if $t_i, t_j$ is an inversion in $B'$.*

*Proof.* A pair $t_i, t_j$ in $B'$ is an inversion whenever $t_i < t_j$ and $B'[t_i] > B'[t_j]$. Noting that $B'[t_i] = s_i$ and $B'[t_j] = s_j$, we are done by observation 1.1. □

Hence, we just output the number of inversions in $B'$ in $O(n \log n)$ time. More formally we have algorithm 1.

**Algorithm:**

**Time Complexity Analysis:**

*Date*: February 23, 2021.

**Algorithm 1** Count intersections for line segments between points on parallel lines

1: **procedure** INTERSECTIONS($A, B$) ▷ $A, B$ have size $n$
2: Initialize an array $S$ of size $n$ as $S[A[i']] \leftarrow i'$ for all $i' \in [n]$ ▷ $S[i]$ stores $s_i$
3: Initialize a new array $B'$ as $B'[i'] \leftarrow S[B[i']]$ ▷ $B'[t_i]$ stores $s_i$
4: **return** the number of inversions in $B'$ by divide and conquer.

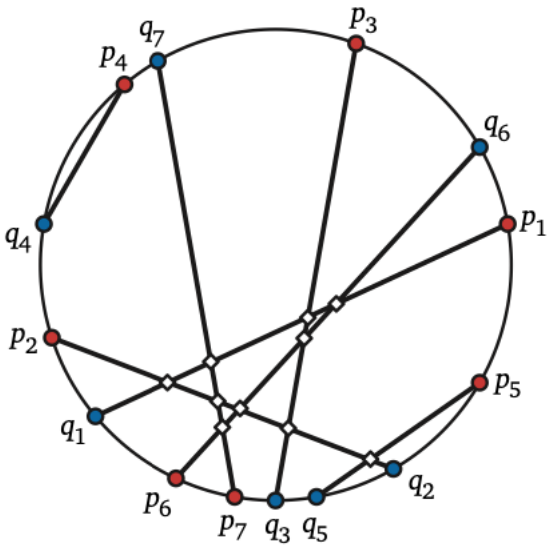
(1) Creating $S, B'$ take linear time.
(2) We can count the number of inversions of an array in $O(n \log n)$ time.

Hence, the time complexity is $O(n \log n)$.

**Correctness:** First some properties of the arrays $S, B'$. $S[i]$ stores the position $s_i$ of $i$ in $A$ by construction (line 2) since $A[i']$ is stored at position $i'$ in $A$. Further, $B'[t_i]$ stores $s_i$ by construction (line 3) since $S[B[t_i]] = S[i]$ (because $B[t_i] = i$ by definition of $t_i$) and $S[i] = s_i$.

Now, by claim 1.2, the number of intersections is just the number of inversions in $B'$.

## 2. Answer 14 (B) and (C)




**Solution:** We reduce the problem to finding the number of *crossings* in a set of intervals on a line.

*Reduction:* Cut the circle at any one point and traverse the circle in any one (say counter-clockwise) direction while creating a relative ordering of the points. Place corresponding copies $p_i', q_i'$ of the points $p_i, q_i$ in this order on a (new) line.

E.g., in the figure if we cut the circle at $p_4$, then the ordered points on the line are

$$(p_4', q_4', p_2', q_1', p_6', p_7', q_3', q_5', q_2', p_5', p_1', q_6', p_3', q_7')$$

A line segment $(p_i, q_i)$ in the circle now corresponds to an *interval* $(p_i', q_i')$ on the line. Intersections of line segments in the circle will correspond to *crossings* of intervals defined as:

**Definition 2.1.** *We say that two intervals $I_1$ and $I_2$ on a line* cross *whenever both $I_1 \setminus I_2 \neq \emptyset$ and $I_2 \setminus I_1 \neq \emptyset$ hold.*

In other words, two intervals cross if they have non-empty intersection and neither contains the other. More precisely, we have

**Observation 2.2.** *Line segments joining $(p_i, q_i)$ and $(p_j, q_j)$ intersect if and only if the intervals $I_i = (p'_i, q'_i)$ and $I_j = (p'_j, q'_j)$ on the line* cross.

Hence our problem reduces to the following following problem for crossings of intervals on a line:

**Problem 2.3.** *Given $n$ points $\mathcal{P}$ on a line and $n/2$ intervals $I_i = (s_i, t_i)$ for $s_i, t_i \in \mathcal{P}$, find the number of pairs $i, j$ such that $I_i$ and $I_j$ cross.*

More precisely, the following algorithm 2 reduces any instance of counting intersections in a circle to an instance of counting crossings of intervals on a line.

**Algorithm 2** Reduction

1: **procedure** $\mathcal{P}$,$S \triangleright \mathcal{P}$ : $n$ points on circle, $S$ : $n/2$ segments joining pairs of points in $\mathcal{P}$
2: For each point in $\mathcal{P}$ find the angle in $[0, 2\pi)$ of the vector joining the point to the center of the circle, with the positive $x$-axis
3: Sort the points by the respective angles, and label each point with its position (in the range 1 to $n$) in the sorted order
4: Update the labels of the points in the segment list $S$
5: Return the segment list $S$ with the updated labels

For example, algorithm 2 on the example above gives the intervals

$$A = (1, 2), (3, 9), (4, 11), (5, 12), (6, 14), (7, 13), (8, 10)$$

Let us now focus on solving problem 2.3.

2.1. **Counting crossings of intervals on a line.** We give two algorithms, one based on directly doing divide and conquer, and the other based on reducing to counting inversions in an array.

2.1.1. *Direct divide and conquer.* We use divide and conquer. First, let us order the intervals by the starting point. More precisely, say $s_i$ denotes the (left) starting point and $t_i$ denotes the (right) ending point of interval $I_i$. Then, order the intervals by $s_i$ and store them in this order in an array $A$.

Now divide the array $A$ into equal halves $L, R$. Then, there are three types of crossings of intervals with respect to $L, R$:

(1) where both intervals lie in $L$
(2) where both intervals lie in $R$
(3) *Split crossings:* where one interval lies in $L$ and the other lies in $R$

We count the crossings of type 1 and 2 by recursing on $L$ and $R$ respectively.

Crossings of type 3 can be counted in linear time if the starting and ending points of intervals in $L, R$ are provided in sorted order. This relies on the following observation:

**Observation 2.4.** *Consider two intervals $I_1$ and $I_2$ such that $s_2 > s_1$. Then, $I_1$ and $I_2$ cross exactly when $t_1 \in I_2$.*


Since any interval of $R$ starts after any interval of $L$, we have to just count the number of pairs $(p, I)$ such that $p$ is an ending point of some interval in $L$ and $I \in R$. This can be done by linearly scanning the combined sorted list of starting and ending points of intervals in $R$, and the ending points of intervals in $L$. More precisely, see algorithm 3.

**Algorithm 3** Finds the number of split crossings of $L, R$

**procedure** SplitCrossings($L^s_{end}, R^s_{start}, R^s_{end}$) ▷ Any interval of $R$ starts after any interval of $L$
  $C \leftarrow$ merged sorted list of $R^s_{start}, R^s_{end}, L^s_{end}$, with each point in $C$ labelled with which list it came from i.e. whether it is a point of $R_{start}, R_{end}$ or $L_{end}$.
  Initialize $count \leftarrow 0, depth \leftarrow 0$
  **for all** point $p \in C$ **do**
    If $p \in L^s_{end}, count \leftarrow count + depth$
    If $p \in R^s_{start}, depth \leftarrow depth + 1$
    If $p \in R^s_{end}, depth \leftarrow depth - 1$
  **end for**
  **return** *count*

Finally we have the overall algorithm 4.

**Algorithm 4** Finds the number of crossings of $A$

**procedure** Crossings($A$) ▷ Intervals in $A$ are sorted by starting point
  *Base case:* If $|A| = 1$, **return** 0
  Partition $A$ equally into $L, R$
  $R^s_{start} \leftarrow$ starting points $R_{start}$ of intervals of $R$ in sorted order.
  $R^s_{end} \leftarrow$ ending points $R_{end}$ of intervals of $R$ in sorted order.
  $L^s_{end} \leftarrow$ ending points $L_{end}$ of intervals of $L$ in sorted order.
  **return** Crossings($L$)+Crossings($R$)+($L^s_{end}, R_{start}, R^s_{end}$)

Algorithm 4 runs in $O(n \log^2 n)$ time due to the sorting. This can be easily improved to $O(n \log n)$ by just sorting the list $A$ at first, and then using the master sorted lists every time to build the needed sorted sub lists. See algorithm 5.

**Algorithm 5** Finds the number of crossings of $A$

**procedure** CROSSINGS($A, A'$) ▷ Intervals in $A$ and $A'$ are sorted by starting point and ending point respectively
  *Base case:* If $|A| = 1$, **return** 0
  Partition $A$ equally into $L, R$ ▷ $k$ is first element in $R$
  Initialize lists $R^s_{start}, R^s_{end}, L^s_{end}$
    ▷ $R^s_{start}, R^s_{end}, L^s_{end}$ contain starting points (resp. ending points and ending points) of intervals in $R, R$ and $L$, in sorted order.
  **for all** intervals $i \in R$ **do**
    Add $s_i$ to $R^s_{start}$
  **end for** ▷
  **for all** intervals $i \in A'$ **do**
    **if** $s_i \geq s_k$ **then**
      Add $t_i$ to $R^s_{end}$
    **else**
      Add $t_i$ to $L^s_{end}$
    **end if**.
  **end for**
  **return** Crossings($L$)+Crossings($R$)+SplitCrossings($L^s_{end}, R_{start}, R^s_{end}$)

**Run-time Analysis for Algorithm 4:** Let $n$ be the size of arrays $L$ and $R$. Then:

(1) SplitCrossing takes $O(n)$ time since sorted lists can be merged in $O(n)$ time (from merge sort), and the loop takes $O(n)$ time since each iteration takes $O(1)$ time
(2) Crossing takes $O(nlogn)$ time at each step as we sort arrays in it

Thus the total work done outside the recursive calls is $O(n \log n)$. Thus if $T(n)$ is the amount of time it takes to solve the problem of size $n$, we get:

$$T(n) = 2T(n/2) + cn \log n$$

Consider the recursion tree where we have a node corresponding to each subproblem which stores the amount of work done outside the recursive calls. The work at the root is $cn \log n$. Now suppose the problem size of some node is $m$. Then, the work at this node is $cm \log m$. Also, the children have problem sizes $m/2$ and their total work is $2 * (cm/2) \log(m/2) \leq cm \log m$. Hence, the total work of the children of any node is at most the work of the node. Therefore, the work at each level is at most the work of the previous level, and hence at most the work at the root. Since there are $\log n$ levels, the total work is $O(n \log^2 n)$.

**Run-time Analysis for Algorithm 5:** Let $n$ be the size of arrays $L$ and $R$. Then

(1) SplitCrossing takes $O(n)$ time.
(2) Crossing $O(n)$ time as we just iterate through $A$ and $A'$.

Hence the recurrence is:

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

We know this comes out to be $T(n) = O(nlogn)$

**Correctness:** Proving by the way of strong induction. We will induct of the size of array.
$P(n)$ : The algorithm returns the correct solution for all $n$.

*Base Case* : $n = 1$

The number of intersections is zero in this case which is trivially true.

Thus $P(1)$ is true.

*Induction Hypothesis* : $P(k)$ is true where $k < n$.

*To prove* : $P(n)$ is true.

The crossings in $A$ are of three types

(1) Crossings in $L$ : correctly returned by the algorithm on $L$ by induction hypothesis since $|L| = n/2 < n$
(2) Crossings in $R$ : similarly as for $L$
(3) Split crossings : returned by algorithm 3

Hence, we just need to show that algorithm 3 is correct.

**Lemma 2.5.** *Algorithm 3 outputs the number of split crossings in $L, R$.*

*Proof.* Observation 2.4 shows that the number of split crossings is exactly the number of pairs $(p, I)$ such that $p \in L_{end}$ and $I \in R$, which we now focus on.

At the start of any iteration of the loop in line 4, we have iterated through a subset of points of $C$. This subset contains points of $L_{end}, R_{start}, R_{end}$.

The key is the following *loop invariant.*

**Claim 2.6.** *At the start of any iteration of the loop, if $p$ is the next point in $C$, depth is the number of intervals $i \in R$ such that $p \in (s_i, t_i]$.*

*Proof.* At the start of the first iteration, the invariant trivially holds as the value of *depth* is zero.

Assuming that the invariant holds till the $j^{th}$ iteration, we will show that it also hold for the $j + 1^{th}$ iteration.

We first show that if the invariant held at the start of iteration $j$, it continues to hold at the start of iteration $j+1$. We can see this by making three cases for the element $c[j]$ in the $j^{th}$ iterations:

(1) $C[j] \in L_{end}$: The number of intervals $i \in R$ such that $C[j] \in (s_i, t_i]$ is the same as the number for which $C[j-1] \in (s_i, t_i]$.
(2) $C[j] \in R_{start}$ : The number of intervals $i \in R$ such that $C[j] \in (s_i, t_i]$ is exactly one more than the number for $C[j-1] \in (s_i, t_i]$. Also, the depth variable is incremented by 1.
(3) $C[j] \in R_{end}$ : The number of intervals $i \in R$ such that $C[j] \in (s_i, t_i]$ is exactly one less than the number in $C[j-1] \in (s_i, t_i]$. Also, the depth variable is decremented by 1.

Hence since the loop invariant held at the previous step, our algorithm sets *depth* to $|\{C[j] \in (s_i, t_i]\}|$, which proves the second condition of the claim.

□

Hence, at the end of the algorithm, *count* contains the number of pairs $(p, I)$ for the points $p \in L_{end}$ and $I \in R$. This is because every time we encounter a $p$ in $C$, such that $p \in L_{end}$, we increment the *count* by the depth variable which is just the number of intervals $I \in R$ such that $p \in I$ by claim 2.6.

□

Since algorithm 3 returns number of SpiltCrossing correctly, P(n) is true. Thus $P(n-1) \implies P(n)$. Hence by principle of mathematical induction, $P(n)$ is true for all $n \in \mathbb{N}$.

2.1.2. *Reduction to inversions.* We now give another algorithm 6 by reducing problem 2.3 to counting inversions in an array.

**Algorithm**

**Algorithm 6**

1: **procedure** InvCross($A$) ▷ $A$ : list of intervals over $n$ points, each a starting-ending point tuple
2: sort the intervals by starting point.
3: Label $\mathcal{P}$ as follows: both the starting and ending points of every interval are labelled by its position in the sorted order.
4: $L \leftarrow$ labels of points in $\mathcal{P}$ ordered from left to right
5: $B \leftarrow$ labels of ending points
6: **return** (number of inversions in $L$) - 2*(number of inversions in $B$)

For example, in the previous example the array $L$ would be

$$L = (1, 1, 2, 3, 4, 5, 6, 7, 2, 7, 3, 4, 6, 5$$

and the array $B$ would be

$$B = (1, 2, 7, 3, 4, 6, 5)$$

**Correctness:** Let us refer to the interval whose starting and ending points are labelled $i$, simply as interval $i$.

The following claim is the key to everything:

**Claim 2.7.** *The number of inversions in $L$ is the number of crossings of intervals over $\mathcal{P}$ plus twice the number of inversions in $B$.*

*Proof.* Consider two intervals $i$ and $j$ with $i < j$. Then let us make cases based on the relative positions of the starting and ending points of $i, j$ in $L$:

(1) $i, j, i, j$: $i, j$ contribute one crossing, one inversion to $L$, and zero inversions to $B$
(2) $i, j, j, i$: $i, j$ contribute zero crossings, two inversions to $L$, and one inversion to $B$
(3) $i, i, j, j$: $i, j$ contribute zero crossings and zero inversions to both $L$ and $B$

Notice that these are the only possibilities because the intervals were sorted (and labelled) by their starting points. Hence, the number of inversions in $L$ is the number of crossings of intervals over $\mathcal{P}$ (type 1) plus twice the number of inversions in $B$ (type 2).

□

**Time complexity Analysis:**

(1) Creating the lists $L, B$ can be done in $O(n \log n)$ time. There are many ways to do this, for instance use a balanced search tree to store the key value mappings.
(2) Inversions take $O(n \log n)$ time to find using divide and conquer

Hence, the total time complexity is $O(n \log n)$.